Università di Pisa
Dipartimento di Informatica

Programmazione in JavaScript

Vincenzo Ambriola

Versione 9.0 ¤ 30 gennaio 2019
Anno Accademico 2018/19

Prefazione

Per chi si avvicina alla programmazione gli ostacoli da superare sono tanti: un nuovo linguaggio (artificiale) da imparare, strumenti di sviluppo da provare per capirne la logica di funzionamento, esercizi da risolvere per apprendere i concetti di base e, successivamente, quelli più avanzati. Ci sono molti modi per insegnare a programmare e ogni docente, nel tempo, ha trovato il suo.

Questo libro è rivolto agli studenti del primo anno del Corso di laurea in *Informatica umanistica* che seguono *Elementi di programmazione* (uno dei due moduli dell'insegnamento di *Fondamenti teorici e programmazione*) e *Programmazione* (uno dei due moduli di *Progettazione e programmazione web*). L'approccio adottato si basa sull'introduzione graduale dei concetti di *JavaScript*, un linguaggio di programmazione ampiamente usato per la programmazione web.

Questo libro non è un manuale di JavaScript. Il linguaggio è troppo complesso per insegnarlo a chi non ha mai programmato e, soprattutto, di manuali di questo tipo ce ne sono tanti in libreria. Il libro presenta gli aspetti più importanti di JavaScript, necessari ma, soprattutto, sufficienti per la soluzione dei problemi proposti. Il lettore interessato agli aspetti più avanzati di JavaScript e al suo uso professionale è invitato a continuare lo studio e la pratica di questo linguaggio. I risultati supereranno di gran lunga le aspettative.

Con l'evoluzione dei browser, JavaScript ha subito numerose modifiche, acquisendo progressivamente nuove funzionalità. Per evitare possibili confusioni, il libro fa riferimento alla versione del linguaggio conforme allo standard *ECMAScript 2017* e accettata dal browser *Firefox* (versione aggiornata alla data del libro).

1.1 Struttura del libro

Il libro è strutturato in due parti: la prima presenta gli elementi di programmazione necessari per risolvere semplici problemi su numeri e testi; la seconda affronta il tema della programmazione web. Ogni parte è strutturata in capitoli, dedicati a un aspetto della programmazione. Alcuni capitoli si concludono con esercizi che il lettore è invitato a risolvere non solo con carta e matita ma mediante un calcolatore. Gli strumenti necessari sono alla portata di tutti: un bro-

wser di nuova generazione è più che sufficiente. La soluzione di alcuni esercizi proposti è riportata al termine delle due parti.

Nel libro sono state usate le seguenti *convenzioni tipografiche*, per facilitare la lettura dei programmi presentati:

- il *corsivo* è usato per indicare la prima volta che un termine rilevante compare nel libro; l'indice analitico contiene l'elenco di questi termini, con l'indicazione della pagina in cui sono introdotti;
- la sintassi di JavaScript e gli esempi sono riportati all'interno di un riquadro colorato.

1.2 Ringraziamenti

La prima parte di questo libro nasce da una lunga collaborazione con *Giuseppe Costa*, coautore di *4 passi in JavaScript*. Senza le sue preziose indicazioni non avrei capito e apprezzato le insidie e la bellezza di un linguaggio di programmazione complesso come JavaScript. La seconda parte è stata scritta seguendo i consigli e i suggerimenti di *Maria Simi*, profonda conoscitrice del web, della sua storia e delle tante tecnologie ad esso collegate.

Indice

Parte prima

Elementi di programmazione

2 Linguaggi e grammatiche

Quando si parla di *linguaggio* viene subito in mente il linguaggio che usiamo quotidianamente per comunicare con chi ci circonda. In realtà, il concetto di linguaggio è molto più generale.

Possiamo individuare due categorie di linguaggi: *naturali* e *artificiali*. I primi sono linguaggi intrinsecamente ambigui, perché il significato delle loro *parole* dipende dal contesto in cui sono inserite. I linguaggi naturali sono inoltre caratterizzati dal fatto di mutare con l'uso, per l'introduzione di neologismi e di parole provenienti da altri linguaggi (per l'italiano è il caso dei termini stranieri o delle espressioni dialettali).

Un esempio di frase ambigua è: *Ho analizzato la partita di calcio*. La parola "calcio" può significare "lo sport del calcio" nella frase *Ho analizzato la partita di calcio dell'Italia* o "il minerale calcio" nella frase *Ho analizzato la partita di calcio proveniente dall'Argentina*.

A differenza dei linguaggi naturali, i linguaggi artificiali hanno regole e parole che non cambiano con l'uso e il cui significato non dipende dal contesto. A questa famiglia appartengono i linguaggi utilizzati per descrivere le operazioni da far compiere a macchine o apparecchiature. Tali descrizioni non possono essere ambigue perché una macchina non deve decidere autonomamente tra più possibilità di interpretazione.

Un esempio di linguaggio artificiale è quello usato per comandare un videoregistratore: una frase del linguaggio è una qualunque sequenza di *registra*, *riproduci*, *avanti*, *indietro*, *stop*. Le parole hanno un significato preciso e indipendente dal contesto in cui sono usate.

L'informatica ha contribuito notevolmente alla nascita di numerosi linguaggi artificiali, i cosiddetti *linguaggi di programmazione*, usati per la scrittura di *programmi* eseguibili da *calcolatori elettronici*. Questo libro è dedicato allo studio di uno di essi, il linguaggio JavaScript.

2.1 Alfabeto, linguaggio

Un *alfabeto* è formato da un insieme finito di *simboli*. Ad esempio, l'alfabeto $\mathcal{A} = \{a, b, c\}$ è costituito da tre simboli.

Una *frase* su un alfabeto è una sequenza di lunghezza finita formata dai simboli dell'alfabeto. Ad esempio, con l'alfabeto $\mathcal{A}$ definito in precedenza si possono formare le frasi *aa*, *abba*, *caab*.

Dato un alfabeto $\mathcal{A}$, l'insieme di tutte le frasi che si possono formare usando i suoi simboli è infinito, anche se ogni frase è di lunghezza finita. Per semplicità questo insieme è chiamato *insieme delle frasi di* $\mathcal{A}$. Sempre usando l'esempio precedente, è possibile formare l'insieme delle frasi di $\mathcal{A}$ ma non è possibile riportarlo in questo libro perché, come già detto, la sua lunghezza è infinita. Ciononostante, è possibile mostrarne una parte finita, usando come artificio i puntini di sospensione per indicare la parte infinita: {*a*, *b*, *c*, *aa*, *ab*, *ac*, *ba*, *bb*, *bc*, *ca*, *cb*, *cc*, ...}.

Dato un alfabeto $\mathcal{A}$, un *linguaggio* $\mathcal{L}$ *su* $\mathcal{A}$ è un sottoinsieme delle frasi di $\mathcal{A}$. Questa definizione è diversa da quella data in precedenza: non tutte le frasi di $\mathcal{A}$ sono, infatti, anche frasi di $\mathcal{L}$.

Abbiamo fatto riferimento a "un" linguaggio e non "del" linguaggio su un alfabeto perché su un dato alfabeto può esistere più di un linguaggio. Prendiamo come esempio il linguaggio italiano e quello inglese: entrambi si basano sull'*alfabeto latino* (detto anche *alfabeto romano*), ma sono costituiti da insiemi diversi di frasi.

I like walking on the grass è una frase del linguaggio inglese, ma non lo è di quello italiano. *Mi piace camminare sull'erba* è una frase del linguaggio italiano, ma non lo è di quello inglese, anche se il significato è lo stesso.

Sdksfdk skjfsfkj sdkfsakjfd, invece, non è una frase né del linguaggio italiano né di quello inglese, nonostante appartenga all'insieme delle frasi sull'alfabeto latino.

2.2 Grammatiche

Un linguaggio ha due aspetti importanti:

- *sintassi*: le regole per la formazione delle frasi del linguaggio (ad esempio, la grammatica italiana);
- *semantica*: il significato da attribuire ad ogni frase del linguaggio (ad esempio, tramite l'analisi logica delle frasi, che permette di individuare il soggetto di una data azione, l'oggetto e così via).

Esistono *formalismi* che definiscono i due aspetti di un linguaggio. In questo libro presentiamo un formalismo per la definizione della sintassi. Non trattiamo, invece, gli aspetti relativi alla definizione formale della semantica.

Una *grammatica* è un formalismo che permette di definire le regole per la formazione delle frasi di un linguaggio. Una grammatica è quindi un *metalinguaggio*, ovvero un linguaggio che definisce un altro linguaggio.

Una grammatica è formata da:

- *simboli terminali*: rappresentano gli *elementi sintattici* del linguaggio che, di norma, coincidono con l'alfabeto del linguaggio;
- *simboli non-terminali* o *metasimboli*: rappresentano le *categorie sintattiche* del linguaggio;
- *produzioni*: definiscono le relazioni tra i simboli terminali e i non-terminali; mediante la loro applicazione si ottengono le frasi del linguaggio.

Un linguaggio $\mathcal{L}$ le cui frasi sono ottenute applicando le produzioni della grammatica $\mathcal{G}$ si dice *generato da* $\mathcal{G}$.

2.3 Backus-Naur Form

La *Backus-Naur Form* (*BNF*) è uno dei formalismi più usati per definire una grammatica. In accordo con il concetto di grammatica, la BNF prevede la definizione di un insieme di simboli non-terminali (di solito racchiusi tra parentesi angolari) e di un insieme di simboli terminali (di solito indicati in corsivo o in grassetto per distinguerli dai simboli non-terminali). Inoltre, deve essere indicato un *simbolo iniziale* che appartiene all'insieme dei simboli non-terminali. La funzione di questo simbolo sarà chiarita nel seguito.

Le produzioni per la formazione delle frasi sono di due tipi:

- $X ::= S$

  si legge "X può essere sostituito con S" o "X produce S". X è un simbolo non-terminale, S è una sequenza di simboli terminali e non-terminali.
- $X ::= S_1 \mid S_2 \mid \ldots \mid S_n$

  si legge "X può essere sostituito con $S_1, S_2, \ldots,$ o S_n". Anche in questo caso X è un simbolo non-terminale, mentre $S_1, S_2, \ldots, S_n$ sono sequenze di simboli terminali e non-terminali.

Il linguaggio $\mathcal{L}$ generato da una grammatica $\mathcal{G}$ definita in BNF è l'insieme di tutte le frasi formate da soli simboli terminali, ottenibili partendo dal simbolo iniziale e applicando in successione le produzioni di $\mathcal{G}$.

Applicare una produzione a una sequenza di simboli significa sostituire (nella sequenza) un'occorrenza del simbolo non-terminale definito dalla produzione con una sequenza della sua parte sinistra.

Per rendere concreti i concetti presentati finora, mostriamo la grammatica $\mathcal{G}_1$, definita in BNF:

- A = {*a*, *b*} alfabeto
- N = {<*S*>, <*A*>, <*B*>} simboli non-terminali
- I = <*S*> simbolo iniziale
- <*S*> ::= <*A*><*B*> | <*B*><*A*> produzione di <*S*>
- <*A*> ::= *a* | *aa* produzione di <*A*>
- <*B*> ::= *b* | *bb* produzione di <*B*>

Il linguaggio generato da $\mathcal{G}_1$ è {*ab*, *aab*, *abb*, *aabb*, *ba*, *bba*, *baa*, *bbaa*}. Si noti che, in questo esempio, il linguaggio è finito perché la grammatica genera esattamente otto frasi.

Una grammatica leggermente più complicata è $\mathcal{G}_2$:

- A = {*a*, *b*}, alfabeto
- N = {<*S*>, <*A*>, <*B*>} simboli non-terminali
- I = <*S*> simbolo iniziale
- <*S*> ::= <*A*><*B*> produzione di <*S*>
- <*A*> ::= *a* | *a* <*A*> produzione di <*A*>
- <*B*> ::= *b* | *b* <*B*> produzione di <*B*>

Il linguaggio infinito generato da $\mathcal{G}_2$ è {*ab*, *aab*, *abb*, *aabb*, *aaab*, *aaabb*, *aaabbb*, *abbb*, *aabbb*, *aaabbb*, ...}.

2.4 Sequenze di derivazione

Il *processo di derivazione* di una frase si può rappresentare tramite una *sequenza di derivazione,* formata da una successione di frasi intermedie, costruite a partire dal simbolo iniziale, con l'indicazione della produzione usata per passare da una frase alla successiva. Ogni frase, tranne l'ultima, è formata da simboli terminali e non-terminali. L'ultima frase è formata esclusivamente da simboli terminali e, pertanto, appartiene al linguaggio generato dalla grammatica.

A titolo di esempio, consideriamo la sequenza di derivazione della frase *aab* appartenente al linguaggio generato da $\mathcal{G}_2$:

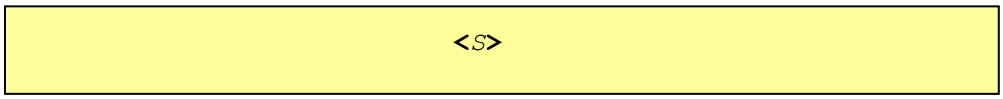
```
<S>
=> Produzione <S> ::= <A> <B>
<A> <B>
=> Produzione <A> ::= a | a <A> seconda opzione
a <A> <B>
=> Produzione <A> ::= a | a <A> prima opzione
a a <B>
=> Produzione <B> ::= b | b <B> prima opzione
a a b
```

2.5 Alberi sintattici

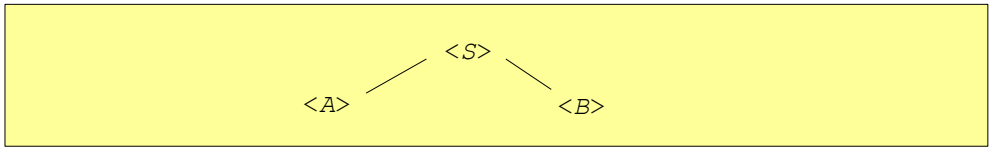
Un metodo alternativo per rappresentare il processo di derivazione si basa sugli *alberi sintattici*. Per costruire l'albero sintattico di una frase si parte dal simbolo iniziale della grammatica, che ne costituisce la *radice*. Successivamente, per ogni *foglia* che è un simbolo non-terminale, si sceglie una produzione da applicare sostituendo la foglia con un *nodo* e generando tanti figli quanti sono i simboli terminali e non-terminali dell'opzione scelta. Le foglie possono essere simboli non-terminali solo nelle fasi intermedie di costruzione. Il procedimento termina quando tutte le foglie sono simboli terminali.

A differenza del precedente, questo metodo fornisce una visione d'insieme del processo di derivazione ed evidenzia il simbolo non-terminale sostituito ad ogni passo. Anche l'albero sintattico per una data frase viene costruito applicando in sequenza le produzioni a partire dal simbolo iniziale, ma la sua struttura finale non dipende dall'ordine di applicazione delle produzioni. Infine, un albero sintattico è caratterizzato dal fatto che la radice e tutti i nodi sono simboli non-terminali e le foglie sono simboli terminali.

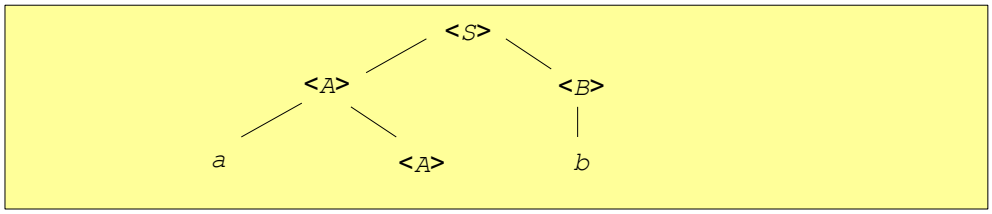
Ecco un esempio di costruzione dell'albero sintattico per la frase *aab* appartenente al linguaggio generato da G_2:



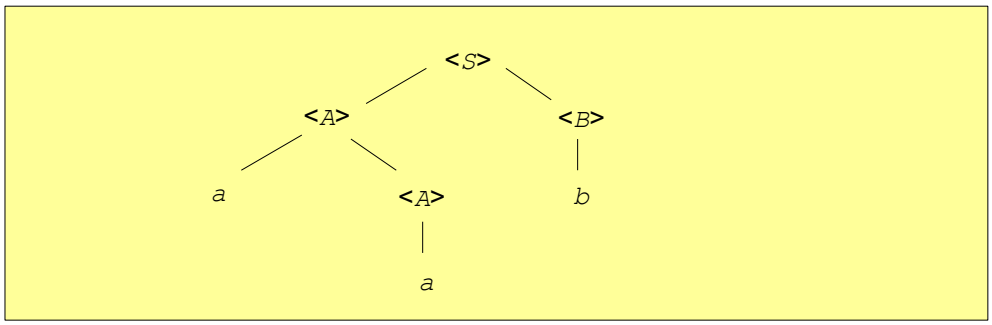
Applichiamo la produzione <*S*> ::= <*A*> <*B*> al nodo radice dell'albero sintattico. L'applicazione della produzione genera due nuovi nodi.



Applichiamo la produzione <*A*> ::= *a* | *a* <*A*> (seconda opzione) al nodo <*A*> e la produzione <*B*> ::= *b* | *b* <*B*> (prima opzione) al nodo <*B*>.



Infine applichiamo la produzione <*A*> ::= *a* | *a* <*A*> (prima opzione) al nodo <*A*>. La frase *aab* è formata dai simboli terminali presenti sulle foglie dell'albero sintattico, letti da sinistra a destra.



3 Programmi, comandi e letterali

JavaScript è un linguaggio di programmazione, in particolare è un linguaggio di *script*, cioè un linguaggio per definire programmi eseguibili all'interno di altri programmi (chiamati *applicazioni*).

JavaScript è usato principalmente per la definizione di programmi eseguibili nei *browser*, applicazioni per la visualizzazione di *pagine web*. In questo ambito, JavaScript permette di rendere interattive queste pagine affinché mostrino contenuti diversi in base alle azioni dell'utente o a situazioni contingenti, come l'ora corrente o il tipo di browser utilizzato. Mediante JavaScript si possono aggiungere effetti grafici e accedere alle funzionalità del browser (lanciare una stampa, aprire una nuova *finestra*, ridimensionare o spostare sullo *schermo* una finestra di visualizzazione).

Nella prima parte del libro JavaScript è presentato esclusivamente come linguaggio di programmazione, evitando di descriverne il suo uso come linguaggio di script. Questa scelta è motivata dalla necessità di introdurre gli aspetti di base del linguaggio prima di quelli più complessi, necessari per affrontare gli argomenti trattati nella seconda parte. Per rendere concreta la presentazione del linguaggio, il libro fa riferimento a un *ambiente di programmazione* chiamato *EasyJS*[1]. Questo ambiente permette di definire un programma JavaScript, eseguirlo e visualizzare il risultato dell'esecuzione.

3.1 Programma

In JavaScript un programma è una *sequenza* di *comandi*. Un comando può essere una *dichiarazione*, un *comando semplice* o un *comando composto*.

```
<Programma> ::= <Comandi>

<Comandi>   ::= <Comando>
              |  <Comando> <Comandi>

<Comando>   ::= <Dichiarazione>
              |  <ComandoSemplice>
              |  <ComandoComposto>
```

1 http://www.di.unipi.it/~ambriola/pw/radice.htm

L'esecuzione di un programma consiste nell'esecuzione della sua sequenza di comandi. I comandi sono eseguiti uno dopo l'altro, nell'ordine con cui compaiono nella sequenza. Questo comportamento è tipico dei *linguaggi di programmazione imperativi*, classe alla quale appartiene JavaScript.

Il *punto e virgola* indica la fine di una dichiarazione o di un comando semplice. Anche se è buona norma terminare, quando previsto, un comando con un punto e virgola, in JavaScript è possibile ometterlo se il comando è interamente scritto su una riga e sulla stessa riga non ci sono altri comandi. Il *ritorno a capo*, normalmente ignorato, in questo caso funge da *terminatore di comando*.

JavaScript è un linguaggio *case sensitive* perché fa distinzione tra lettere maiuscole e minuscole. Gli *spazi bianchi* e le *interruzioni di riga* servono per migliorare la leggibilità dei programmi. Si possono utilizzare per separare gli elementi del linguaggio, ma non possono essere inseriti al loro interno.

Un *commento* è formato da una o più righe di testo. In JavaScript ci sono due tipi di commenti:

- su una riga sola, introdotti da "//"
- su più righe, introdotti da "/*" e chiusi da "*/".

```
// questo è un commento su una riga
/* questo è un commento
   su due righe */
```

3.2 Letterali numerici e logici

In JavaScript un *letterale* rappresenta un *valore numerico* o un *valore logico* (o *booleano*). I valori numerici appartengono al *tipo primitivo* dei numeri, quelli logici al tipo primitivo dei booleani.

```
<Letterale> ::= <Numero>
              |  <Booleano>

<Numero>    ::= <Intero>
              |  <Intero>.<Cifre>
              |  <Intero>E<Esponente>
              |  <Intero>.<Cifre>E<Esponente>

<Intero>    ::= <Cifra>
              |  <CifraNZ> <Cifre>

<Cifra>     ::= 0 | <CifraNZ>

<CifraNZ>   ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<Cifre>     ::= <Cifra>
             |  <Cifra> <Cifre>

<Esponente> ::= <Intero>
             |  + <Intero>
             |  - <Intero>

<Booleano>  ::= true | false
```

Alcuni esempi di *letterali numerici* sono i seguenti:

- *12*
- *159.757*
- *23E5*
- *51E+2*
- *630E−6*
- *2.321E4*
- *3.897878E+7*
- *9.7777E−1*

Il letterale *2* rappresenta il valore *due*, il letterale *159.757* rappresenta il valore *centocinquantanove virgola settecentocinquantasette*. Gli altri letterali, espressi in *rappresentazione esponenziale*, possono essere espressi anche in *rappresentazione decimale*, tenendo presente che l'*esponente* positivo sposta la virgola verso destra e quello negativo verso sinistra:

- *23E5* equivale a *2 300 000*
- *51E+2* equivale a *5 100*
- *630E−6* equivale a *0.000630*
- *2.321E4* equivale a *23 210*
- *3.897878E+7* equivale a *38 978 780*
- *9.7777E−1* equivale a *0.97777*

Il letterale *23E5* rappresenta il valore *duemilionitrecentomila*, il letterale *51E+2* rappresenta il valore *cinquemilacento*, mentre il letterale *630E−6* rappresenta il valore *zero virgola zerozerozeroseicentotrenta*. Il valore rappresentato dagli altri letterali è facilmente desumibile.

I *letterali logici* sono due e rappresentano i due valori logici *vero* e *falso*. Il letterale *true* rappresenta il valore logico *vero*, il letterale *false* rappresenta il valore logico *falso*.

3.3 Letterali stringa

Una *stringa* è una sequenza di *caratteri* ed è il tipo di dato usato in JavaScript per rappresentare testi. Una stringa che appare esplicitamente in un programma è chiamata *letterale stringa* ed è una sequenza (anche vuota) di caratteri racchiusi tra *apici singoli* o *apici doppi*. I caratteri sono tutti i *caratteri stampabili*: le lettere alfabetiche minuscole e maiuscole, le *cifre numeriche* (da non confondere con i numeri), i *segni di interpunzione* e gli altri simboli che si trovano sulla *tastiera* di un calcolatore (e qualcuno di più).

```
<Letterale>      ::= <Stringa>

<Stringa>        ::= ""
                   |  "<CaratteriStr>"
                   |  ''
                   |  '<CaratteriStr>'

<CaratteriStr>   ::= <CarattereStr>
                   |  <CarattereStr><CaratteriStr>

<CarattereStr>   ::= <Lettera>
                   |  <Cifra>
                   |  <Speciale>

<Lettera>        ::= a | b | c | d | e | f | g | h | i
                   |  j | k | l | m | n | o | p | q | r
                   |  s | t | u | v | w | x | y | z
                   |  A | B | C | D | E | F | G | H | I
                   |  J | K | L | M | N | O | P | Q | R
                   |  S | T | U | V | W | X | Y | Z

<Speciale>       ::= Space[2]
                   |  ! | " | # | $ | % | & | ' | ( | )
                   |  * | + | , | - | . | / | : | ; | <
                   |  = | > | ? | @ | [ | \ | ] | ^ | _
                   |  ` | { | | | } | ~
```

In un programma, i letterali stringa devono essere scritti su una sola riga. Per inserire ritorni a capo, *tabulazioni*, particolari caratteri o informazioni di *formattazione* si utilizza la *barra diagonale decrescente*: '\'[3] chiamata anche *carattere di quotatura*. La coppia formata da *backslash* e da un altro carattere è chiamata *sequenza di escape*. Le principali sequenze di *escape* sono:

- \n: *nuova riga*;

2 Carattere di spaziatura.
3 In inglese questo carattere si chiama *backslash*.

- \r: ritorno a capo;
- \t: *tabulazione orizzontale*;
- \': apice singolo;
- \": apice doppio;
- \\: *backslash*.

3.4 Comando di stampa

Il primo comando semplice che prendiamo in considerazione è il *comando di stampa*[4]:

```
<ComandoSemplice> ::= print(<Letterale>);
```

L'esecuzione di un comando di stampa ha l'effetto di scrivere sulla finestra inferiore di *EasyJS* il letterale racchiuso tra *parentesi tonde*.

Ad esempio, il seguente programma stampa il valore di tre letterali.

```
print(12);
print(true);
print("alfa");
```

4 Come vedremo nel seguito, il comando di stampa è un'invocazione di funzione. Per semplicità, in questo capitolo è trattato come un comando semplice.

4 Espressioni

Un'*espressione* rappresenta un calcolo che restituisce un valore. In JavaScript, come in tutti i linguaggi di programmazione, le espressioni possono essere *semplici* o *composte*. Un letterale è un'espressione semplice, il valore del letterale coincide con il valore dell'espressione, il tipo dell'espressione coincide con il tipo del suo valore. Un'espressione composta si ottiene combinando una o più espressioni mediante un *operatore*, che rappresenta l'*operazione* da effettuare sulle espressioni, dette *operandi*.

```
<Espressione> ::= <Letterale>
                | (<Espressione>)
                | <UnOp> <Espressione>
                | <Espressione> <BinOp> <Espressione>
<UnOp>        ::= -  | +  | !
<BinOP>       ::= -  | +  | *  | /  | %
                | && | || |
                | <  | <= | >  | >= | == | !=
```

L'introduzione delle espressioni richiede la modifica della sintassi del comando di stampa.

```
<ComandoSemplice> ::= print(<Espressione>);
```

4.1 Operatori

Gli operatori che hanno un unico operando sono detti *unari*. Gli operatori unari sono:

- – : *segno negativo*
- + : *segno positivo*
- ! : *negazione*

L'operatore di segno negativo ha un operando il cui valore è un numero. L'operatore cambia il segno del valore. Anche l'operatore di segno positivo ha un operando il cui valore è un numero. In questo caso, però, il segno del valore non è modificato. Questi due operatori sono detti *numerici*, perché il loro risultato è un numero.

L'operatore di negazione ha un operando il cui valore è un booleano. Se il valore dell'operando è *true*, il risultato è *false*, se il valore è *false* il risultato è *true*. Questo operatore è detto *booleano*, perché il suo risultato è un booleano.

Gli operatori che hanno due operandi sono detti *binari*. Anche gli operatori binari possono essere numerici o booleani. Quelli numerici sono:

- – : *sottrazione*
- + : *addizione*
- * : *moltiplicazione*
- / : *divisione*
- % : *modulo*

I primi quattro operatori (sottrazione, addizione, moltiplicazione, divisione) sono quelli usualmente conosciuti e utilizzati per effettuare i calcoli aritmetici. Il modulo è un'operazione su numeri interi, che restituisce il resto della divisione intera del primo operando per il secondo. Il valore dell'espressione *5%2* è *1*, cioè il resto della divisione intera di *5* per *2*.

Gli operatori binari booleani si dividono in due gruppi: quelli che hanno due operandi booleani, quelli che hanno due operandi dello stesso tipo. Gli operatori che appartengono al secondo gruppo sono anche detti *operatori di confronto*.

L'elenco dei primi operatori è il seguente:

- && : *congiunzione*
- || : *disgiunzione*

L'operatore di congiunzione restituisce *true* solo se i suoi operandi valgono *true*, *false* altrimenti. Se il primo operando vale *false* il secondo non è valutato.

L'operatore di disgiunzione restituisce *true* se almeno uno dei suoi operandi vale *true*, *false* altrimenti. Se il primo operando vale *true* il secondo non è valutato.

Gli operatori di confronto sono:

- == : *uguaglianz*a
- != : *disuguaglianza*
- > : *maggiore*
- >= : *maggiore o uguale*
- < : *minore*
- <= : *minore o uguale*

Come già detto, gli operandi degli operatori di confronto devono essere dello stesso tipo. Nel caso dell'uguaglianza, l'operatore restituisce *true* se i valori dei

due operandi sono uguali, *false* altrimenti. L'operatore di disuguaglianza si comporta in maniera simmetricamente opposta. L'operatore maggiore restituisce *true* se il valore del primo operando è maggiore di quello del secondo, *false* altrimenti. L'operatore di maggiore o uguale è leggermente diverso, in quanto restituisce *true* se il valore del primo operando è maggiore o uguale a quello del secondo, *false* altrimenti. Gli altri due operatori si comportano in maniera simmetricamente diversa.

La *relazione di ordinamento* per i numeri è quella usuale, mentre per i booleani si assume che *true* sia maggiore di *false*. Per le stringhe vale la *relazione di ordinamento lessicografico*, basata sulla *relazione di ordinamento alfanumerico*, in cui le lettere minuscole e quelle maiuscole sono ordinate secondo l'alfabeto inglese, tutte le maiuscole precedono le minuscole, le cifre numeriche sono ordinate secondo il loro valore, tutte le cifre numeriche precedono tutte le lettere. L'ordinamento lessicografico tra due stringhe stabilisce che *a* è minore di *b* se il primo carattere di *a* è minore del primo carattere di *b*, secondo la relazione di ordinamento alfanumerico. Se i due caratteri sono uguali si passa al carattere successivo e così via. Ad esempio, la stringa *"alfa"* è minore sia di *"beta"* che di *"alfio"*.

L'unico operatore sulle stringhe è l'*operatore di concatenazione*, rappresentato dal simbolo +. È un operatore binario che restituisce la *giustapposizione* di due stringhe. Ad esempio, il valore di *"Java" + "Script"* è *"JavaScript"*.

4.2 Valutazione delle espressioni

La *valutazione di un'espressione* avviene secondo regole ben precise. Si valutano prima le espressioni più interne e, utilizzando i valori ottenuti, si valutano le altre. Al termine di questo processo si ottiene il valore dell'espressione.

L'*ordine di valutazione* delle espressioni può essere alterato, tenendo in considerazione la *precedenza* degli operatori. Gli operatori con una precedenza più alta sono valutati prima degli altri. A parità di precedenza si valuta l'operatore più a sinistra. Queste due regole rendono univoca la valutazione di un'espressione, evitando possibili ambiguità. Ad esempio, il valore dell'espressione *2 + 3 * 4* può essere calcolato in due modi diversi. Un modo prevede il calcolo della somma tra *2* e *3* (risultato *5*) e poi il prodotto tra *5* e *4* (risultato *20*). Un altro, invece, prevede prima il prodotto tra *3* e *4* (risultato *12*) e poi la somma tra *2* e *12* (risultato *14*). Dando precedenza all'operatore di moltiplicazione rispetto a quello di addizione, si elimina questa ambiguità: il valore dell'espressione *2 + 3 * 4* è univocamente determinato ed è *14*.

La precedenza degli operatori è la seguente, in ordine di precedenza maggiore:

- segno negativo, segno positivo, negazione

- moltiplicazione, divisione, modulo, congiunzione, disgiunzione
- addizione, sottrazione
- operatori di confronto.

Le parentesi tonde permettono di alterare arbitrariamente l'ordine di valutazione determinato dalla priorità degli operatori. Ad esempio, il valore dell'espressione *(2 + 3) * 4* è *20* anziché *14*.

4.3 Casi particolari

L'addizione e la moltiplicazione possono dare come risultato il valore *Infinity*, ovvero un numero troppo grande per essere rappresentato. Nel caso della moltiplicazione questo valore si può ottenere valutando l'espressione *1E308 * 2* oppure dividendo per zero un numero intero. Il valore *–Infinity*, un numero troppo piccolo per essere rappresentato, si ottiene valutando l'espressione *–1E308 * 2* oppure dividendo per zero un numero negativo.

Il modulo si basa su una versione leggermente modificata della definizione euclidea della divisione, in cui il resto è sempre un numero positivo. Se il primo operando è negativo il risultato è negativo. Ad esempio, valutando l'espressione *–10%3* si ottiene il valore *–1*.

4.4 Conversione implicita

Gli operatori numerici assumono che i due operandi siano numerici. Cosa succede se uno di questi operandi appartiene a un altro tipo? La risposta è articolata e si basa sul concetto di *conversione implicita di tipo*. Affrontiamo questo argomento per tutti gli operatori.

Un valore booleano che compare come operando di un operatore numerico è convertito in un numero. In particolare, il valore *true* è convertito nel valore 1, il valore *false* è convertito nel valore zero. La valutazione dell'espressione *1 + true* ha come risultato *2*, la valutazione dell'espressione *1 + false* ha come risultato *1*.

Se una stringa compare come operando di un operatore numerico, di norma il risultato è il valore *NaN* (*Not a Number*). Tuttavia, se la stringa rappresenta un numero, l'operatore converte la stringa nel numero corrispondente ed effettua correttamente l'operazione, restituendo un numero. Ad esempio, la valutazione dell'espressione *2 * '2'* restituisce il valore *4*, anziché *NaN*. L'operatore di addizione introduce un'ulteriore regola: se uno dei due operandi è una stringa che non rappresenta un numero, il risultato è una stringa ottenuta giustapponendo il valore del primo operando con quello del secondo. Ad esempio, la valutazione dell'espressione *200E3 + 'a'* restituisce il valore *200000a*.

Se l'operatore booleano di negazione ha come operando un numero diverso da zero o un carattere il risultato è *false*. Se il valore dell'operando è zero il risultato è *true*.

Diverso è il comportamento degli operatori di congiunzione e di disgiunzione. Se il primo operando dell'operatore di congiunzione (disgiunzione) vale *false* (*true*) il risultato è sempre *false* (*true*). Se il primo operando dell'operatore di congiunzione (disgiunzione) vale *true* (*false*) il risultato è sempre il valore del secondo operando.

4.5 Esercizi

I seguenti problemi devono essere risolti usando letterali e operatori e visualizzando il risultato con il comando di stampa.

1. Calcolare la somma dei primi quattro multipli di 13.
2. Verificare se la somma dei primi sette numeri primi è maggiore della somma delle prime tre potenze di due.
3. Verificare se 135 è dispari, 147 è pari, 12 è dispari, 200 è pari.
4. Calcolare l'area di un triangolo rettangolo i cui cateti sono 23 e 17.
5. Calcolare la circonferenza di un cerchio il cui raggio è 14.
6. Calcolare l'area di un cerchio il cui diametro è 47.
7. Calcolare l'area di un trapezio la cui base maggiore è 48, quella minore è 25 e l'altezza è 13.
8. Verificare se l'area di un quadrato di lato quattro è minore dell'area di un cerchio di raggio tre.
9. Calcolare il numero dei minuti di una giornata, di una settimana, di un mese di 30 giorni, di un anno non bisestile.
10. Verificare se conviene acquistare una camicia che costa 63 € in un negozio che applica uno sconto fisso di 10 € o in un altro che applica uno sconto del 17%.

5 Variabili e assegnamento

Per descrivere un calcolo è necessario tener traccia dei valori intermedi, memorizzandoli per usarli in seguito. Nei linguaggi di programmazione questo ruolo è svolto dalle *variabili*. In JavaScript, come negli altri linguaggi di programmazione imperativi, una *variabile* è un *identificatore* a cui è associato un valore.

```
<Identificatore>  ::= <CarIniziale>
                    |  <CarIniziale> <Caratteri>

<CarIniziale>     ::= <Lettera>
                    |  _
                    |  $

<Caratteri>       ::= <CarNonIniziale>
                    |  <CarNonIniziale> <Caratteri>

<CarNonIniziale>  ::= <Lettera>
                    |  <Cifra>
                    |  _
                    |  $
```

Un identificatore è formato da una sequenza di lettere e cifre e dai caratteri _ e $. Questa sequenza deve iniziare con una lettera o con uno dei caratteri _ e $ ma non può iniziare con una cifra.

Non tutti gli identificatori sono utilizzabili come variabili. In JavaScript, infatti, le *parole riservate* non possono essere usate come variabili. Le parole riservate usate nel libro sono le seguenti.

```
break, case, catch, const, default, else, false, finally, for,
function, if, in, new, null, return, switch, this, throw, true, try,
var, while
```

La *dichiarazione di variabile* è un comando che definisce una variabile associandole un identificatore. Quando la dichiarazione comprende l'assegnamento di un valore si parla più propriamente di *inizializzazione*. Se la variabile non è stata inizializzata il suo valore è il valore speciale *undefined*.

```
<Dichiarazione> ::= <DicVar>

<DicVar>        ::= var <Identificatore>;
                  | var <Identificatore> = <Espressione>;
```

5.1 Dichiarazione di costante

In JavaScript è possibile dichiarare una *costante*, associando a un identificatore un valore che non può essere modificato. Una convenzione molto diffusa prevede che l'identificatore usato in una *dichiarazione di costante* sia formato solo da lettere maiuscole, da numeri e dal carattere _. Ciò permette di distinguere immediatamente una costante da una variabile.

```
<DicCost> ::= const <Identificatore> = <Espressione>;
```

La costante π (pi greco), il cui valore approssimato alle prime dieci cifre decimali è 3,1415926535, può essere dichiarata come segue.

```
 const PI_GRECO = 3,1415926535;
```

5.2 Variabili ed espressioni

Le variabili possono essere usate nelle espressioni. In particolare, una variabile è un'espressione semplice il cui valore è proprio quello della variabile.

```
<Espressione> ::= <Identificatore>
```

La possibilità di dichiarare variabili non inizializzate, il cui valore è *undefined*, richiede di definire il comportamento degli operatori per trattare questo caso particolare. Se almeno uno degli operandi di un operatore numerico è il valore *undefined*, il risultato sarà *NaN*.

Molto più complessa è la casistica relativa agli operatori booleani. Se l'operatore di negazione ha un operando che vale *undefined* il risultato è *true*. Se gli operandi dell'operatore di congiunzione valgono *undefined* il risultato è *undefined*, tranne quando il primo operando vale *false*, nel qual caso il risultato è *false*. Se il primo operando dell'operatore di disgiunzione vale *undefined* il risultato è *true* se il secondo operando vale *true*, *false* se il secondo operando vale *false*. Se, invece, il primo operando vale *false* il risultato è *undefined*. Infine, se entrambi gli operandi dell'operatore di congiunzione o di quello di disgiunzione valgono *undefined*, il risultato è *undefined*.

Gli operatori di confronto seguono una logica più semplice. Se solo un operando vale *undefined*, il risultato è sempre *false*, tranne nel caso dell'operatore di disuguaglianza, per il quale il risultato è *true*. Se i due operandi valgono *undefined*, il risultato è sempre *false*.

5.3 Comando di assegnamento

Il *comando di assegnamento* modifica il valore associato a una variabile, assegnandole quello di un'espressione.

```
<ComandoSemplice> ::= <Assegnamento>

<Assegnamento>    ::= <Identificatore> = <Espressione>;
```

Vediamo alcuni esempi di dichiarazione di variabile, di assegnamento e di uso delle variabili nelle espressioni.

```
var x = 11;
print(x);
var y;
print(y);
y = 7;
print(x + y);
x = y + 10;
print(x);
x = x + 10;
print(x);
```

La prima dichiarazione definisce la variabile *x* e le assegna il valore *11*. Il comando di stampa successivo stampa il valore di *x*. La seconda dichiarazione definisce la variabile *y*, senza assegnarle un valore iniziale. Pertanto, il comando di stampa successivo stampa il valore *undefined*. Il primo comando di assegnamento assegna il valore *7* a *y*. Il successivo comando stampa il valore *18*, ottenuto sommando *11* a *7*. Il secondo comando di assegnamento assegna a *x* la somma del valore di *y* e di *10*, in questo caso *17*, come si può verificare osservando l'effetto del successivo comando di stampa. L'ultimo comando di assegnamento assegna a *x* il suo valore sommato a *10*, in questo caso *27*, risultato verificabile osservando l'effetto dell'ultimo comando di stampa.

5.4 Abbreviazioni del comando di assegnamento

In JavaScript è possibile usare alcune *abbreviazioni* per incrementare o decrementare il valore di una variabile. Il comando $i = i + 1$ è equivalente a $i{+}{+}$ e il comando $i = i - 1$ è equivalente a $i{-}{-}$.

Un'altra caratteristica di JavaScript è la possibilità di abbreviare la forma del comando di assegnamento, quando è riferito a una variabile. Per incrementare il valore della variabile *x* del valore della variabile *i*, anziché scrivere $x = x + i$ è possibile usare la forma più compatta $x \mathrel{+}= i$, il cui significato è: applica l'operatore + alla variabile *x* e all'espressione alla destra del segno = e assegna il risul-

tato alla variabile *x*. Lo stesso ragionamento è applicabile agli altri operatori numerici.

```
<Assegnamento> ::= <Identificatore>++;
                 |  <Identificatore>--;
                 |  <Identificatore> += <Espressione>;
                 |  <Identificatore> -= <Espressione>;
                 |  <Identificatore> *= <Espressione>;
                 |  <Identificatore> /= <Espressione>;
                 |  <Identificatore> %= <Espressione>;
```

5.5 Esercizi

Risolvere i seguenti problemi utilizzando variabili, costanti, letterali e operatori. Il risultato deve essere visualizzato mediante il comando di stampa *print*, disponibile nell'ambiente *EasyJS*.

1. Calcolare il costo di un viaggio in automobile, sapendo che la lunghezza è 750 Km, che il consumo di gasolio è 3,2 litri ogni 100 Km, che un litro di gasolio costa 1,432 €, che due terzi del percorso prevedono un pedaggio pari a 1,2 € ogni 10 Km.
2. Calcolare il costo di una telefonata, sapendo che la durata è pari a 4 minuti e 23 secondi, che il costo alla chiamata è pari a 0,15 €, che i primi 30 secondi sono gratis, che il resto della telefonata costa 0,24 € al minuto.
3. Calcolare il costo di un biglietto aereo acquistato una settimana prima della partenza, sapendo che il costo di base è pari a 200 € (se acquistato il giorno della partenza) e che questo costo diminuisce del 2,3% al giorno (se acquistato prima del giorno della partenza).
4. Calcolare il costo di un prodotto usando la seguente formula

   $$costo = (prezzo + prezzo \cdot 0{,}20) - sconto$$

   e sapendo che il prezzo è 100 € e lo sconto è 30 €.
5. Calcolare la rata mensile di un mutuo annuale usando la seguente formula

   $$rata = \frac{importo}{12} \cdot (1 + tasso)$$

   e sapendo che l'importo annuale è 240 € e il tasso è il 5%.

6 Comandi condizionali

La sintassi di JavaScript che abbiamo definito finora permette di scrivere programmi molto semplici, formati da una sequenza di dichiarazioni di variabili, di assegnamenti e di comandi di stampa. Con questi comandi è possibile affrontare solo problemi la cui soluzione richiede una sequenza di calcoli. Problemi più complessi richiedono l'uso di comandi composti. In questo capitolo presentiamo i *comandi condizionali*.

6.1 Comando condizionale

Il *comando if* è un comando composto che consente di scegliere i comandi da eseguire in base al valore di un'espressione booleana.

```
<ComandoComposto> ::= <If>
                    |  <Blocco>

<If>              ::= if (<Espressione>)
                         <Blocco>
                    |  if (<Espressione>)
                         <Blocco>
                       else <Blocco>
```

Il comando inizia con la parola riservata *if*, seguita da un'espressione tra parentesi tonde, detta *condizione*. Il valore della condizione deve essere un booleano. Se la condizione vale *true* (la condizione è vera), si esegue il primo *blocco di comandi*, detto *blocco*. Se la condizione vale *false* (la condizione è falsa) ci possono essere due casi: se è presente ed è preceduto dalla parola riservata *else*, viene eseguito il secondo blocco, altrimenti non si esegue alcun comando.

Un blocco è costituito da una *sequenza di comandi* racchiusa tra *parentesi graffe*. Un blocco è considerato un comando unico e permette di utilizzare una sequenza di comandi dove normalmente sarebbe possibile usarne solo uno.

Nel seguito mostriamo un programma che usa due comandi condizionali. Nel primo comando la condizione è vera se il valore della variabile x è maggiore di zero: in tal caso il suo valore è decrementato di uno. Non è prevista alcuna azione nel caso in cui il valore di x sia minore o uguale a zero. Il secondo comando è un esempio che mostra come sia possibile assegnare a x un valore maggiore di

zero, indipendentemente dal valore di y. Se il valore di y è minore di zero e quindi la condizione è vera, a x è assegnato tale valore cambiato di segno (e dunque positivo). In caso contrario a x è assegnato il valore di y.

```
if (x > 0) {
   x--;
}
if (y < 0) {
   x = -y;
} else {
   x = y;
}
```

I comandi condizionali possono essere *annidati* per effettuare una scelta tra più di due possibilità. Il seguente esempio mostra due comandi condizionali annidati e mostra anche come sia possibile evidenziare la struttura di un programma mediante l'*indentazione*.

```
if (a < 6) {
   b = 1;
} else {
   if (a < 8) {
      b = 2;
   } else {
      b = 3;
   }
}
```

Un'altra indentazione è la seguente, preferibile alla prima quando sono presenti più di due alternative.

```
if (a < 6) {
   b = 1;
} else if (a < 8) {
   b = 2;
} else {
   b = 3;
}
```

6.2 Comando di scelta multipla

Il *comando di scelta multipla* è un comando composto che rappresenta un'alternativa a una sequenza di comandi condizionali annidati. Si usa quando si devono eseguire comandi diversi, associati a determinati valori di un'espressione.

```
<ComandoComposto> ::= <Switch>

<Switch>          ::= switch (<Espressione>)
                         {<Alternative>}
                    |  switch (<Espressione>)
                         {<Alternative> default: <Comandi>}

<Alternativa>     ::= case <Letterale>: <Comandi>

<Alternative>     ::= <Alternativa>
                    |  <Alternativa> <Alternative>

<ComandoSemplice> ::= break;
```

L'espressione (chiamata anche *selettore*) è valutata e il suo valore è confrontato con quello dei letterali di ciascuna alternativa, partendo dalla prima. Se il confronto ha esito positivo per un'alternativa, si eseguono i comandi ad essa associati e quelli di tutte le alternative successive. Se il confronto ha esito negativo per tutte le alternative, non si esegue alcun comando.

```
switch (x) {
   case 5 : y += 5;
   case 6 : z += 6;
}
```

In questo esempio, se il valore dell'espressione *x* è uguale a *6*, la variabile *z* è incrementata di *6*. Se il valore è *5*, la variabile *y* è incrementata di *5* e la variabile *z* è incrementata di *6*. Se il valore è diverso da *5* o da *6*, non si esegue alcun comando.

In molti casi è preferibile che dopo l'esecuzione del codice associato a un'alternativa l'esecuzione passi al comando successivo a quello di selezione multipla. Per ottenere questo comportamento l'ultimo comando di quelli associati a un'alternativa deve essere il comando *break*.

```
switch (x) {
   case 5 : y += 5; break;
   case 6 : z += 6; break;
}
```

Con questa modifica, anche quando il valore del selettore è uguale a *5*, la variabile *z* non è incrementata.

È possibile inserire, rigorosamente per ultima, un'alternativa speciale i cui comandi sono eseguiti quando il confronto ha avuto esito negativo per tutte le alternative.

```
switch (x) {
   case  5 : y += 5; break;
   case  6 : z += 6; break;
   default : z++;
}
```

Con questa modifica, in tutti i casi in cui il valore del selettore è diverso da *5* o da 6, la variabile *z* è incrementata di uno.

7 Funzioni

Una *dichiarazione di funzione* è un comando che definisce un identificatore a cui è associata una *funzione*. La definizione della funzione comprende un'*intestazione* e un blocco di comandi.

```
<Dichiarazione> ::= <DicFun>

<DicFun>        ::= function <Identificatore>()
                    <Blocco>
                  | function <Identificatore>(<Parametri>)
                    <Blocco>

<Parametri>     ::= <Identificatore>
                  | <Identificatore>, <Parametri>

<Blocco>        ::= {<Comandi>}
```

L'intestazione della funzione definisce la lista dei suoi *parametri* (chiamati anche *parametri formali*) racchiusi tra due parentesi tonde. La lista dei parametri formali può essere vuota.

Come esempio, definiamo la funzione *stampaSomma* che stampa la somma dei suoi parametri.

```
function stampaSomma (n, m) {
   print(n + m);
}
```

Dopo aver dichiarato una funzione, è possibile usarla in un punto qualunque del programma. Il punto in cui si usa una funzione è detto punto di *chiamata* o di *invocazione*. Quando si invoca la funzione, si assegna a ciascun parametro formale il valore dell'espressione utilizzata nel punto di chiamata e poi si esegue il blocco di comandi.

Le espressioni utilizzate nel punto di invocazione (chiamate anche *parametri attuali*) sono valutate prima dell'esecuzione del blocco di comandi associato alla funzione e il loro valore è associato ai parametri formali. Questo procedimento è detto *passaggio dei parametri*. L'associazione tra i parametri formali e i valori dei parametri attuali avviene con il *sistema posizionale*, ovvero ad ogni parame-

tro formale è associato il valore del parametro attuale che occupa la medesima posizione nella rispettiva lista. Il primo parametro attuale è dunque legato al primo parametro formale, il secondo parametro attuale è legato al secondo parametro formale e così via.

```
<ComandoSemplice> ::= <Invocazione>

<Invocazione>     ::= <Identificatore> ();
                    |  <Identificatore> (<Espressioni>);

<Espressioni>     ::= <Espressione>
                    |  <Espressione>, <Espressioni>
```

Gli esempi che seguono mostrano due invocazioni di funzione con un numero di parametri attuali pari a quello dei parametri formali. L'effetto delle invocazioni è, rispettivamente, stampare prima il valore *5* e poi il valore *15*.

```
stampaSomma(10, -5);
stampaSomma(10, 5);
```

Normalmente il numero dei parametri attuali è uguale a quello dei parametri formali definiti nella dichiarazione. Se il numero dei parametri attuali è diverso da quello dei parametri formali, non si ottiene un errore, ma:

- se il numero dei parametri attuali è minore di quello dei parametri formali, il valore *undefined* è assegnato ai parametri formali che non hanno un corrispondente parametro attuale, ovvero agli ultimi della lista;
- se il numero dei parametri attuali è maggiore di quello dei parametri formali, i parametri attuali in eccesso sono ignorati.

Una funzione può *restituire* un valore al termine della propria esecuzione, eseguendo il comando *return* seguito da un'espressione.

```
<ComandoSemplice> ::= return <Espressione>;
```

Ad esempio, definiamo una funzione che calcola e restituisce la somma dei suoi parametri.

```
function calcolaSomma (x, y) {
   return x + y;
}
```

Una funzione che restituisce un valore può essere invocata in un qualunque punto di un programma in cui è lecito usare un'espressione, ad esempio in un comando di assegnamento o in un comando di stampa.

```
var x = 1;
var y = 2;
x = calcolaSomma(x, y);
print(calcolaSomma(x, y));
```

La sintassi delle espressioni è estesa con l'invocazione di funzione.

```
<Espressione>     ::= <Identificatore>()
                    |  <Identificatore>(<Espressioni>)
```

Le funzioni che restituiscono un valore booleano sono chiamate *predicati*. Come esempio, definiamo il predicato *compreso* che verifica se *x* appartiene all'*intervallo* [*a*, *b*)[5]. Il predicato può essere usato per verificare se il numero *1* appartiene all'intervallo [*0, 7*).

```
function compreso (x, a, b) {
   return (x >= a) && (x < b);
}
print(compreso(1, 0, 7));
```

7.1 Funzioni anonime

In JavaScript è anche possibile definire una funzione senza indicare il suo nome. La funzione così definita, chiamata *anonima*, può essere assegnata a un identificatore oppure può essere utilizzata immediatamente. Ad esempio, la funzione che verifica se un numero appartiene a un intervallo può essere definita in maniera anonima e poi assegnata alla variabile *compreso*. L'invocazione della funzione segue le regole già indicate.

```
var compreso = function (x, a, b) {
   return (x >= a) && (x < b);
}
print(compreso(1, 0, 7));
```

Alternativamente è possibile definire una funzione e invocarla con i suoi parametri attuali.

```
print((function (x, a, b) {
          return (x >= a) && (x < b);
       }(1, 0, 7));
```

La leggibilità delle funzioni definita con questa tecnica è molto bassa e la loro corretta indentazione è assolutamente necessaria.

5 L'intervallo [*a, b*) è formato dai numeri maggiori o uguali di *a* e minori di *b*. Per convenzione si assume che *a* sia minore o uguale di *b*. Quando *a* è uguale a *b* l'intervallo è vuoto.

7.2 Anno bisestile

L'*anno bisestile* è un anno solare in cui il mese di febbraio ha 29 giorni anziché 28. Questa variazione evita lo slittamento delle stagioni che ogni quattro anni accumulerebbero un giorno in più di ritardo. Nel *calendario giuliano* è bisestile un anno ogni quattro (quelli la cui numerazione è divisibile per quattro). Nel *calendario gregoriano* si mantiene questa variazione ma si eliminano tre anni bisestili ogni 400 anni [Wikipedia, alla voce *anno bisestile*].

Il predicato *bisestile* verifica se un anno è bisestile.

```
function bisestile (anno) {
   if (anno % 400 == 0) {
      return true;
   } else if (anno % 100 == 0) {
      return false;
   } else if (anno % 4 == 0) {
      return true;
   } else {
      return false;
   }
}
```

Una versione più semplice del predicato *bisestile* non fa uso del comando condizionale.

```
function bisestile (anno) {
   return (anno % 400 == 0) ||
          ((anno % 4 == 0) && (anno % 100 != 0));
}
```

7.3 Visibilità

Quando si dichiara una variabile o una funzione è necessario tenere in debita considerazione la loro *visibilità*. In pratica è necessario sapere quali variabili sono definite nel programma e quali sono, invece, definite nel corpo della funzione. Il modello adottato da JavaScript è complesso, essendo basato sull'esecuzione del programma ma anche sulla sua struttura sintattica. Ai fini di questo libro, tuttavia, si ritiene sufficiente presentare una versione semplificata del modello, versione che permette di spiegare il comportamento degli esempi riportati.

Un *ambiente* è formato da un insieme di variabili e di funzioni. In un ambiente non possono esserci due variabili che hanno lo stesso identificatore, altrimenti non sarebbe possibile sapere quale dei due dovrà essere utilizzato al momento

della valutazione della variabile. Allo stesso modo non ci possono essere due funzioni con lo stesso identificatore.

Un ambiente è modificato dalle dichiarazioni, che introducono nuove variabili e nuove funzioni, e dai comandi, che modificano il valore delle variabili presenti nell'ambiente. Ogni punto di un programma ha un ambiente, formato dalle variabili definite in quel punto. Al contrario delle variabili, tutte le funzioni dichiarate in un programma fanno parte del suo ambiente.

Ogni variabile è visibile in una porzione ben definita del programma in cui essa è dichiarata. In particolare, la variabile è visibile dal punto in cui è dichiarata la prima volta (o assegnata la prima volta) fino al termine del programma. Una variabile visibile solo in alcune porzioni del programma è detta *variabile locale*. Una variabile la cui visibilità è l'intero programma è detta *variabile globale*. Una variabile globale può essere nascosta da una variabile locale che ha lo stesso identificatore. Un variabile nascosta è definita, ma non è accessibile.

In JavaScript le uniche variabili locali sono quelle dichiarate nel corpo delle funzioni. I parametri formali di una funzione sono trattati come variabili dichiarate al suo interno e quindi locali. Tutte le altre variabili sono globali.

```
var risultato = 0;
function calcolaSomma (x, y) {
   var somma = 0;
   somma = x + y;
   return somma;
}
risultato = calcolaSomma(2, 4);
```

In questo esempio compaiono le variabili *x*, y, *somma* e *risultato*. Le variabili *x*, *y* e *somma* sono locali al corpo della funzione *calcolaSomma*. La variabile *risultato* è globale. In base alla definizione di visibilità, le variabili locali sono utilizzabili solo nel corpo di *calcolaSomma*, mentre la variabile globale *risultato* è utilizzabile sia nel corpo di *calcolaSomma*, sia nel resto del programma.

Il seguente programma sfrutta la visibilità delle variabili per ottenere lo stesso risultato.

```
var risultato = 0;
function calcolaSomma (x, y) {
   var somma = 0;
   somma = x + y;
   risultato = somma;
}
calcolaSomma(2, 4);
```

Questo esempio mostra come sia possibile modificare il valore di una variabile globale all'interno del corpo di una funzione. L'utilità di questa tecnica sarà apprezzata nella seconda parte del libro, quando presenteremo problemi che richiedono soluzioni complesse.

7.4 Funzioni di conversione

In JavaScript è possibile convertire un valore di un tipo in un valore di un altro tipo. Il caso più semplice è la conversione di una stringa formata esclusivamente da caratteri numerici in un valore numerico. Un altro caso semplice è la conversione di un valore numerico nella stringa corrispondente.

- *parseInt* (x): converte la stringa x in un valore intero,
- *parseFloat* (x): converte la stringa x in un valore numerico,
- *String* (x): converte il valore numerico x in una stringa.

7.5 Funzioni predefinite

JavaScript mette a disposizione numerose funzioni matematiche di uso corrente, chiamate *funzioni predefinite*. Un elenco non esaustivo di queste funzioni è il seguente:

- *Math.abs* (*x*): restituisce il *valore assoluto* di *x*,
- *Math.ceil* (*x*): restituisce il primo intero più grande di *x*,
- *Math.floor* (*x*): restituisce il primo intero più piccolo di *x*,
- *Math.log* (*x*): restituisce il *logaritmo* naturale in base *e* di *x*,
- *Math.pow* (*x*, *y*): restituisce *x* elevato alla potenza di *y*,
- *Math.random* (): restituisce un *numero casuale* compreso tra zero e uno,
- *Math.round* (x): restituisce l'intero più vicino a *x*,
- *Math.sqrt* (*x*): restituisce la *radice quadrata* di *x*.

7.6 Esercizi

1. L'*equazione di secondo grado* $ax^2 + bx + c = 0$ ha due radici [Wikipedia, alla voce *Equazione di secondo grado*]:

   $$x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   Definire in JavaScript una funzione che stampa le radici di un'equazione i cui coefficienti sono *a*, *b* e *c* (con *a* diverso da zero).

   La funzione ha tre parametri: *a*, *b*, *c*.

Invocare la funzione con i seguenti valori: *1*, *-5*, *6*; *1*, *8*, *16*; *1*, *2*, *3*.

2. I giorni di un anno possono essere numerati consecutivamente, assumendo che al primo gennaio sia assegnato il valore *1* e al trentuno dicembre il valore *365* negli anni non bisestili o *366* negli anni bisestili.

   Definire in JavaScript una funzione che restituisce il numero assegnato a un giorno dell'anno, assumendo che i mesi siano numerati da *1* a *12*.

   La funzione ha tre parametri: *anno*, *mese*, *giorno*.

   Invocare la funzione con i seguenti valori: *1957*, *4*, *25*; *2004*, *11*, *7*; *2000*, *12*, *31*; *2012*, *2*, *29*.

3. Definire in JavaScript una funzione che calcola e restituisce una stringa che rappresenta il valore in lettere di un intero che appartiene all'intervallo [*0, 100*).

   La funzione ha un parametro: *n*.

   Invocare la funzione con i seguenti valori: *0*, *1*, *12*, *21*, *32*, *43*, *70*, *88*, *90*.

4. Definire in JavaScript una funzione che calcola il tipo di un triangolo (*equilatero*, *isoscele*, *rettangolo*, *scaleno*) in base alla lunghezza dei suoi lati.

   La funzione restituisce una stringa che rappresenta il tipo del triangolo e ha tre parametri: *a*, *b*, *c*.

   Invocare la funzione con i seguenti valori: *3*, *3*, *3*; *3*, *4*, *4*; *3*, *4*, *5*; *3*, *4*, *6*.

5. Definire in JavaScript un predicato che verifica se l'intersezione dell'intervallo [*a*, *b*) con l'intervallo [*c*, *d*) è vuota.

   Il predicato ha quattro parametri: *a*, *b*, *c*, *d*.

   Invocare il predicato con i seguenti valori: *2*, *4*, *5*, *7*; *2*, *4*, *4*, *7*; *2*, *4*, *3*, *7*; *5*, *7*, *2*, *4*; *4*, *7*, *2*, *4*; *3*, *7*, *2*, *4*.

6. Definire in JavaScript una funzione che calcola e restituisce la somma delle cifre di un intero che appartiene all'intervallo [*0, 100*).

   La funzione ha un parametro: *n*.

   Invocare la funzione con i seguenti valori: *0*, *1*, *23*, *99*.

8 Comandi iterativi

Molti problemi richiedono un calcolo che deve essere ripetuto più volte per ottenere il risultato finale. In JavaScript la ripetizione di un calcolo si ottiene utilizzando un *comando iterativo*.

8.1 Comando iterativo determinato

Il *comando iterativo determinato* esegue un blocco di comandi un numero determinato di volte. Il comando è formato da un'intestazione che contiene un comando di inizializzazione di una variabile (chiamata anche *indice di iterazione*) che conta il numero delle iterazioni, un'espressione (chiamata *guardia*) che controlla quante volte il blocco di comandi è eseguito, un comando che aggiorna il valore dell'indice di iterazione.

La forma sintattica del comando iterativo determinato è molto generale ma in questo libro ne adotteremo una che ci permette di risolvere i problemi che richiedono l'uso di un'iterazione determinata.

```
<ComandoComposto> ::= <For>
<For> ::= for (<Comando>; <Espressione>; <Comando>)
             <Blocco>
```

L'esecuzione di un comando iterativo determinato segue il seguente schema:

- l'indice di iterazione è inizializzato;
- la guardia è valutata prima di eseguire il blocco di comandi;
- se il valore della guardia è *true* (cioè se la guardia è verificata), il blocco è eseguito, l'indice di iterazione è aggiornato e il ciclo è ripetuto;
- se il valore della guardia è *false* (cioè se la guardia non è verificata), il blocco non è eseguito e l'esecuzione dell'intero comando termina.

La seguente funzione calcola la somma dei numeri da uno a *n*.

```
function somma (n) {
   var s = 0;
   for (var i = 1; i <= n; i++) {
      s += i;
   }
```

```
   return s;
}
```

La funzione può essere definita usando un comando iterativo determinato che somma i numeri da *n* a uno.

```
function somma (n) {
   var s = 0;
   for (var i = n; i > 0; i--) {
      s += i;
   }
   return s;
}
```

8.2 Comando iterativo indeterminato

Il *comando iterativo indeterminato* è utilizzato quando un blocco di comandi deve essere eseguito più volte, ma non è possibile sapere a priori quante. Il comando è formato da un'espressione, chiamata guardia, e un blocco di comandi.

```
<ComandoComposto> ::= <While>
<While>           ::= while (<Espressione>)
                          <Blocco>
```

L'esecuzione di un comando iterativo indeterminato segue il seguente schema:

- la guardia è valutata ad ogni iterazione, prima di eseguire il blocco di comandi;
- se il valore della guardia è *true* (cioè se la guardia è verificata), il blocco è eseguito e poi si ripete il ciclo;
- se il valore della guardia è *false* (cioè se la guardia non è verificata), il blocco non è eseguito e l'esecuzione dell'intero comando termina.

In JavaScript un comando iterativo determinato può sempre essere espresso mediante un comando iterativo indeterminato (ma non viceversa).

```
function sommaA (n) {
   var s = 0;
   var i = 1;
   while (i <= n) {
      s += i;
      i++;
   }
   return s;
}
```

Da un punto di vista pratico è bene utilizzare il comando iterativo determinato quando il numero di volte che sarà eseguito il blocco di comandi è noto a priori, usando il comando iterativo indeterminato in tutti gli altri casi.

8.3 Primalità

Un *numero primo* è un numero naturale maggiore di uno, divisibile solamente per uno e per sé stesso [Wikipedia, alla voce *Numero primo*]. I primi dieci numeri primi sono *2*, *3*, *5*, *7*, *11*, *13*, *17*, *19*, *23*, *29*.

Dato un numero naturale maggiore di uno è possibile verificare se è primo, cioè se soddisfa la condizione di primalità, definendo in JavaScript il predicato *ePrimo* basato su un comando di iterazione determinata.

```
function ePrimo (n) {
   var c = 0;
   for (var i = 2; i < n; i++) {
      if (n % i == 0) {
         c++;
      }
   }
   return (c == 0);
}
```

Una soluzione alternativa si basa su una variabile booleana.

```
function ePrimo (n) {
   var b = true;
   for (var i = 2; i < n; i++) {
      b = b && (n % i != 0);
   }
   return b;
}
```

L'espressione usata per aggiornare il valore della variabile booleana può essere utilizzata come guardia di un comando iterativo indeterminato.

```
function ePrimo (n) {
   var i = 2;
   while ((i < n) && (n % i != 0)) {
      i++;
   }
   return (i == n);
}
```

La soluzione basata sul comando iterativo indeterminato può essere migliorata tenendo conto del fatto che se *n* è primo allora non esiste un divisore *d* di *n* minore della sua radice quadrata:

$$d < \sqrt{n}$$
$$d^2 < n$$

```
function ePrimo (n) {
   var i = 2;
   while ((i * i < n) && (n % i != 0)) {
      i++;
   }
   return (i * i > n);
}
```

8.4 Radice quadrata intera

La radice quadrata di un numero razionale non negativo *z* è un numero *x*, anch'esso non negativo, che soddisfa l'equazione

$$x^2 = z$$

[Wikipedia, alla voce *Radice quadrata*]. Ad esempio, la radice quadrata di *2* è *1,4142135623*.

La *radice quadrata intera* di un numero razionale non negativo *z* è un intero positivo *x* che soddisfa l'equazione

$$x^2 \leq z < (x+1)^2$$

Ad esempio, la radice quadrata intera di *5* è *2*. In JavaScript si può definire una funzione che calcola la radice quadrata intera di un numero razionale non negativo.

```
function radiceQuadrata (z) {
   var x = 0;
   while (x * x <= z) {
      x++;
   }
   return x - 1;
}
```

8.5 Esercizi

1. Un anno è perfetto per una persona se è un multiplo della sua età in quell'anno, assumendo come età massima cento anni. Ad esempio gli anni

perfetti per chi è nato nel *1984* sono nove: *1985*, *1986*, *1988*, *1992*, *2000*, *2015*, *2016*, *2046*, *2048*.

Definire in JavaScript una funzione che stampa gli anni perfetti relativi a un anno di nascita.

La funzione ha un parametro: *annoDiNascita*.

Invocare la funzione con i seguenti valori: *1984*, *1990*, *1992*, *2000*, *2009*.

2. In matematica, con reciproco di un numero x si indica il numero che moltiplicato per x dà come risultato 1.

   Definire in JavaScript una funzione che calcola e restituisce la somma dei reciproci dei primi n numeri interi maggiori di zero.

   La funzione ha un parametro: n.

   Invocare la funzione con i seguenti valori: *1*, *2*, *4*, *7*.

3. Definire in JavaScript una funzione che stampa, se esistono, le radici intere di un'equazione di secondo grado di coefficienti a, b, c comprese nell'intervallo $[l, u)$.

   La funzione ha i seguenti parametri: a, b, c, l, u.

   Invocare la funzione con i seguenti valori:

   1, *−2*, *−8*, *1*, *5*

   1, *−2*, *−8*, *−5*, *5*

   1, *−2*, *−8*, *5*, *10*.

9 Array

In JavaScript, oltre ai numeri, i booleani e i caratteri, esistono altri tipi di dato che, per come sono strutturati, sono chiamati *tipi composti*, per differenziarli da quelli primitivi. In questo capitolo presentiamo gli array, un tipo di dato indispensabile per la soluzione di numerosi problemi.

9.1 Elementi e indici di un array

Un *array* è un tipo composto, formato da una sequenza numerata di valori[6]. Ogni valore è detto *elemento* dell'array e il numero a esso associato è detto *indice*. Il primo elemento di un array ha sempre indice zero.

Per dichiarare un array è necessario indicarne il nome, un identificatore, e un valore determinato dall'espressione a destra del segno di assegnamento. La categoria sintattica delle espressioni è così estesa.

```
<Espressione>  ::= []
                |  [<Espressioni>]
```

Vediamo alcuni esempi di dichiarazione di array.

```
var a = [];
var b = [12];
var c = ['f', 'u', 'n', 'e']
```

Nel primo caso si ottiene un array vuoto. Nel secondo caso un array con un elemento, *12*. Nel terzo caso un array con quattro elementi, i caratteri *f*, *u*, *n*, ed *e*.

Ccreato un array, si può accedere ai suoi elementi per utilizzarne i valori. L'espressione tra parentesi quadre deve avere un valore maggiore o uguale a zero.

```
<Espressione>  ::= <Identificatore>[<Espressione>]
```

Vediamo un esempio in cui si dichiara un array e si utilizzano i suoi elementi.

```
var a = [2, 3, 5, 7, 11];
print(a[0] + a[1] + a[2] + a[3] + a[4]);
```

6 In JavaScript gli array possono essere *omogenei* (tutti i valori sono dello stesso tipo) o *disomogenei* (i valori sono di più tipi). Per semplicità di presentazione assumiamo che gli array siano omogenei.

Il valore degli elementi di un array può essere modificato mediante il comando di assegnamento. Anche in questo caso l'espressione tra parentesi quadre deve avere un valore maggiore o uguale a zero.

```
<Assegnamento> ::= <Identificatore>[<Espressione>] =
                                    <Espressione>;
```

Vediamo un esempio in cui si dichiara l'array *a* e si modifica il secondo elemento, il cui indice è uno.

```
var a = [2, 3, 5, 7, 11];
a[1] = 0;
print(a[0] + a[1] + a[2] + a[3] + a[4]);
```

9.2 Lunghezza di un array

Come vedremo nella seconda parte, in JavaScript è possibile definire *oggetti*, tipi di dato composti caratterizzati da un insieme di *proprietà* e di *metodi*.

Una proprietà è un valore associato a un oggetto. Per accedere al valore di una proprietà si utilizza la cosiddetta *notazione a punti* (*dot notation*): l'oggetto è seguito da un punto e dal nome della proprietà.

Un metodo è una funzione associata a un oggetto. Anche per invocare il metodo si utilizza la notazione a punti. Maggiori dettagli su proprietà e metodi saranno forniti nella seconda parte.

Gli array sono *oggetti predefiniti* che hanno la proprietà *length*, il cui valore è pari al numero degli elementi dell'array. Questa proprietà può essere utilizzata per scandire un array, come mostrato nella funzione *stampaElementi* che stampa tutti gli elementi di un array. La funzione è invocata prima con un array di tre elementi e poi con uno di cinque elementi.

```
function stampaElementi (a) {
   for (var i = 0; i < a.length; i++) {
      print(a[i]);
   }
}
var a1 = [2, 3, 5];
stampaElementi(a1);
var a2 = [0, -9, 17, 4, 100];
stampaElementi(a2);
```

Anche quando non si intende effettuare una scansione completa di un array è spesso utile usare esplicitamente la proprietà *length*. Ad esempio, la funzione *stampaElementiIndicePari*, è così definita.

```
function stampaElementiIndicePari (a) {
   for (var i = 0; i < a.length; i += 2) {
      print(a[i]);
   }
}
```

9.3 Array dinamici

In JavaScript, a differenza di altri linguaggi di programmazione, è possibile aggiungere dinamicamente nuovi elementi a un array. Per fare ciò, si esegue un assegnamento all'elemento che si vuole aggiungere, come se già esistesse. Nell'esempio che segue creiamo un array di quattro elementi a cui, dopo, ne aggiungiamo uno.

```
var a = [2, 3, 5, 7, 11];
a[5] = 13;
print(a[0] + a[1] + a[2] + a[3] + a[4] + a[5]);
```

Quando si aggiunge un elemento il cui indice non è immediatamente successivo a quello dell'ultimo elemento definito dell'array, automaticamente tutti gli elementi intermedi sono aggiunti e inizializzati con il valore *undefined*.

```
var a = [2, 3, 5, 7, 11];
a[10] = 13;
```

Nell'esempio precedente si aggiunge l'elemento di indice *10* a un array il cui ultimo elemento definito ha indice *4*. L'aggiunta di questo elemento provoca la creazione degli elementi di indice *5*, *6*, *7*, *8* e *9*, tutti inizializzati con il valore *undefined*.

Per aggiungere a un array un elemento con indice successivo a quello del suo ultimo elemento si usa il metodo *push*.

```
var a = [2, 3, 5, 7, 11];
a.push(13);
```

9.4 Array associativi

In JavaScript è possibile utilizzare come indici di un array anche valori di tipo stringa. In questo caso si parla di *array associativi*.

Un array associativo può essere creato in due modi: esplicitamente, mediante l'indicazione dei suoi indici e dei valori associati; implicitamente, mediante assegnamenti che creano dinamicamente l'array. La creazione esplicita di un array associativo formato da tre elementi è indicata nel seguito.

```
var a = {alfa : 10, beta : 20, gamma : 30};
print(a["alfa"]);
print(a["beta"]);
print(a["gamma"]);
```

La creazione implicita dello stesso array è la seguente.

```
var a = {};
a["alfa"] = 10;
a["beta"] = 20;
a["gamma"] = 30;
```

Per gli array associativi la proprietà *length* non è definita perché gli indici di questi array non hanno un valore numerico. Per scandire tutti gli elementi di un array associativo si usa una forma particolare di iterazione determinata in cui l'indice di iterazione assume tutti i valori utilizzati per definire l'array[7].

```
<For> ::= for (var <Identificatore> in <Espressione>)
          <Blocco>
```

Nell'esempio che segue *i* assume, in sequenza, i valori *alfa*, *beta* e *gamma*.

```
var a = {alfa : 10, beta : 20, gamma : 30};
for (var i in a) {
   print (i);
}
```

Per verificare se una stringa è un indice di un array associativo si usa un'espressione booleana con la seguente sintassi.

```
<Espressione> ::= <Espressione> in <Espressione>
```

Per verificare se la stringa *alfa* è un indice di *a* si usa la seguente espressione.

```
print("alfa" in a);
```

9.5 Stringhe di caratteri

Un caso particolare di array è costituito dalle stringhe di caratteri che in JavaScript sono oggetti predefiniti con proprietà e metodi. Tutto quanto detto sugli array si applica alle stringhe. In particolare, anche per le stringhe è definita la proprietà *length*, il cui valore è pari al numero dei caratteri di una stringa.

```
var alfa = "ciao";
print(alfa.length);
```

7 Il valore dell'indice di iterazione non è di tipo numerico ma di tipo stringa.

Per selezionare un carattere di una stringa si utilizza la stessa notazione prevista per gli array: la posizione del carattere è indicata tra parentesi quadre.

```
var alfa = "ciao";
print(alfa[0]);
print(alfa[1]);
print(alfa[2]);
print(alfa[3]);
```

La variabile *alfa* è inizializzata con la stringa *ciao*. I quattro caratteri di *alfa* sono stampati in sequenza. A differenza degli array, tuttavia, non è possibile modificare i caratteri di una stringa.

Per scandire i caratteri di una stringa si usa il comando di iterazione determinata.

```
function stampaCaratteri (a) {
   for (var i = 0; i < a.length; i++) {
      print(a[i]);
   }
}
function stampaElementiIndicePari (a) {
   for (var i = 0; i < a.length; i += 2) {
      print(a[i]);
   }
}
```

Le stringhe hanno molti metodi, usati per svolgere operazioni di manipolazione su testi. Quelli più comunemente usati sono i seguenti:

- *substr*: ha due parametri *p* e *n*. Restituisce, a partire dalla posizione indicata dal valore di *p*, *n* caratteri della stringa su cui è invocato.

```
var a = 'alfabeto'
var b = a.substr(2, 4);      // b vale 'fabe'
```

- *indexOf*: ha un parametro *s*. Restituisce l'indice della prima occorrenza della stringa *s*, incontrata a partire da sinistra, nella stringa su cui è invocato. Il valore restituito è *-1* se non vi sono occorrenze di *s*.

```
var a = 'alfabeto'
var b = a.indexOf('beto');    // b vale  4
var c = a.indexOf('Alfa');    // c vale -1
```

- *toLowerCase:* non ha parametri. Restituisce una stringa uguale a quella su cui è invocato, ma con tutti i caratteri minuscoli.

```
var a = 'ALFAbeto'
var b = a.toLowerCase();      // b vale 'alfabeto'
```

Questi metodi non modificano la stringa su cui sono invocati.

9.6 Ricerca lineare

Molti problemi di programmazione che prevedono l'uso di array possono essere risolti utilizzando uno schema chiamato *ricerca lineare*, che può essere *certa* o *incerta*. Lo schema di *ricerca lineare certa* si utilizza quando si ha la certezza a priori che il valore cercato sia presente nell'array. Negli altri casi si utilizza lo schema di *ricerca lineare incerta*.

Lo schema di ricerca lineare certa si basa su un'iterazione indeterminata in cui la guardia è formata da un'unica espressione logica che è vera se la condizione di ricerca non è soddisfatta. Un esempio, molto semplice, di problema che può essere risolto con questo schema è il seguente: dato un array *a*, definire una funzione che calcola e restituisce il valore dell'indice di un elemento di *a* che vale *k*, sapendo a priori che un tale elemento appartiene all'array.

```
function indice (a, k) {
   var i = 0;
   while (a[i] != k) {
      i++;
   }
   return i;
}
```

Anche lo schema di ricerca lineare incerta si basa su un'iterazione indeterminata e da una guardia formata da due espressioni logiche in congiunzione tra loro: la prima è vera se il valore dell'indice è valido per l'array (cioè se appartiene all'intervallo compreso tra zero e la lunghezza dell'array meno uno), la seconda è vera se la condizione di ricerca non è soddisfatta.

La funzione *indice* può essere modificata secondo questo schema, facendo però in modo che restituisca il valore *-1* se il valore *k* non appartiene all'array *a*.

```
function indice (a, k) {
   var i = 0;
   while ((i < a.length) && (a[i] != k)) {
      i++;
   }
   if (i < a.length) {
      return i;
   } else {
      return -1;
```

```
    }
}
```

Usando lo schema di ricerca incerta è anche possibile definire un predicato che verifica se il valore *k* appartiene all'array *a*.

```
function appartiene (a, k) {
   var i = 0;
   while ((i < a.length) && (a[i] != k)) {
      i++;
   }
   return (i < a.length);
}
```

Alcuni problemi si incontrano spesso in varie formulazioni. La loro soluzione è considerata un classico della programmazione. Di seguito presentiamo alcuni dei più conosciuti.

9.7 Minimo e massimo di un array

Dato un array *a* (non vuoto) di numeri, il *minimo* di *a* è quell'elemento di *a* minore o uguale a tutti gli altri elementi di *a*. Analogamente, il *massimo* di *a* è quell'elemento di *a* maggiore o uguale a tutti gli altri elementi di *a*. La funzione *minimo* calcola e restituisce il minimo di *a*.

```
function minimo (a) {
   var min = a[0];
   for (var i = 1; i < a.length; i++) {
      if (a[i] < min) {
         min = a[i];
      }
   }
   return min;
}
```

La funzione *massimo* calcola e restituisce il massimo di *a*.

```
function massimo (a) {
   var max = a[0];
   for (var i = 1; i < a.length; i++) {
      if (a[i] > max) {
         max = a[i];
      }
   }
   return max;
}
```

9.8 Array ordinato

Un array (non vuoto) è ordinato se ogni coppia adiacente di elementi soddisfa una relazione di ordinamento. In un array ordinato in senso crescente (decrescente) l'ultimo elemento è il massimo (minimo) dell'array e il primo elemento è il minimo (massimo) dell'array. Per ogni coppia di elementi adiacenti, il primo elemento è minore (maggiore) del secondo. Gli array possono essere anche ordinati in senso non decrescente (non crescente). In questo caso la relazione di ordinamento tra gli elementi adiacenti è quella di minore o uguale (maggiore o uguale).

Il predicato *ordinatoCrescente* verifica se *a* è ordinato in senso crescente.

```
function ordinatoCrescente (a) {
   var c = 0;
   for (var i = 1; i < a.length; i++) {
      if (a[i - 1] < a[i]) {
         c++;
      }
   }
   return (c == (a.length - 1));
}
```

La funzione scandisce tutte le coppie adiacenti, incrementando di uno la variabile *c* per ogni coppia che rispetta l'ordinamento. Se tutte le coppie rispettano l'ordinamento, la funzione restituisce il valore *true*. Una versione basata sullo schema di ricerca lineare incerta è mostrata nel seguito.

```
function ordinatoCrescente (a) {
   var i = 1;
   while ((i < a.length) && (a[i - 1] < a[i])) {
      i++;
   }
   return (i == a.length);
}
```

Il predicato *ordinatoDecrescente* si ottiene a partire da *ordinatoCrescente* invertendo la relazione di ordinamento.

9.9 Filtro

Un *filtro* è uno strumento per la selezione (filtraggio) delle parti di un insieme [Wikipedia, alla voce Filtro]. Un esempio di filtro, chiamato *filtro passa banda*, seleziona tutti i valori che appartengono all'intervallo [*lo*, *hi*). I valori *lo* e *hi* sono chiamati *livello basso* (*low level*) e *livello alto* (*high level*).

La funzione *filtro* ha tre parametri (*a*, *lo*, hi) e restituisce un nuovo array formato da tutti gli elementi di *a* i cui valori sono compresi tra *lo* e *hi*.

```
function filtro (a, lo, hi) {
   var b = [];
   for (var i = 0; i < a.length; i++) {
      if ((a[i] >= lo) && (a[i] < hi)) {
         b.push(a[i]);
      }
   }
   return b;
}
```

9.10 Inversione di una stringa

Data una stringa, la sua stringa inversa si ottiene leggendola al contrario. Ad esempio, la stringa inversa di *alfa* è *afla*.

La funzione *inverti* ha un parametro (*s*) e restituisce la stringa inversa di *s*.

```
function inverti (s) {
   var t = "";
   for (var i = 0; i < s.length; i++) {
      t = s[i] + t;
   }
   return t;
}
```

Un'altra soluzione utilizza un'iterazione determinata che inizia dall'indice dell'ultimo carattere di *s* e termina con zero.

```
function inverti (s) {
   var t = "";
   for (var i = s.length - 1; i >= 0; i--) {
      t += s[i];
   }
   return t;
}
```

9.11 Palindromo

Un *palindromo* è una sequenza di caratteri che, letta al contrario, rimane identica [Wikipedia, alla voce *Palindromo*]. Alcuni esempi sono il nome *Ada*, la voce verbale *aveva* e la frase *I topi non avevano nipoti*.

Il predicato *ePalindromo* ha un parametro (*s*) e verifica se la stringa *s* è un palindromo.

```
function ePalindromo (s) {
   return s == inverti(s);
}
```

Un'altra soluzione che non fa uso della funzione *inverti* è la seguente.

```
function ePalindromo (s) {
   var c = 0;
   for (var i = 0; i < s.length; i++) {
      if(s[i] == s[s.length - 1 - i]) {
         c++;
      }
   }
   return (c == s.length);
}
```

Per evitare di scandire tutta la stringa quando non è un palindromo si usa un'iterazione indeterminata.

```
function ePalindromo (s) {
   var i = 0;
   while ((i < s.length) &&
          (s[i] == s[s.length - 1 - i])) {
      i++;
   }
   return (i == s.length);
}
```

9.12 Ordinamento di array

Un array può essere ordinato in senso crescente o decrescente usando un *algoritmo di ordinamento*. Il più semplice di questi algoritmi si basa sulla ricerca successiva del minimo (crescente) o del massimo (ordinamento decrescente).

```
function ordina (a) {
   for (var i = 0; i < a.length - 1; i++) {
      for (var j = i + 1; j < a.length; j++) {
         if (a[j] < a[i]) {
            var tmp = a[j];
            a[j] = a[i];
            a[i] = tmp;
         }
      }
   }
}
```

Un altro algoritmo si basa su una scansione che individua e scambia le coppie di elementi che non rispettano l'ordinamento. La scansione è ripetuta fino a quando tutti gli elementi dell'array sono ordinati.

```
function ordina (a) {
   var ordinato = false;
   while (!ordinato) {
      ordinato = true;
      for (var i = 1; i < a.length; i++) {
         if (a[i - 1] > a[i]) {
            var tmp = a[i - 1];
            a[i - 1] = a[i];
            a[i] = tmp;
            ordinato = false;
         }
      }
   }
}
```

La stessa funzione può essere riscritta usando la forma alternativa del comando iterativo indeterminato.

```
function ordina (a) {
   do {
      var scambi = 0;
      for (var i = 1; i < a.length; i++) {
         if (a[i - 1] > a[i]) {
            var tmp = a[i - 1];
            a[i - 1] = a[i];
            a[i] = tmp;
            scambi++;
         }
      }
   } while (scambi > 0);
}
```

9.13 Esercizi

1. La *media aritmetica semplice* di *n* numeri è così definita [Wikipedia, alla voce *Media (statistica)*]:

   $$m = \frac{x_1 + x_2 + \ldots + x_n}{n}$$

   Ad esempio, la media aritmetica semplice di *3*, *12*, *24* è *13*.

   Definire in JavaScript una funzione che calcola e restituisce la media aritmetica semplice degli elementi di un array *a* formato da *n* numeri.

La funzione ha un parametro: *a*.

Invocare la funzione con i seguenti valori:

[3, 12, 24]

[5, 7, 9, -12, 0].

2. Dato un array *a* e un valore *k*, il numero di occorrenze di *k* in *a* è definito come il numero degli elementi di *a* il cui valore è uguale a *k*.

   Definire in JavaScript una funzione che calcola e restituisce il numero di occorrenze del valore *k* nell'array *a*.

   La funzione ha due parametri: *a*, *k*.

   Invocare la funzione con i seguenti valori:

   [10, −5, 34, 0], 1

   [10, −5, 34, 0], −5.

3. Definire in JavaScript un predicato che verifica se ogni elemento di un array di numeri (tranne il primo) è uguale alla somma degli elementi che lo precedono.

   La funzione ha un parametro: *a*.

   Invocare la funzione con i seguenti valori:

   [1, 2, 6, 10, 32]

   [1, 1, 2, 4, 8].

4. Definire in JavaScript una funzione che ha come parametro un array *a* di numeri e che restituisce un nuovo array che contiene le differenze tra gli elementi adiacenti di *a*.

   La funzione ha un parametro: *a*.

   Invocare la funzione con i seguenti valori:

   [1, 2, -6, 0, 3]

   [2, 2, 3, 3, 4, 4].

10 Soluzione degli esercizi della prima parte

10.1 Esercizi del capitolo 4

1. Calcolare la somma dei primi quattro multipli di 13.

```
print((13 * 1) + (13 * 2) + (13 * 3) + (13 * 4));
```

2. Verificare se la somma dei primi sette numeri primi è maggiore della somma delle prime tre potenze di due.

```
print((2+3+5+7+11+13+17) > (2*1+2*2+2*2*2));
```

3. Verificare se 135 è dispari, 147 è pari, 12 è dispari, 200 è pari.

```
print(135 % 2 == 1);
print(147 % 2 == 0);
print(12  % 2 == 1);
print(200 % 2 == 0);
```

4. Calcolare l'area di un triangolo rettangolo i cui cateti sono 23 e 17.

```
print((23 * 17) / 2);
```

5. Calcolare la circonferenza di un cerchio il cui raggio è 14.

```
print(3.141592 * 2 * 14);
```

6. Calcolare l'area di un cerchio il cui diametro è 47.

```
print(3.141592 * (47 / 2) * (47 / 2));
```

7. Calcolare l'area di un trapezio la cui base maggiore è 48, quella minore è 25 e l'altezza è 13.

```
print(((48 + 25) / 2) * 13);
```

8. Verificare se l'area di un quadrato di lato quattro è minore dell'area di un cerchio di raggio tre.

```
print((4 * 4) < (3.141592 * 3 * 3));
```

9. Calcolare il numero dei minuti di una giornata, di una settimana, di un mese di 30 giorni, di un anno non bisestile.

```
print(60 * 24);
print(60 * 24 * 7);
print(60 * 24 * 30);
print(60 * 24 * 365);
```

10. Verificare se conviene acquistare una camicia che costa 63 € in un negozio che applica uno sconto fisso di 10 € o in un altro che applica uno sconto del 17%.

```
print((63 - 10) < (63 - 63 * (17 / 100)));
```

10.2 Esercizi del capitolo 5

1. Calcolare il costo di un viaggio in automobile, sapendo che la lunghezza è 750 Km, che il consumo di gasolio è 3,2 litri ogni 100 Km, che un litro di gasolio costa 1,432 €, che due terzi del percorso prevedono un pedaggio pari a 1,2 € ogni 10 Km.

```
var lunghezza = 750;
var kmGasolio = 3.2 / 100;
var ltGasolio = 1.432;
var carburante = lunghezza * kmGasolio * ltGasolio;
var lunghezzaAutostrada = lunghezza * (2 / 3);
var kmPedaggio = 1.2 / 10;
var pedaggio = lunghezzaAutostrada * kmPedaggio;
var costoTotale = carburante + pedaggio;
print(costoTotale);
```

2. Calcolare il costo di una telefonata, sapendo che la durata è pari a 4 minuti e 23 secondi, che il costo alla chiamata è pari a 0,15 €, che i primi 30 secondi sono gratis, che il resto della telefonata costa 0,24 € al minuto.

```
var minuti = 4;
var secondi = 23;
var costoFisso = 0.15;
var tariffaMinuto = 0.24;
var durata = minuti * 60 + secondi;
var tariffaSecondo = tariffaMinuto / 60;
var costoVariabile = (durata - 30) * tariffaSecondo;
var costo = costoFisso + costoVariabile;
print(costo);
```

3. Calcolare il costo di un biglietto aereo acquistato una settimana prima della partenza, sapendo che il costo di base è pari a 200 € (se acquistato il giorno della partenza) e che questo costo diminuisce del 2,3% al giorno (se acquistato prima del giorno della partenza).

```
var costoBase = 200;
var scontoGiornaliero = 2.3 / 100;
var giorni = 7;
var sconto = costoBase * scontoGiornaliero * giorni;
var costo = costoBase - sconto;
print(costo);
```

4. Calcolare il costo di un prodotto usando la seguente formula

$$costo = (prezzo + prezzo \cdot 0{,}20) - sconto$$

e sapendo che il prezzo è 100 € e lo sconto è 30 €.

```
var prezzo = 100;
var sconto = 30;
var costo = (prezzo + prezzo * 0.2) - sconto;
print(costo);
```

5. Calcolare la rata mensile di un mutuo annuale usando la seguente formula

$$rata = \frac{importo}{12} \cdot (1 + tasso)$$

e sapendo che l'importo annuale è 240 € e il tasso è il 5%.

```
var importo = 240;
var tasso = 5 / 100;
var rata = (importo / 12) * (1 + tasso);
print(rata);
```

10.3 Esercizi del capitolo 7

1. L'equazione di secondo grado $ax^2 + bx + c = 0$ ha due radici [Wikipedia, alla voce *Equazione di secondo grado*]:

$$x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Definire in JavaScript una funzione che stampa le radici di un'equazione i cui coefficienti sono *a*, *b* e *c* (con *a* diverso da zero).

La funzione ha tre parametri: *a*, *b*, *c*.

Invocare la funzione con i seguenti valori: *1*, *-5*, *6*; *1*, *8*, *16*; *1*, *2*, *3*.

```
function radici (a, b, c) {
   var delta = b*b - 4*a*c;
   if (delta > 0) {
      print((-b + Math.sqrt(delta)) / (2*a));
      print((-b - Math.sqrt(delta)) / (2*a));
   } else if (delta == 0) {
      print(-b / (2*a));
   } else { // (delta < 0)
      print("radici immaginarie");
   }
}
radici(1, -5, 6);
radici(1, 8, 16);
radici(1, 2, 3)
```

La funzione predefinita *Math.sqrt* calcola la radice quadrata del suo parametro.

2. I giorni di un anno possono essere numerati consecutivamente, assumendo che al primo gennaio sia assegnato il valore *1* e al trentuno dicembre il valore *365* negli anni non bisestili o *366* negli anni bisestili.

   Definire in JavaScript una funzione che restituisce il numero assegnato a un giorno dell'anno, assumendo che i mesi siano numerati da *1* a *12*.

   La funzione ha tre parametri: *anno*, *mese*, *giorno*.

   Invocare la funzione con i seguenti valori: *1957*, *4*, *25*; *2004*, *11*, *7*; *2000*, *12*, *31*; *2012*, *2*, *29*.

```
function numeroGiorno (anno, mese, giorno) {
   var n = giorno;
   switch (mese) {
      case 12: n += 30;
      case 11: n += 31;
      case 10: n += 30;
      case  9: n += 31;
      case  8: n += 31;
      case  7: n += 30;
      case  6: n += 31;
      case  5: n += 30;
      case  4: n += 31;
      case  3: if((anno % 400 == 0) ||
                  ((anno % 4 == 0) &&
                   (anno % 100 != 0))) {
                  n += 29;
               } else {
                  n += 28;
               }
```

```
        case  2: n += 31;
    }
    return n;
}
print(numeroGiorno(1957, 4, 25));
print(numeroGiorno(2004, 11, 7));
print(numeroGiorno(2000, 12, 31));
print(numeroGiorno(2012, 2, 29));
```

3. Definire in JavaScript una funzione che calcola e restituisce una stringa che rappresenta il valore in lettere di un intero che appartiene all'intervallo [*0, 100*).

   La funzione ha un parametro: *n*.

   Invocare la funzione con i seguenti valori: *0*, *1*, *12*, *21*, *32*, *43*, *70*, *88*, *90*.

```
function lettere (n) {
    if (n < 10) {
        return (unità[n]);
    }
    if (n < 20) {
        return (secondaDecina[n - 10]);
    }
    var u = (n % 10);
    var d = (n - u) / 10;
    var s = prefissoDecina[d];
    if (u == 0) {
        if (d == 2) {
            return prefissoDecina[d] + "i"
        } else {
            return prefissoDecina[d] + "a"
        }
    } else if (u == 1 || u == 8) {
            return prefissoDecina[d] + unità[u];
    } else if (d == 2) {
            return prefissoDecina[d] + "i" + unità[u];
    } else {
            return prefissoDecina[d] + "a" + unità[u];
    }
}
var unità = [
    "zero",
    "uno",
    "due",
    "tre",
    "quattro",
    "cinque",
```

```
    "sei",
    "sette",
    "otto",
    "nove"
]
var secondaDecina = [
    "dieci",
    "undici",
    "dodici",
    "tredici",
    "quattordici",
    "quindici",
    "sedici",
    "diciassette",
    "diciotto",
    "diciannove"
]
var prefissoDecina = [
    "",
    "",
    "vent",
    "trent",
    "quarant",
    "cinquant",
    "sessant",
    "settant",
    "ottant",
    "novant"
]
print(lettere(0));
print(lettere(1));
print(lettere(12));
print(lettere(21));
print(lettere(32));
print(lettere(43));
print(lettere(70));
print(lettere(88));
print(lettere(90));
```

4. Definire in JavaScript una funzione che calcola il tipo di un triangolo (*equilatero*, *isoscele*, *rettangolo*, *scaleno*) in base alla lunghezza dei suoi lati.

   La funzione restituisce una stringa che rappresenta il tipo del triangolo e ha tre parametri: *a*, *b*, *c*.

   Invocare la funzione con i seguenti valori: *3*, *3*, *3*; *3*, *4*, *4*; *3*, *4*, *5*; *3*, *4*, *6*.

```
function triangolo (a, b, c) {
   if ((a == b) &&
       (b == c)) {
      return "equilatero";
   } else if (((a == b) && (a != c)) ||
              ((a == c) && (a != b)) ||
              ((b == c) && (a != b))) {
      return "isoscele";
   } else if ((Math.sqrt(a*a + b*b) == c) ||
              (Math.sqrt(a*a + c*c) == b) ||
              (Math.sqrt(b*b + c*c) == a)) {
      return "rettangolo";
   } else {
      return "scaleno";
   }
}
print(triangolo(3, 3, 3));
print(triangolo(3, 4, 4));
print(triangolo(3, 4, 5));
print(triangolo(3, 4, 6));
```

5. Definire in JavaScript un predicato che verifica se l'intersezione dell'intervallo [*a*, *b*) con l'intervallo [*c*, *d*) è vuota.

   Il predicato ha quattro parametri: *a*, *b*, *c*, *d*.

   Invocare il predicato con i seguenti valori: *2*, *4*, *5*, *7*; *2*, *4*, *4*, *7*; *2*, *4*, *3*, *7*; *5*, *7*, *2*, *4*; *4*, *7*, *2*, *4*; *3*, *7*, *2*, *4*.

```
function intersezione (a, b, c, d) {
   return ((b <= c) || (d <= a));
}
print(intersezione(2, 4, 5, 7));
print(intersezione(2, 4, 4, 7));
print(intersezione(2, 4, 3, 7));
print(intersezione(5, 7, 2, 4));
print(intersezione(4, 7, 2, 4));
print(intersezione(3, 7, 2, 4));
```

6. Definire in JavaScript una funzione che calcola e restituisce la somma delle cifre di un intero che appartiene all'intervallo [*0*, *100*).

   La funzione ha un parametro: *n*.

   Invocare la funzione con i seguenti valori: *0*, *1*, *23*, *99*.

```
function sommaCifre (n) {
   var u = n % 10;
   var d = (n - u) / 10;
   return (u + d);
```

```
}
print(sommaCifre(0));
print(sommaCifre(1));
print(sommaCifre(23));
print(sommaCifre(99));
```

10.4 Esercizi del capitolo 8

1. Un anno è perfetto per una persona se è un multiplo della sua età in quell'anno, assumendo come età massima cento anni. Ad esempio gli anni perfetti per chi è nato nel *1984* sono nove: *1985*, *1986*, *1988*, *1992*, *2000*, *2015*, *2016*, *2046*, *2048*.

   Definire in JavaScript una funzione che stampa gli anni perfetti relativi a un anno di nascita.

   La funzione ha un parametro: *annoDiNascita*.

   Invocare la funzione con i seguenti valori: *1984*, *1990*, *1992*, *2000*, *2009*.

```
function anniPerfetti (annoDiNascita) {
   const ETA_MASSIMA = 100;
   for (var i = 1; i <= ETA_MASSIMA; i++) {
      if (((annoDiNascita + i) % i) == 0) {
         print(annoDiNascita + i);
      }
   }
}
anniPerfetti(1984);
anniPerfetti(1990);
anniPerfetti(1992);
anniPerfetti(2000);
anniPerfetti(2009);
```

2. In matematica, con reciproco di un numero *x* si indica il numero che moltiplicato per *x* dà come risultato 1.

   Definire in JavaScript una funzione che calcola e restituisce la somma dei reciproci dei primi *n* numeri interi maggiori di zero.

   La funzione ha un parametro: *n*.

   Invocare la funzione con i seguenti valori: *1*, *2*, *4*, *7*.

```
function armonica (n) {
   var s = 0;
   for (var i = 1; i <= n; i++) {
      s += 1 / i;
   }
   return s;
```

```
}
print(armonica(1));
print(armonica(2));
print(armonica(4));
print(armonica(7));
```

3. Definire in JavaScript una funzione che stampa, se esistono, le radici intere di un'equazione di secondo grado di coefficienti *a*, *b*, *c* comprese nell'intervallo [*l*, *u*).

   La funzione ha i seguenti parametri: *a*, *b*, *c*, *l*, *u*.

   Invocare la funzione con i seguenti valori:

   1, −2, −8, 1, 5

   1, −2, −8, −5, 5

   1, −2, −8, 5, 10.

```
function radici(a, b, c, l, u) {
   for (var i = l; i < u; i++) {
      if (a * i * i + b * i + c == 0) {
         print(i);
      }
   }
}
radici(1, -2, -8, 1, 5);
radici(1, -2, -8, -5, 5);
radici(1, -2, -8, 5, 10);
```

10.5 Esercizi del capitolo 9

1. La *media aritmetica semplice* di *n* numeri è così definita [Wikipedia, alla voce *Media (statistica)*]:

$$m = \frac{x_1 + x_2 + \ldots + x_n}{n}$$

   Ad esempio, la media aritmetica semplice di *3*, *12*, *24* è *13*.

   Definire in JavaScript una funzione che calcola e restituisce la media aritmetica semplice degli elementi di un array *a* formato da *n* numeri.

   La funzione ha un parametro: *a*.

   Invocare la funzione con i seguenti valori:

   [3, 12, 24]

   [5, 7, 9, -12, 0].

```
function media (a) {
   var s = 0;
   for (var i = 0; i < a.length; i++) {
      s += a[i];
   }
   return (s / a.length);
}
print(media([3, 12, 24]));
print(media([5, 7, 9, -12, 0]));
```

2. Dato un array *a* e un valore *k*, il numero di occorrenze di *k* in *a* è definito come il numero degli elementi di *a* il cui valore è uguale a *k*.

   Definire in JavaScript una funzione che calcola e restituisce il numero di occorrenze del valore *k* nell'array *a*.

   La funzione ha due parametri: *a*, *k*.

   Invocare la funzione con i seguenti valori:

   [10, −5, 34, 0], 1

   [10, −5, 34, 0], −5.

```
function occorrenze (a, k) {
   var c = 0;
   for (var i = 0; i < a.length; i++) {
      if (a[i] == k) {
         c++;
      }
   }
   return c;
}
print(occorrenze([10, -5, 34, 0], 1));
print(occorrenze([10, -5, 34, 0], -5));
```

3. Definire in JavaScript un predicato che verifica se ogni elemento di un array di numeri (tranne il primo) è uguale alla somma degli elementi che lo precedono.

   La funzione ha un parametro: *a*.

   Invocare la funzione con i seguenti valori:

   [1, 2, 6, 10, 32]

   [1, 1, 2, 4, 8].

```
function verificaSomma (a) {
   var c = a[0];
   var i = 1;
   while ((i < a.length) && (a[i] == c)) {
      c += a[i];
```

```
        i++;
    }
    return (i == a.length);
}
print(verificaSomma([1, 2, 6, 10, 32]));
print(verificaSomma([1, 1, 2, 4, 8]));
```

4. Definire in JavaScript una funzione che ha come parametro un array *a* di numeri e che restituisce un nuovo array che contiene le differenze tra gli elementi adiacenti di *a*.

   La funzione ha un parametro: *a*.

   Invocare la funzione con i seguenti valori:

   [1, 2, -6, 0, 3]

   [2, 2, 3, 3, 4, 4].

```
function differenze (a) {
    var b = [];
    for (var i = 0; i < a.length - 1; i++) {
        b.push(a[i] - a[i + 1]);
    }
    return b;
}
print(differenze([1, 2, -6, 0, 3]));
print(differenze([2, 2, 3, 3, 4, 4]));
```

Parte seconda

Programmazione web

11 Ricorsione

In JavaScript è possibile definire *funzioni ricorsive*. Una funzione è ricorsiva quando nel suo corpo compare un'invocazione alla funzione stessa. Il concetto di ricorsione è stato già introdotto nella definizione delle grammatiche: una produzione è ricorsiva quando la definizione di un simbolo non-terminale contiene il simbolo non-terminale stesso.

Per definire una funzione ricorsiva non è necessario modificare le regole sintattiche viste finora. Il capitolo presenta alcuni problemi la cui soluzione può essere definita naturalmente mediante una funzione ricorsiva.

11.1 Fattoriale

Il *fattoriale* di *n* (indicato con *n*!) è il prodotto dei primi *n* numeri interi positivi minori o uguali di quel numero [Wikipedia, alla voce *Fattoriale*]. 0! vale 1, 1! vale 1, 2! vale 2, 3! vale 6, 4! vale 24 e così via.

La funzione *fattorialeR* ha un parametro (*n*) e restituisce il fattoriale di *n*. La funzione è definita in maniera ricorsiva, sfruttando la definizione matematica del fattoriale.

```
function fattorialeR (n) {
   if ((n == 0) || (n == 1)) {
      return 1;
   } else {
      return n * fattorialeR(n - 1);
   }
}
```

Il calcolo del fattoriale può essere effettuato anche con una funzione iterativa, che non fa uso della ricorsione. Anche in questo caso la funzione ha un parametro (*n*) e restituisce il fattoriale di *n*.

```
function fattorialeI (n) {
   var r = 1;
   for (var i = 1; i <= n; i++) {
      r *= i;
   }
   return r;
}
```

11.2 Successione di Fibonacci

La *successione di Fibonacci* [Wikipedia, alla voce *Successione di Fibonacci*] è una sequenza di numeri interi naturali così definita: i primi due termini valgono zero e uno, rispettivamente; i termini successivi sono definiti come la somma dei due precedenti. Storicamente il primo termine della successione non era zero ma uno.

La sequenza prende il nome dal matematico pisano del XIII secolo *Leonardo Fibonacci* e i termini di questa successione sono chiamati *numeri di Fibonacci*. Definendo questa successione, Fibonacci voleva stabilire una legge che descrivesse la crescita di una popolazione di conigli. Fibonacci fece alcune assunzioni: la prima coppia diventa fertile al compimento del primo mese e dà alla luce una nuova coppia al compimento del secondo mese; le coppie nate nei mesi successivi si comportano in modo analogo; le coppie fertili, dal secondo mese di vita, danno alla luce una coppia di conigli al mese.

Per verificare la validità della successione si parte con una singola coppia di conigli, che dopo un mese sarà fertile. Dopo due mesi ci saranno due coppie di conigli, di cui una sola fertile. Nel mese seguente le coppie saranno tre (*2 + 1 = 3*), perché solo la coppia fertile avrà partorito. Di queste tre, ora saranno due le coppie fertili e, quindi, nel mese seguente ci saranno (*3 + 2 = 5*) cinque coppie. Come si può vedere, il numero di coppie di conigli di ogni mese coincide con i valori della successione di Fibonacci.

In base a queste considerazioni si può definire una funzione ricorsiva che ha come parametro un intero *n*, maggiore o uguale a zero, e che restituisce il corrispondente numero della successione di Fibonacci (assumendo che il primo numero della successione abbia indice zero).

```
function fibonacciR (n) {
   if ((n == 0) || (n == 1)) {
      return n;
   } else {
      return fibonacciR(n - 2) +
             fibonacciR(n - 1);
   }
}
```

Il calcolo di un elemento della successione di Fibonacci può essere effettuato definendo un'altra funzione, questa volta iterativa, più complessa e meno intuitiva ma molto più efficiente in termini di numero di operazioni elementari (ad esempio le somme) necessarie per effettuare il calcolo.

```
function fibonacciI (n) {
   if ((n == 0) || (n == 1)) {
```

```
      return n;
   }
   var f1 = 0;
   var f2 = 1;
   var f;
   for (var i = 2; i <= n; i++) {
       f = f1 + f2;
       f1 = f2;
       f2 = f;
   }
   return f;
}
```

11.3 Aritmetica di Peano

Gli *assiomi di Peano* sono stati ideati dal matematico *Giuseppe Peano* per definire l'insieme dei numeri naturali. Informalmente tali assiomi possono essere così enunciati [Wikipedia, alla voce *Assiomi di Peano*]:

- esiste un numero naturale, lo *zero*
- ogni numero naturale ha un numero naturale *successore*
- numeri diversi hanno successori diversi
- *zero* non è il successore di alcun numero naturale
- ogni insieme di numeri naturali che contiene lo *zero* e il successore di ogni suo elemento coincide con tutto l'insieme dei numeri naturali (induzione).

L'*aritmetica di Peano* è una *teoria del prim'ordine* basata su una versione degli a*ssiomi di Peano* espressi nel linguaggio del prim'ordine [Wikipedia, alla voce *Aritmetica di Peano*]. In questa aritmetica le operazioni sono definite per *induzione* usando la costante *zero*, l'operatore unario *successore*[8], le relazioni di *uguaglianza* e *disuguaglianza* con *zero*.

Le operazioni di *addizione* (indicata con $\underline{+}$) e di *moltiplicazione* (indicata con $\underline{\times}$) sono così definite per induzione:

$x \underline{+} 0 = x$

$x \underline{+} y = x \underline{+} y - 1 + 1 = (x \underline{+} (y - 1)) + 1 \text{ se } y \neq 0$

$x \underline{\times} 0 = 0$

$x \underline{\times} y = x \underline{\times} (y - 1 + 1) = (x \underline{\times} (y - 1)) \underline{+} x \text{ se } y \neq 0$

8 Per semplicità di presentazione, l'operatore unario successore è stato sostituito dagli operatori unari di incremento (*+1*) e decremento (*–1*) unitario.

In JavaScript è possibile definire ricorsivamente le funzioni che realizzano queste operazioni usando esclusivamente la costante *0*, l'operatore binario di uguaglianza (==), l'operatore binario di disuguaglianza (!=), l'operatore di incremento unitario (*+1*) e l'operatore di decremento unitario (*–1*).

```
function addizione (x, y) {
   if (y == 0) {
      return x;
   } else {
      return addizione(x, y -1) +1;
   }
}
function moltiplicazione (x, y) {
   if (y == 0) {
      return 0;
   } else {
      return addizione(moltiplicazione(x, y -1), x);
   }
}
```

L'operazione di *sottrazione*[9] (indicata con *S*) è così definita per induzione:

- $x \; S \; 0 = x$
- $x \; S \; y = (x \; S \; y) + 1 - 1 = ((x \; S \; y) + 1) - 1 = (x \; S \; (y - 1)) - 1 \; se \; y \neq 0$

L'operazione di *potenza* è così definita per induzione:

- $x^0 = 1 \; se \; x \neq 0$
- $x^y = x \times x^{y-1} \; se \; x \neq 0 \; e \; y \neq 0$
- $0^y = 0 \; se \; y \neq 0$

Da notare che 0^0 non è definito.

Per le operazioni *divisione* e *resto* è necessaria la definizione induttiva dell'operatore di relazione <:

- $x < 0 = false$
- $0 < y = true \; se \; y \neq 0$
- $x < y = x - 1 < y - 1 \; se \; x \neq 0 \; e \; y \neq 0$

Le operazioni *divisione* (indicata con *D*) e *resto* (indicata con *R*) sono così definite per induzione:

9 L'operazione è definita se e solo se $x \geq y$.

- $x\ D\ y = 0$ *se* $x < y$
- $x\ D\ y = 1$ *se* $x = y$
- $x\ D\ y = (x\ S\ y \pm y)\ D\ y = (x\ S\ y)\ D\ y \pm y\ D\ y = ((x\ S\ y)\ D\ y) + 1$ *se* $y < x$

Da notare che $x\ D\ 0$ non è definito.

- $x\ R\ y = x$ *se* $x < y$
- $x\ R\ y = 0$ *se* $x = y$
- $x\ R\ y = (x\ S\ y)\ R\ y$ *se* $y < x$

Anche in questo caso $x\ R\ 0$ non è definito.

11.4 Esercizi

1. Definire in Javascript una funzione ricorsiva che calcola e restituisce la differenza tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   5, 0

   3, 3

   9, 2.
2. Definire in JavaScript una funzione ricorsiva che calcola e restituisce la relazione di minore tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   0, 0

   0, 4

   3, 0

   2, 6

   9, 2.
3. Definire in JavaScript una funzione ricorsiva che calcola e restituisce la divisione tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   0, 4

   3, 3

   9, 2.

4. Definire in JavaScript una funzione ricorsiva che calcola e restituisce il resto della divisione tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: x, y.

   Invocare la funzione con le seguenti coppie di valori:

   0, 4

   3, 3

   9, 2.

5. Definire in JavaScript una funzione ricorsiva che calcola e restituisce l'operatore potenza definito tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: x, y.

   Invocare la funzione con le seguenti coppie di valori:

   6, 0

   3, 3

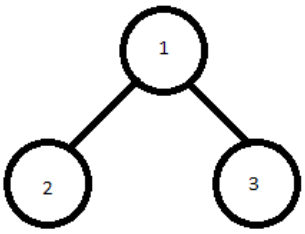
   0, 2.

12 Alberi

Nella teoria dei grafi un *albero* è un *grafo non orientato* nel quale due *vertici* qualsiasi sono connessi da uno e un solo *cammino* (grafo non orientato connesso e privo di cicli) [Wikipedia, alla voce *Albero (grafo)*].

12.1 Alberi binari

Un *albero binario* è un albero i cui nodi possono avere al massimo due *sottoalberi* o *figli*. Un nodo senza figli è anche chiamato *foglia*. Un nodo che non ha un *nodo padre* è chiamato *radice*. A ogni nodo dell'albero è associato un valore.



Un albero può essere visualizzato mostrando i suoi valori in sequenza. La costruzione della sequenza dei valori si effettua mediante una *visita* dei nodi che, nel caso degli alberi binari, può essere definita in tre modi diversi: *visita anticipata*, *visita differita*, *visita simmetrica*.

La visita anticipata costruisce una sequenza in cui prima compare il valore della radice seguito dalla sequenza prodotta dalla visita anticipata del figlio sinistro e da quella della visita anticipata del figlio destro. La visita anticipata dell'albero binario mostrato nella figura precedente è *1 2 3*.

La visita differita costruisce una sequenza in cui prima compare la visita anticipata del figlio sinistro seguita dalla visita anticipata del figlio destro e, alla fine, dal valore della radice. La visita differita dell'albero binario mostrato nella figura precedente costruisce la sequenza *2 3 1*.

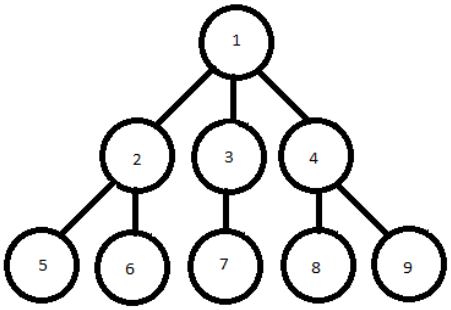
La visita simmetrica costruisce una sequenza in cui prima compare la visita anticipata del figlio sinistro, poi il valore della radice e, alla fine, la visita anticipata del figlio destro. La visita simmetrica dell'albero binario mostrato nella figura precedente costruisce la sequenza *2 1 3*.

L'*altezza* di un albero è la lunghezza del massimo cammino dalla radice a una foglia. In particolare, l'altezza di un albero formato da un solo nodo radice è uguale a zero. L'altezza dell'albero binario mostrato nella figura precedente è *1*.

La *frontiera* di un albero è la sequenza dei valori delle foglie dell'albero, visitate da sinistra verso destra. La frontiera dell'albero binario mostrato nella figura precedente è la sequenza *2 3*.

12.2 Alberi n-ari

Un *albero n-ario* è un albero i cui nodi, a differenza di quelli degli alberi binari, possono avere un numero qualsiasi di figli. Come per gli alberi binari, i nodi di un albero n-ario hanno un valore associato.



Un'altra differenza con gli alberi binari consiste nel fatto che per un albero n-ario sono definite solo due visite: anticipata e differita. La visita simmetrica, infatti, non avrebbe senso. La visita anticipata di un albero n-ario si comporta analogamente a quella di un albero binario. La visita anticipata dell'albero n-ario mostrato nella figura precedente costruisce la sequenza *1 2 5 6 3 7 4 8 9*. Anche la visita differita di un albero n-ario si comporta analogamente a quella di un albero binario. La visita differita dell'albero n-ario mostrato nella figura precedente costruisce la sequenza *5 6 2 7 3 8 9 4 1*.

Le definizioni di altezza e di frontiera di un albero n-ario sono equivalente a quelle di un albero binario. L'altezza dell'albero n-ario mostrato nella figura precedente è 3, la sua frontiera è la sequenza *5 6 7 8 9*.

Un *albero n-ario con attributi* è un albero n-ario esteso associando uno o più attributi ad ogni nodo. Ogni attributo ha un nome e un valore associato.

12.3 Esercizi

1. Ogni nodo di un albero binario può essere rappresentato mediante un array associativo con tre elementi, identificati da *val*, *sx*, *dx*. Il primo elemento rappresenta il *valore* associato al nodo, il secondo rappresenta il *figlio sinistro* del nodo, il terzo rappresenta il *figlio destro* del nodo. Per

convenzione, l'assenza di un figlio si rappresenta con il valore *null*. Definire in JavaScript una funzione che ha come parametro un nodo, che rappresenta la radice di un albero binario. La funzione visita in ordine anticipato l'albero binario, a partire dalla sua radice, e restituisce una stringa che rappresenta, nell'ordine, la sequenza dei valori dei nodi visitati, separati da uno spazio bianco.

Invocare la funzione con un parametro attuale che rappresenta un albero binario formato da tre nodi, di cui uno è la radice e gli altri due sono foglie.

13 Document Object Model

Un sito web è formato da pagine definite in HTML e rappresentate come alberi DOM. Questo capitolo presenta il linguaggio HTML e l'interfaccia programmatica DOM.

13.1 HTML

HTML (*HyperText Mark-up Language*) è un linguaggio per la descrizione strutturale di *documenti*, usato prevalentemente per definire i *siti web*. La sua presentazione è volutamente molto ridotta: il lettore interessato ha a disposizione un'ampia letteratura su HTML, utile per approfondirne la conoscenza.

HTML non è un linguaggio di programmazione, in quanto non descrive algoritmi, ma si basa sull'uso di *etichette* o *marche* (*tag*) per la *marcatura* di porzioni di testo di un *documento HTML*[10]. Una marca è un elemento inserito nel testo di un documento per indicarne la struttura. Una marca è formata da una stringa racchiusa tra *parentesi angolari* ("<" e ">"). Subito dopo la parentesi angolare aperta si trova il nome della marca, seguito da eventuali attributi, che ne specificano le caratteristiche.

Poiché ogni marca determina la formattazione di una parte di un documento è necessario che questa parte sia indicata da una *marca di apertura*, che segna l'inizio della porzione di testo da formattare, e da una *marca di chiusura*, che ne segna la fine. La marca di chiusura è formata dal nome della marca, preceduto dal simbolo "/". Nel testo racchiuso dalle due marche (ovvero compreso tra quella di apertura e quella di chiusura) possono comparire altre marche il cui effetto si aggiunge a quello delle marche più esterne.

Ogni marca ha un diverso insieme di possibili attributi, dipendente dalla funzione della marca stessa. Alcune marche hanno uno o più attributi obbligatori, senza i quali non possono svolgere la loro funzione. La sintassi della definizione di un attributo è: *NomeAttributo*="*valore*". Il valore dell'attributo è racchiuso tra doppi apici.

Un documento è formato da un testo racchiuso dalla marca *html* e si divide in due parti: *intestazione* e *corpo*. L'intestazione, racchiusa dalla marca *head*, ri-

10 Nel seguito, per semplicità, i documenti HTML saranno chiamati documenti quando il contesto non crea ambiguità.

porta il *titolo* del documento. Il corpo rappresenta il documento vero e proprio e contiene sia il testo, sia le marche di formattazione. Il corpo del documento è racchiuso dalla marca *body*.

Lo standard HTML 5 prevede che prima della marca *html* ci sia una dichiarazione DOCTYPE e che nella parte *head* ci sia una marca *meta* che indica l'insieme dei caratteri da usare per la visualizzazione del documento.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>Titolo del documento</title>
 </head>
 <body>
  Contenuto del documento
<!-- questo è un commento -->
 </body>
</html>
```

La visualizzazione di un documento HTML è effettuata da un browser che, dopo averlo caricato, crea una pagina web che ne mostra il testo formattato secondo le marche utilizzate. Il titolo del documento è riportato nella *barra del browser*.

13.2 Script

Un programma JavaScript può essere associato a un documento HTML tramite la marca *script*, che può comparire sia nell'intestazione che nel corpo del documento. In questo libro la marca *script* è sempre usato solo nell'intestazione. Il programma JavaScript ha la funzione di rendere interattivo il comportamento della pagina web visualizzata dal browser.

Un programma JavaScript può essere associato a un documento in due modi:

- inserendolo direttamente tra la marca *script* di apertura e di chiusura;
- scrivendolo in un file separato che ha, per convenzione, *estensione* ".js" e indicando il nome del file come valore dell'attributo *src*.

Come mostrato nell'esempio che segue, in questo libro sarà sempre utilizzata la seconda tecnica. Da notare che la marca *script* è chiusa esplicitamente con una marca di chiusura. A differenza di altre marche, infatti, per la marca *script* l'uso della marca esplicita di chiusura è rigorosamente obbligatorio.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
```

```
  <script src="nomeFile.js"></script>
  <title>Titolo del documento</title>
 </head>
 <body>
  Contenuto del documento
 </body>
</html>
```

Il programma JavaScript è caricato ed eseguito durante la visualizzazione della pagina web. Poiché è possibile associare più di un file JavaScript, l'ordine di esecuzione rispetta l'ordine secondo il quale questi file sono inseriti nel documento HTML.

13.3 DOM

DOM (*Document Object Model*, modello a oggetti del documento), è un'*interfaccia programmatica* per la rappresentazione di documenti. DOM è uno standard ufficiale del *W3C* (*World Wide Web Consortium*) utilizzato per la programmazione di *applicazioni web* basate su documenti in formato HTML [Wikipedia, alla voce *Document Object Model*].

Secondo il DOM, un documento è rappresentato da un albero che può essere visitato, modificato e trasformato mediante strumenti specifici o programmi scritti a tal scopo. Un *albero DOM*[11] è un albero n-ario con attributi. I nodi dell'albero sono di varia natura:

- *nodi documento* (document nodes)
- *nodi elemento* (element nodes)
- *nodi testo* (text nodes)
- *nodi attributo* (attribute nodes).

La struttura principale di un albero è formata da nodi elemento che possono avere un numero variabile di figli. La radice è sempre un nodo documento e i nodi testo possono essere solo foglie di un albero. I nodi attributo definiscono gli attributi dei nodi elemento. La radice dell'albero associato a un documento corrisponde all'oggetto *document*.

Ogni nodo ha le seguenti proprietà, oltre ad altre descritte nel seguito:

- *nodeType*
- *nodeName*
- *nodeValue*
- *childNodes*

11 Nel seguito, per semplicità, gli alberi DOM saranno chiamati alberi quando il contesto non crea ambiguità.

- *attributes*.

Il *tipo* di un nodo, definito dalla proprietà *nodeType*, può avere i seguenti valori

- nodo elemento: *1*;
- nodo attributo: *2*;
- nodo testo: *3*;
- nodo documento: *9*.

Il *nome* di un nodo, definito dalla proprietà *nodeName*, ha un significato che dipende dal tipo del nodo:

- nodo elemento: marca HTML dell'elemento;
- nodo attributo: nome dell'attributo;
- nodo testo: valore predefinito *#text*;
- nodo documento: valore predefinito *#document*.

Il *valore* di un nodo, definito dalla proprietà *nodeValue*, ha un significato che dipende dal tipo del nodo:

- nodo elemento: *null*;
- nodo attributo: valore dell'attributo;
- nodo testo: stringa associata al nodo;
- nodo documento: *null*.

Il valore della proprietà *childNodes* è un array che contiene i figli di un nodo elemento o di un nodo documento, *null* negli altri casi. Se un nodo non ha figli, il valore della proprietà è un array vuoto.

Il valore della proprietà *attributes* è un array che contiene gli attributi di un nodo elemento o di un nodo documento. Se un nodo non ha attributi, il valore della proprietà *attributes* è *null*.

13.4 Navigazione

Il modello DOM prevede diverse tecniche per visitare gli alberi e per muoversi (navigare) tra i suoi nodi. Queste tecniche si basano sul valore della proprietà *childNodes* e di altre proprietà che, quando sono definite (cioè quando il loro valore non è *null*), collegano un nodo ai suoi nodi vicini:

- *firstChild*: primo figlio del nodo;
- *lastChild*: ultimo figlio del nodo;
- *firstSibling*: primo fratello del nodo;

- *lastSibling*: ultimo fratello del nodo;
- *previousSibling*: fratello precedente del nodo;
- *nextSibling*: fratello successivo del nodo;
- *parentNode*: padre del nodo.

La visita dei figli di un nodo (la variabile *nodo* nel seguito) si effettua con un'iterazione determinata che usa la proprietà *childNodes*.

```
for (var i = 0; i < nodo.childNodes.length; i++) {
   var nodoFiglio = nodo.childNodes[i];
}
```

In alternativa, la stessa visita può essere effettuata con un'iterazione indeterminata, utilizzando le proprietà *firstSibling* e *nextSibling*.

```
var nodoFiglio = nodo.firstChild;
while (nodoFiglio != null) {
   nodoFiglio = nodoFiglio.nextSibling;
}
```

La visita può essere effettuata anche in ordine inverso, partendo dall'ultimo figlio del nodo e arrivando al primo. In questo caso si usano le proprietà *lastSibling* e *previousSibling*.

```
var nodoFiglio = nodo.lastChild;
while (nodoFiglio != null) {
   nodoFiglio = nodoFiglio.previousSibling;
}
```

Per visitare un albero partendo da un nodo fino ad arrivare alla sua radice si usa la proprietà *parentNode*. Nell'esempio che segue la visita parte dalla radice del nodo, procede scegliendo il primo figlio e poi torna indietro fino a ritornare alla radice.

```
while (nodo.firstChild != null) {
   nodo = nodo.firstChild;
}
while (nodo.parentNode != null) {
   nodo = nodo.parentNode;
}
```

13.5 Ricerca

La ricerca di un nodo si effettua utilizzando i seguenti metodi:

- *getElementsByTagName*

- *getElementsByClassName*
- *getElementById*.

Il metodo *getElementsByTagName* ha un unico argomento, una stringa *t* e può essere invocato su un nodo elemento o su un nodo documento. Il metodo effettua una visita anticipata dell'albero radicato nel nodo su cui è stato invocato e raccoglie in una collezione tutti e soli gli elementi con marca *t*, ovvero tutti e soli i nodi la cui proprietà *nodeName* ha un valore uguale a *t*. Alla fine della visita il metodo restituisce la collezione (possibilmente vuota).

Nell'esempio che segue la ricerca parte dalla radice dell'albero. L'iterazione determinata[12] scandisce la collezione così ottenuta.

```
var collezione = document.getElementsByTagName("script");
var nomiFile = [];
for (var i = 0; i < collezione.length; i++) {
   nomiFile.push(collezione[i].getAttribute("src"));
}
```

Il metodo *getElementsByClassName* è simile a *getElementsByTagName*, con la differenza che raccoglie tutti in nodi associati alla stessa classe[13].

Il metodo *getElementById* ha un unico argomento *v* e può essere invocato su un nodo elemento o su un nodo documento. Nell'esempio che segue il metodo effettua una visita anticipata a partire dalla radice dell'albero e restituisce il nodo il cui attributo *id* ha un valore uguale alla stringa a*lfa*, *null* altrimenti.

```
var nodo = document.getElementById("alfa");
```

13.6 Attributi

Gli attributi di un nodo sono gestiti dai seguenti metodi:

- *getAttribute*
- *setAttribute*
- *removeAttribute*.

Il metodo *getAttribute* ha un unico argomento, una stringa *a*. Può essere invocato su un nodo elemento o un nodo documento e restituisce il valore dell'attributo il cui nome è *a*, *undefined* altrimenti.

```
var valore = nodo.getAttribute("a");
```

12 La scansione deve essere effettuata indicando esplicitamente la lunghezza della collezione.
13 In questo contesto la classe è definita in un *foglio di stile*.

Il metodo *setAttribute* ha due argomenti: il nome *a* di un attributo e il valore *v*. Il metodo modifica il valore dell'attributo *a* assegnandogli il valore *v*. Se l'attributo *a* non esiste, il metodo lo crea e gli assegna il valore *v*.

```
nodo.setAttribute("a", 123);
nodo.setAttribute("b", true);
```

Il metodo *removeAttribute* ha un parametro, il nome *a* dell'attributo che si intende eliminare dagli attributi nel nodo su cui si invoca il metodo.

```
nodo.removeAttribute("a");
```

13.7 Creazione e modifica

Un albero può essere modificato in due modi diversi: creando un nuovo nodo e aggiungendolo all'albero, eliminando un nodo esistente. Un nodo può essere creato usando i seguenti metodi:

- *createElement*
- *createTextNode*.

Il metodo *createElement* ha un parametro, una stringa, che determina il nome del nodo elemento che si vuole creare. Il metodo *createTextNode* ha un parametro, una stringa, che determina il valore del nodo testo che si vuole creare. Questi due metodi devono essere invocati sull'oggetto *document*.

```
var nodo1 = document.createElement("br");
var nodo2 = document.createTextNode("alfa");
```

Dopo aver creato un nuovo nodo è possibile aggiungerlo all'albero, utilizzando uno dei seguenti metodi:

- *appendChild*
- *insertBefore*
- *insertAfter*.

Il metodo *appendChild* ha un parametro, un nodo elemento, che è aggiunto in coda ai figli del nodo elemento su cui si invoca il metodo.

```
var nodo = document.createElement("br");
nodo.appendChild(nodo);
```

Il metodo *insertBefore* ha due parametri, il nodo che si intende inserire e il nodo prima del quale deve essere inserito. Il metodo è invocato sul padre del secondo parametro.

```
var nodo1 = document.createTextNode("alfa");
nodo.insertBefore(nodo1, nodo.firstChild);
```

Il metodo *insertAfter* ha due parametri, il nodo che si intende inserire e il nodo dopo il quale deve essere inserito. Il metodo è invocato sul padre del secondo parametro.

```
var nodo1 = document.createTextNode("alfa");
nodo.insertAfter(nodo1, nodo.firstChild);
```

Un nodo, e tutto l'albero di cui tale nodo è la radice, può essere rimosso usando i seguenti metodi:

- *removeChild*
- *replaceChild*.

Il metodo *removeChild* ha un parametro, il nodo che si intende rimuovere. Il metodo deve essere invocato sul padre del nodo che si intende eliminare.

```
nodo.removeChild(nodo.firstChild);
```

Il metodo *replaceChild* ha due parametri, il nodo che si intende inserire e il nodo che si intende sostituire. Il metodo deve essere invocato sul padre del nodo che si intende sostituire.

```
var nodo1 = document.createElement("br");
nodo.replaceChild(nodo1, nodo.firstChild);
```

In alcune situazioni è necessario generare dinamicamente una parte di un documento. Anziché creare un albero DOM mediante i metodi di creazione appena visti e collegarlo al nodo di un albero DOM già esistente, è possibile creare il nuovo albero a partire da una stringa che rappresenta un frammento di codice HTML e assegnare questa stringa alla proprietà *innerHTML* del nodo. La stringa è trasformata in un albero DOM e tale albero è inserito come figlio del nodo.

```
nodo.innerHTML = "<br />Una riga dinamica<br />";
```

Pur non essendo definita nello standard *W3C* questa proprietà è, di fatto, gestita dalla maggior parte dei browser[14].

13.8 Esercizi

1. Definire una funzione JavaScript che visita un albero DOM e che restituisce il numero dei suoi nodi di tipo testo. Invocare la funzione usando come argomento la radice dell'albero DOM corrispondente alla pagina web dell'ambiente *EasyJS*.

14 In questo libro la proprietà *innerHTML* non è utilizzata negli esempi e negli esercizi.

14 Interattività

Una delle caratteristiche più interessanti delle pagine web è la loro *interattività*, cioè la capacità di reagire ad azioni effettuate dall'utente. Questo capitolo presenta i concetti e le tecniche di base necessarie per rendere interattiva una pagina web.

14.1 Eventi

Un *evento* è qualcosa che accade in seguito a un'azione di un utente che sta visualizzando un documento HTML. Gli eventi sono raggruppati in base ai seguenti aspetti che ne causano la *generazione*:

- tasti del *mouse*
- movimenti del mouse
- trascinamento del mouse
- tastiera
- modifiche del documento
- cambiamento del *focus*
- caricamento degli oggetti
- movimenti delle finestre
- pulsanti speciali.

Nel seguito è presentata una parte degli eventi, quelli più comuni. Il lettore interessato può trovare l'elenco completo degli eventi consultando la relativa letteratura.

Gli eventi generati dai tasti del mouse sono:

- *click*: generato quando si clicca su un oggetto;
- *dblClick*: generato con un doppio click;
- *mouseDown*: generato quando si schiaccia il tasto sinistro del mouse;
- *mouseUp*: generato quando si alza il tasto sinistro del mouse precedentemente schiacciato.

Gli eventi *mouseDown* e *mouseUp* sono generati dai due movimenti del tasto sinistro del mouse, il primo quando si preme il tasto e il secondo quando lo si solleva dopo il click. Il doppio click è un evento che ingloba gli altri e, per la precisione, genera in successione *mouseDown*, *mouseUp*, *click*.

Gli eventi generati dai movimenti del mouse sono:

- *mouseOver*: generato quando il mouse si muove su un elemento;
- *mouseOut*: generato quando il mouse si sposta fuori da un elemento;

Gli eventi *mouseOver* e *mouseOut* sono complementari: il primo è generato nel momento in cui il puntatore è posto nell'area dell'oggetto e il secondo quando ne esce.

Gli eventi generati dalla tastiera sono:

- *keyPress*: generato quando si preme e si rilascia un tasto o anche quando lo si tiene premuto;
- *keyDown*: generato quando si preme un tasto;
- *keyUp*: generato quando un tasto, che era stato premuto, viene rilasciato.

Gli eventi legati al caricamento delle pagine sono:

- *load*: generato quando si carica una pagina;
- *unload*: è l'opposto del precedente ed è generato quando si lascia una finestra per caricarne un'altra o per ricaricare la stessa (col tasto *refresh*).

14.2 Gestione delle eccezioni

Durate l'esecuzione di un programma JavaScript possono verificarsi degli *errori dinamici*, detti anche errori a *run time* o *eccezioni*. Ad esempio, il programma può accedere agli elementi di un array senza averlo inizializzato correttamente. Se si verifica un errore dinamico l'esecuzione del programma è interrotta e l'errore è segnalato nell'apposita *console* del browser[15].

Per controllare queste situazioni, che in un programma corretto non dovrebbero verificarsi mai, si usa il comando *try* la cui sintassi è riportata nel seguito.

```
<ComandoComposto> ::= <Try>
<Try> ::= try {<Blocco>}
          catch (<Identificatore>) {<Blocco>}
        | try {<Blocco>}
          catch (<Identificatore>) {<Blocco>}
          finally {<Blocco>}
```

15 Ogni browser ha la sua console in cui sono visualizzati gli errori dinamici. In *Firefox* questa console (chiamata *Console web*) può essere aperta pigiando *CTRL+Maiusc+K* oppure con la sequenza di menu *Strumenti*, *Sviluppo web*, *Console web*.

Il comando *try* esegue il primo blocco di comandi. Se durante l'esecuzione si verifica un'eccezione viene eseguito il secondo blocco di comandi nel quale il valore della variabile indicata dopo la parola chiave *catch* è un oggetto che contiene informazioni sull'eccezione. La versione estesa del comando *try* prevede un terzo blocco di comandi, che segue la parola chiave *finally*. Questo blocco sarà sempre e comunque eseguito al termine dell'esecuzione del primo blocco. Di norma si utilizza la prima versione del comando *try*.

14.3 Gestione degli eventi

Quando l'utente compie un'azione sulla pagina che sta visualizzando il browser determina l'azione da compiere in risposta, affinché l'utente percepisca l'interazione con la pagina. Ad esempio, quando l'utente clicca il pulsante sinistro del mouse su un particolare punto della pagina accade quanto segue:

- il browser individua il nodo che corrisponde all'elemento puntato dal mouse e verifica se questo nodo può "sentire" l'evento *click*;
- in caso affermativo il browser esegue il codice associato all'evento;
- in caso negativo il browser ignora l'evento.

La gestione di un evento è effettuata da un frammento di codice JavaScript chiamato *gestore di evento*, associato al nodo coinvolto nell'evento. Questo codice è il valore di un attributo il cui nome è determinato convenzionalmente facendo precedere da *on* il nome dell'evento. Ad esempio, l'attributo per l'evento *click* è *onclick*.

Il valore degli attributi relativi ai gestori degli eventi è stabilito usando due tecniche distinte. La prima tecnica, statica, consiste nell'inserimento di un frammento di codice direttamente nel codice relativo a un elemento HTML. La seconda tecnica prevede l'assegnazione statica di un identificatore univoco all'elemento HTML e l'assegnazione dinamica al nodo corrispondente all'elemento HTML di una funzione JavaScript. La prima tecnica è sconsigliata, perché introduce nel documento HTML parti di codice JavaScript che controllano il comportamento interattivo della pagina. Una buona regola da seguire consiste nel separare nettamente i due linguaggi. La seconda tecnica, adottata in questo libro, è più complessa da attuare ma evita l'inconveniente appena citato.

Il gestore di un evento è una funzione che può avere un argomento, il nome dell'evento, che stabilisce cosa si deve fare in reazione del verificarsi dell'evento stesso. All'interno della funzione il nodo associato all'evento è indicato dalla variabile *this*. Il codice di un gestore di evento deve essere protetto da eventuali errori dinamici, mediante il comando *try*.

L'associazione di una funzione che gestisce un evento a un particolare nodo equivale a una *registrazione dell'evento*. Per semplicità, tutte le registrazioni relative a una pagina web sono effettuate in una particolare funzione, di norma chiamata *gestoreLoad*, associata all'evento *load* dell'oggetto *window*. Questo evento accade quando termina il caricamento del documento HTML.

```
window.onload = gestoreLoad;
```

La funzione *gestoreLoad* è, a tutti gli effetti, il gestore dell'evento *load*. Nella prima parte della funzione *gestoreLoad* si identificano tutti i nodi che saranno interessati agli eventi, utilizzando gli identificatori che compaiono nel documento HTML come valore dell'attributo *id* degli elementi interattivi. Ad ogni nodo sarà associato, tramite l'opportuno attributo, il nome del gestore dell'evento che si prenderà cura dell'evento stesso, al momento della sua generazione. Nella seconda parte della funzione si inizializzano tutte le variabili globali dichiarate ma non inizializzate nel codice JavaScript.

14.4 Ciao mondo

L'esempio classico per mostrare il funzionamento di un linguaggio si chiama *Hello world!* (in italiano *Ciao mondo!*). Si tratta di un programma che fa comparire sullo schermo la scritta *Hello world!*, salutando chi lo esegue.

Si può realizzare questo programma mediante una pagina interattiva definita da un documento HTML che contiene un unico pulsante (*Saluta*). Per semplicità di presentazione non si utilizzano fogli di stile CSS.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="CiaoMondo.js"></script>
  <title>Ciao mondo</title>
 </head>
 <body>
  <input type="button"
         id="saluta"
         value="Saluta"/>
 </body>
</html>
```

Il comportamento interattivo della pagina è semplice e intuitivo: quando l'utente pigia il pulsante *Saluta* compare la scritta *Ciao mondo!*

Il programma JavaScript che realizza questo comportamento interattivo è il seguente.

```
function gestoreSaluta () {
   try {
      alert("Ciao mondo!");
   } catch ( e ) {
      alert("gestoreSaluta " + e);
   }
}
var nodoSaluta;
function gestoreLoad () {
   try {
      nodoSaluta = document.getElementById("saluta");
      nodoSaluta.onclick = gestoreSaluta;
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
window.onload = gestoreLoad;
```

Quando il browser carica il file *CiaoMondo.html*, non appena incontra la marca *script* esegue il codice JavaScript associato. Poiché la marca fa riferimento al file *CiaoMondo.js* il browser esegue il programma JavaScript contenuto nel file.

La prima parte del programma contiene la dichiarazione del gestore di eventi *gestoreSaluta* che gestisce gli eventi generati quando l'utente pigia il pulsante *Imposta*.

La seconda parte del programma contiene la dichiarazione delle variabile globale relativa al nodo associato all'elemento interattivo della pagina: *nodoSaluta*.

Nella terza parte del programma si dichiara il gestore di eventi *gestoreLoad* e lo si registra all'evento *load* dell'oggetto *window*, che rappresenta la finestra in cui compare il documento appena caricato.

Al termine del caricamento del file *CiaoMondo.html* viene generato l'evento *load*, gestito da *gestoreLoad*. Questa funzione effettua le seguenti azioni: inizializza la variabile globale relativa al nodo associato all'elemento interattivo della pagina, registra *gestoreSaluta* alla proprietà *onclick* di *nodoSaluta*. Da questo momento in poi il documento diventa interattivo: *gestoreSaluta* è pronta a gestire gli eventi *click* generati dal pulsante *Saluta*.

Quando l'utente pigia il pulsante *Saluta*, il browser genera un evento *click*, gestito da *gestoreSaluta*. La funzione visualizza la stringa *Ciao mondo!* utilizzando il comando *alert("Ciao mondo!")* che apre una finestra in cui compare *Ciao mondo!* Per continuare l'esecuzione del programma l'utente deve pigiare il pulsante *OK* visualizzato nella finestra. Il comando *alert* deve essere usato con

parsimonia e sempre per segnalare all'utente situazioni eccezionali. Le tecniche per comunicare correttamente con l'utente saranno presentate nel seguito.

14.5 Il convertitore di valuta

Un *convertitore di valuta* permette di calcolare il valore di un importo di denaro espresso in una valuta (ad esempio *Euro*) nel corrispondente importo espresso in un'altra valuta (ad esempio *Lire*).

Si può realizzare un convertitore di valuta mediante una pagina interattiva definita da un documento HTML che contiene due pulsanti (*Imposta*, *Converti*), tre campi di inserimento dati (*Valuta*, *Fattore*, *Importo da convertire*) e un campo di sola lettura (*Importo convertito*). Per semplicità di presentazione non si utilizzano fogli di stile CSS.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="ConvertitoreValuta.js"></script>
  <title>Convertitore di valuta</title>
 </head>
 <body>
  <input type="text"
         id="valuta"/> Valuta
  <input type="text"
         id="fattore"/> Fattore di conversione
  <input type="button"
         id="imposta"
         value="Imposta"/>
  <br />
  <input type="text"
         id="importo"/> Importo da convertire
  <input type="button"
         id="converti"
         value="Converti"/>
  <br />
  <input type="text"
         id="risultato"
         readonly="readonly"/> Importo convertito
 </body>
</html>
```

Il comportamento interattivo della pagina è molto semplice e intuitivo. Per definire la valuta e il fattore di conversione l'utente inserisce i relativi valori nei campi *Valuta* e *Fattore* e poi pigia il pulsante *Imposta*. Una volta definita la valuta e il fattore di conversione l'utente converte un importo inserendo il suo va-

lore nel campo *Importo da convertire* e pigiando il pulsante *Converti*. Il valore dell'importo, convertito secondo il fattore di conversione, apparirà nel campo *Importo convertito*.

Il programma JavaScript che realizza questo comportamento interattivo è il seguente.

```
function gestoreImposta () {
   try {
      valutaCorrente = nodoValuta.value;
      fattoreCorrente = nodoFattore.value;
      nodoImporto.value = "";
      nodoRisultato.value = "";
   } catch ( e ) {
      alert("gestoreImposta " + e);
   }
}
function gestoreConverti () {
   try {
      nodoRisultato.value = valutaCorrente + " " +
                            nodoImporto.value * fattoreCorrente;
   } catch ( e ) {
      alert("gestoreConverti " + e);
   }
}
var nodoValuta;
var nodoFattore;
var nodoImposta;
var nodoImporto;
var nodoConverti;
var nodoRisultato;
// stato del convertitore
var valutaCorrente;
var fattoreCorrente;
function gestoreLoad () {
   try {
      nodoValuta = document.getElementById("valuta");
      nodoFattore = document.getElementById("fattore");
      nodoImposta = document.getElementById("imposta");
      nodoImporto = document.getElementById("importo");
      nodoConverti = document.getElementById("converti");
      nodoRisultato = document.getElementById("risultato");
      nodoValuta.value = "";
      nodoFattore.value = "";
      nodoImporto.value = "";
      nodoRisultato.value = "";
      nodoImposta.onclick = gestoreImposta;
```

```
      nodoConverti.onclick = gestoreConverti;
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
window.onload = gestoreLoad;
```

Quando il browser carica il file *ConvertitoreValutaInterattivo.html*, non appena incontra la marca *script* esegue il codice JavaScript associato. Poiché la marca fa riferimento al file *ConvertitoreValutaInterattivo.js* il browser esegue il programma JavaScript contenuto nel file.

La prima parte del programma contiene la dichiarazione di due gestori di eventi: *gestoreImposta* e *gestoreConverti*. Il primo gestisce gli eventi generati quando l'utente pigia il pulsante *Imposta*, il secondo quelli generati quando l'utente pigia il pulsante *Converti*.

Nella seconda parte del programma ci sono le dichiarazioni delle variabili globali relative ai nodi associati agli elementi interattivi della pagina: *nodoValuta*, *nodoFattore*, *nodoImposta*, *nodoImporto*, *nodoConverti*, *nodoRisultato*. Le altre variabili globali, *valutaCorrente* e *fattoreCorrente*, costituiscono lo *stato* della pagina e sono utilizzate dai due gestori di eventi.

Nella terza parte del programma si dichiara il gestore di eventi *gestoreLoad* e lo si registra all'evento *load* dell'oggetto *window*, che rappresenta la finestra in cui compare il documento appena caricato.

Al termine del caricamento del file *ConvertitoreValutaInterattivo.html* viene generato l'evento *load*, gestito da *gestoreLoad*. Questa funzione effettua le seguenti azioni: inizializza le variabili globali relative ai nodi associati agli elementi interattivi della pagina, inizializza i campi di testo della pagina, registra *gestoreImposta* alla proprietà *onclick* di *nodoImposta* e *gestoreConverti* alla proprietà *onclick* di *nodoConverti*. Da questo momento in poi il documento diventa interattivo: *gestoreImposta* e *gestoreConverti* sono pronte a gestire gli eventi *click* generati dai pulsanti *Imposta* e *Converti*.

Quando l'utente pigia il pulsante *Imposta*, il browser genera un evento *click*, gestito da *gestoreImposta*. La funzione assegna il campo *value* di *nodoValuta* alla variabile globale *valutaCorrente* e il campo *value* di *nodoFattore* alla variabile globale *fattoreCorrente*. Poiché la valuta è cambiata, la funzione inizializza i campi *value* di *nodoImporto* e *nodoRisultato*.

Quando l'utente pigia il pulsante *Converti*, il browser genera un evento *click*, gestito dalla funzione *gestoreConverti* associata al nodo relativo al pulsante. La funzione assegna all'attributo *value* di *nodoRisultato* la stringa ottenuta concatenando il valore della variabile *valutaCorrente* al risultato del prodotto dell'atributo *value* di *nodoImporto* con la variabile globale *fattoreCorrente*.

14.6 Validazione dei valori di ingresso

Uno dei problemi da affrontare nella realizzazione di una pagina interattiva è la *validazione* dei *valori di ingresso*, ovvero dei valori inseriti dall'utente negli appositi campi. Nel caso del convertitore di valuta interattivo, i campi di ingresso sono tre: *Valuta*, *Fattore* e *Importo*. L'utente stabilisce quali valori inserire in questi campi prima di pigiare i tasti *Imposta* e *Converti*.

La validazione è effettuata dai gestori degli eventi che accedono in lettura al valore dei campi di ingresso. La validazione consiste in una sequenza di controlli che individuano i valori che non soddisfano i *vincoli* relativi ai campi. Se un valore non soddisfa anche uno solo dei vincoli, il gestore interrompe il calcolo e comunica all'utente che il valore non può essere accettato per portare a termine l'operazione associata all'evento. La comunicazione deve essere chiara ed esplicita, affinché l'utente possa modificare il valore e ripetere la richiesta.

Nel caso del convertitore di valuta il primo campo di ingresso da validare è quello relativo alla valuta (*Valuta*). Il valore che l'utente deve inserire è una qualsiasi stringa di caratteri, purché la sua lunghezza sia maggiore di zero. Tuttavia, per rendere realistica l'interfaccia, si può trasformare questo vincolo richiedendo che la stringa sia lunga esattamente tre caratteri. Una rapida consultazione di un qualsiasi programma per la conversione di valuta disponibile su Internet rivela, infatti, che il codice internazionale per la codifica delle valute prevede una stringa di tre caratteri. Ad esempio, il codice per la lira italiana è *LIT*, per il dollaro statunitense è *USD*, per la valuta giapponese è *YEN*.

Da notare l'uso della *rappresentazione esadecimale \u00E8* per il carattere *è* nel messaggio visualizzato nella finestra di *alert*. Se si fosse utilizzato direttamente tale carattere nella stringa il messaggio sarebbe stato visualizzato erroneamente perché in JavaScript la codifica dei letterali stringa non prevede lìuso di caratteri speciali con segni diacritici.

Il secondo campo da validare è quello relativo al fattore di conversione (*Fattore*). L'utente deve inserire un valore numerico maggiore di zero. La validazione avviene a fasi successive. Per prima cosa si verifica che il campo non sia vuoto, poi si verifica che il valore contenuto nel campo sia numerico e, infine, che il valore numerico sia maggiore di zero. Il primo controllo si basa sulla lunghezza della stringa, che deve essere maggiore di zero. Il secondo controllo prevede la conversione della stringa in un numero mediante la funzione predefinita *Number(x)*, che riceve una stringa *x* e restituisce un valore numerico, *NaN* altrimenti. Il valore *NaN* è riconosciuto nel secondo controllo dal predicato predefinito *isNaN(x)*. Il terzo controllo usa un predicato sul valore numerico.

Il gestore dell'evento associato al pulsante *Imposta* è così modificato.

```
function gestoreImposta () {
   try {
      var valuta = nodoValuta.value;
      if (valuta.length != 3) {
         alert("la valuta non \u00E8 valida");
         return;
      }
      if (nodoFattore.value == "") {
         alert("il fattore di conversione \u00E8 vuoto");
         return;
      }
      var fattore = Number(nodoFattore.value);
      if (isNaN(fattore)) {
         alert(nodoFattore.value + " non \u00E8 un numero");
         return;
      }
      if (fattore <= 0) {
         alert("il fattore di conversione non \u00E8 valido");
         return;
      }
      valutaCorrente = valuta;
      fattoreCorrente = fattore;
      nodoImporto.value = "";
      nodoRisultato.value = "";
   } catch ( e ) {
      alert("gestoreImposta " + e);
   }
}
```

Il terzo campo da validare è quello relativo all'importo da convertire (*Importo*). Come per il fattore di conversione, l'utente deve inserire un valore numerico maggiore di zero. I controlli da effettuare sono analoghi a quelli del campo *Fattore*. Il gestore dell'evento associato al pulsante *Converti* è così modificato.

```
function gestoreConverti () {
   try {
      if (nodoImporto.value == "") {
         alert("l'importo \u00E8 vuoto");
         return;
      }
      var importo = Number(nodoImporto.value);
      if (isNaN(importo)) {
         alert(nodoImporto.value + " non \u00E8 un numero");
         return;
      }
      if (importo <= 0) {
         alert("l'importo non \u00E8 valido");
```

```
         return;
      }
      nodoRisultato.value = valutaCorrente + " " +
                            importo * fattoreCorrente;
   } catch ( e ) {
      alert("gestoreConverti " + e);
   }
}
```

14.7 Dialogo

L'interazione tra l'utente e la pagina segue un *dialogo*, le cui regole stabiliscono la sequenza corretta di azioni che l'utente deve effettuare per ottenere i risultati desiderati. Nel caso del convertitore interattivo, l'unica regola che l'utente deve seguire prevede che prima di convertire un importo è necessario impostare la valuta e il fattore di conversione. Da quel momento in avanti l'utente può convertire uno o più importi, cambiare la valuta e il fattore di conversione e continuare a convertire altri importi. In pratica, l'utente non può effettuare come prima azione dopo il caricamento della pagina la conversione dell'importo.

Per gestire questo semplice dialogo è sufficiente aggiungere un controllo sul valore delle variabili globali *valutaCorrente* e *fattoreCorrente* nella funzione gestoreConverti. Se il valore delle due variabili è *undefined* la conversione non può essere effettuata perché l'utente non ha impostato la valuta e il fattore di conversione.

```
function gestoreConverti () {
   try {
      if ((valutaCorrente == undefined) &&
          (fattoreCorrente == undefined)) {
         alert("valuta e fattore indefiniti");
         return;
      }
      if (nodoImporto.value == "") {
         alert("l'importo \u00E8 vuoto");
         return;
      }
      var importo = Number(nodoImporto.value);
      if (isNaN(importo)) {
         alert(nodoImporto.value + " non \u00E8 un numero");
         return;
      }
      if (importo <= 0) {
         alert("l'importo non \u00E8 valido");
         return;
      }
```

```
      nodoRisultato.value = valutaCorrente + " " +
                            importo * fattoreCorrente;
   } catch ( e ) {
      alert("gestoreConverti " + e);
   }
}
```

14.8 Visualizzazione dei messaggi

L'uso del comando *alert* per comunicare con l'utente deve essere limitato a situazioni eccezionali. In tutti gli altri casi è bene utilizzare altre forme di comunicazione che si limitano a informare l'utente senza richiedere alcuna reazione da parte sua. Il comando *alert*, infatti, richiede che l'utente pigi il pulsante *OK* per chiudere la finestra che contiene la comunicazione.

Tra le tante forme possibili di comunicazione si deve prendere in considerazione la scrittura del messaggio che si vuole presentare all'utente in un'area della finestra. Nel caso del convertitore di valuta i messaggi sono relativi all'impostazione della valuta e alla conversione dell'importo. I primi possono essere visualizzati in un'area subito a destra del pulsante *Imposta*, i secondi subito a destra del pulsante *Converti*. Naturalmente altre soluzioni solo possibili: subito sotto (o sopra) il campo interessato al messaggio, in un'area unica destinata a visualizzare tutti i messaggi e così via.

Per visualizzare i messaggi in due aree della pagina è necessario modificare il codice HTML, introducendo due marche *span* a cui sono associati gli identificatori *messaggioImposta* e *messaggioConverti*.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="ConvertitoreValuta.js"></script>
  <title>Convertitore di valuta</title>
 </head>
 <body>
  <input type="text"
         id="valuta"/> Valuta
  <input type="text"
         id="fattore"/> Fattore di conversione
  <input type="button"
         id="imposta"
         value="Imposta"/>
  <span id="messaggioImposta"></span>
  <br />
  <input type="text"
```

```
         id="importo"/> Importo da convertire
  <input type="button"
         id="converti"
         value="Converti"/>
  <span id="messaggioConverti"></span>
  <br />
  <input type="text"
         id="risultato"
         readonly="readonly"/> Importo convertito
 </body>
</html>
```

Come per gli altri elementi interattivi già inseriti in HTML, anche i nodi associati a *messaggioImposta* e *messaggioConverti* devono essere definiti e inizializzati in *inizializza*. In vista della visualizzazione dei messaggi di errore, è necessario creare un nodo testo e appenderlo come primo figlio dell'elemento *messaggioImposta*. Analogamente va fatto per l'elemento *messaggioConverti*.

La funzione *gestoreLoad* è così modificata.

```
var nodoValuta;
var nodoFattore;
var nodoImposta;
var nodoImporto;
var nodoConverti;
var nodoRisultato;
var nodoMessaggioImposta;
var nodoMessaggioConverti;
// stato del convertitore
var valutaCorrente;
var fattoreCorrente;
function gestoreLoad() {
   try {
      nodoValuta = document.getElementById("valuta");
      nodoFattore = document.getElementById("fattore");
      nodoImposta = document.getElementById("imposta");
      nodoImporto = document.getElementById("importo");
      nodoConverti = document.getElementById("converti");
      nodoRisultato = document.getElementById("risultato");
      nodoMessaggioImposta = document.getElementById("messaggioImposta");
      nodoMessaggioConverti = document.getElementById("messaggioConverti");
      nodoValuta.value = "";
      nodoFattore.value = "";
      nodoImporto.value = "";
      nodoRisultato.value = "";
      nodoImposta.onclick = gestoreImposta;
      nodoConverti.onclick = gestoreConverti;
```

```
      var nodoTesto1 = document.createTextNode("");
      nodoMessaggioImposta.appendChild(nodoTesto1);
      var nodoTesto2 = document.createTextNode("");
      nodoMessaggioConverti.appendChild(nodoTesto2);
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
```

Il gestore di eventi *gestoreImposta* deve essere modificato, sostituendo opportunamente le invocazioni del comando *alert* che comunicano all'utente gli errori relativi ai campi *Valuta* e *Fattore*. Anziché essere usata come argomento del comando *alert*, la stringa che contiene la comunicazione dell'errore deve essere assegnata a un nuovo nodo testo, che diventerà il primo figlio del nodo associato alla marca *span* identificato come *messaggioImposta*. Il gestore usa la funzione di supporto *scriviMessaggio*, con i parametri *nodo* e *messaggio*, che crea un nuovo nodo di testo con la stringa *messaggio* e lo sostituisce al primo figlio di *nodo*.

```
function scriviMessaggio (nodo, messaggio) {
   var nodoTesto = document.createTextNode(messaggio);
   nodo.replaceChild(nodoTesto, nodo.firstChild);
}
function gestoreImposta () {
   try {
      var valuta = nodoValuta.value;
      if (valuta.length != 3) {
         scriviMessaggio(nodoMessaggioImposta,
                                    "la valuta non \u00E8 valida");
         return;
      }
      if (nodoFattore.value == "") {
         scriviMessaggio(nodoMessaggioImposta,
                                  "il campo fattore \u00E8 vuoto");
         return;
      }
      var fattore = Number(nodoFattore.value);
      if (isNaN(fattore)) {
         scriviMessaggio(nodoMessaggioImposta,
                        nodoFattore.value + " non \u00E8 un numero");
         return;
      }
      if (fattore <= 0) {
         scriviMessaggio(nodoMessaggioImposta,
                      "il fattore di conversione non \u00E8 valido");
         return;
```

```
      }
      valutaCorrente = valuta;
      fattoreCorrente = fattore;
      nodoImporto.value = "";
      nodoRisultato.value = "";
   } catch ( e ) {
      alert("gestoreImposta " + e);
   }
}
```

Analogamente si dovrà operare per *gestoreConverti*.

Questa soluzione, però, presenta un inconveniente: i messaggi non sono mai cancellati dall'area in cui compaiono, confondendo l'utente che non sa più se ha correttamente inserito i dati nei campi. Per eliminare l'inconveniente è sufficiente cancellare l'eventuale messaggio da ogni area come prima azione effettuata dai gestori degli eventi. Non è infatti sufficiente che ogni gestore cancelli i messaggi scritti sulla sua area perché l'utente può pigiare i pulsanti in qualsiasi ordine. Il codice modificato, relativo alla funzione *gestoreImposta*, è il seguente. Per semplicità è mostrata solo la parte iniziale della funzione.

```
function gestoreImposta () {
   try {
      scriviMessaggio(nodoMessaggioImposta, "");
      scriviMessaggio(nodoMessaggioConverti, "");
      ...
}
```

14.9 Cookie

Un *cookie* è un'informazione trasmessa da un sito web e memorizzata dal browser mentre l'utente sta visitando una pagina. Ogni volta che l'utente carica la pagina relativa al sito, il browser invia il cookie al server del sito per informarlo dell'attività svolta in precedenza dall'utente. In questo modo, il server tiene traccia del comportamento e delle abitudini dell'utente, raccogliendo informazioni utili che hanno un rilevante valore commerciale [Wikipedia, alla voce *HTTP cookie*].

Ci sono numerose categorie di cookie, ognuna delle quali svolge un ruolo nella navigazione su *Internet*. Per semplicità, il libro presenta solo i cosiddetti *cookie di sessione*, ovvero di quei cookie temporanei che vivono solo durante una *sessione di connessione* a una pagina web e che sono cancellati quando l'utente chiude il browser.

Un cookie di sessione ha la forma *nome* = *valore*, dove *nome* è un identificatore che indica il nome del cookie e *valore* è una stringa che ne rappresenta il

valore. I cookie sono memorizzati in una variabile chiamata *document.cookie*, gestita dal browser. In scrittura è possibile assegnare a questa variabile una stringa che contiene il nome e il valore di un cookie, in lettura si ottiene una stringa che contiene tutti i cookie memorizzati fino a quel momento. La parte più complessa per la gestione dei cookie è l'estrazione del valore di uno specifico cookie dalla stringa memorizzata nella variabile *document.cookie*.

I cookie di sessione scadono (e sono eliminati dalla stringa memorizzata in *document.cookie*) quando l'utente chiude il browser. Per far sì che un cookie di sessione sopravviva alla fine della sessione è necessario indicare una *data di scadenza*. In questo caso, il cookie sarà cancellato solo allo scadere di tale data. La data di scadenza deve essere indicata quando si crea un cookie o quando se ne cambia il valore usando il formato *nome = valore; expires = data*, dove *data* è una stringa che codifica la data nel formato *UTC*[16]. Gli altri aspetti relativi ai cookie (cammino, dominio, sicurezza) non sono affrontati in questo libro.

Il convertitore di valuta può utilizzare i cookie di sessione per memorizzare l'ultima impostazione di valuta effettuata dall'utente. In questo semplice caso, servono due cookie per memorizzare il nome della valuta e il fattore di conversione. Al momento del caricamento della pagina, la funzione inizializza leggerà i cookie e imposterà la valuta e il fattore di conversione utilizzando i valori memorizzati nei cookie.

Prima di procedere alla modifica del convertitore di valuta è bene definire due funzioni che permettono di creare un cookie (*creaCookie*) e una che estrae un cookie (*estraiCookie*).

La funzione *creaCookie* restituisce una stringa nel formato previsto per i cookie. La parte più complessa consiste nella costruzione della stringa relativa alla data di scadenza, se il parametro *scadenza* è definito ed è un intero. Per prima cosa la funzione crea un oggetto *data* del tipo *Date*, inizializzato con la data di creazione. Poi calcola il numero di millisecondi corrispondenti ai giorni di scadenza (*scadenzaMilliSecondi*), estrae da data il numero di millisecondi corrispondenti alla data di creazione[17] (*dataOdierna*), somma a questo valore la *scadenzaMilliSecondi* e assegna il valore ottenuto a *dataFutura*, aggiorna l'oggetto *data* utilizzando il valore di *dataFutura*. Dopo questo laborioso procedimento la funzione crea la stringa nel formato previsto e la restituisce.

```
function creaCookie (nome, valore, scadenza) {
   if (scadenza == undefined || isNaN(scadenza)) {
```

16 UTC (un acronimo che sta per *Coordinated Universal Time*) è il *fuso orario* di riferimento da cui sono calcolati tutti gli altri fusi orari del mondo. Deriva dal *tempo medio di Greenwich* (in inglese *Greenwich Mean Time*, *GMT*), con il quale coincide a meno di approssimazioni infinitesimali [Wikipedia, alla voce UTC].

17 L'oggetto *data* memorizza il numero di millisecondi trascorsi dalle ore 00:00:00 del 1 gennaio 1970. Per convenzione, questa è la data di inizio del tempo.

```
      return nome + "=" + valore;
   }
   var data = new Date();
   var scadenzaMilliSecondi = scadenza * 24 * 60 * 60 * 1000;
   var dataOdierna = data.getTime();
   var dataFutura = dataOdierna + scadenzaMilliSecondi;
   data.setTime(dataFutura);
   return nome + "=" + valore + "; expires=" + data.toGMTString();
}
```

La funzione *estraiCookie* estrae il valore del cookie *nome* da *stringa*, in cui è memorizzato il valore di *document.cookie*. Per prima cosa la funzione crea un array (*coppie*) che contiene tutte le sottostringhe contenute in *stringa* e separate dal punto e virgola (per mezzo del metodo *split* del tipo *String*). Poi la funzione scandisce l'array *coppie* cercando la stringa che inizia con la sottostringa *nome*. Se non la trova restituisce *undefined*, altrimenti restituisce la seconda parte della coppia, ovvero il valore del cookie. La funzione usa il metodo *trim*, il metodo *substr* e il metodo *indexOf* del tipo *String*.

```
function estraiCookie (nome, stringa) {
   var coppie = stringa.split(";");
   var i = 0;
   while (i < coppie.length &&
          (coppie[i].trim().indexOf(nome) != 0)) {
      i++;
   }
   if (i < coppie.length) {
      var l = nome.length;
      var s = coppie[i].trim();
      return s.substr(l + 1, s.length - l);
   } else {
      return undefined;
   }
}
```

La prima modifica da apportare è la scrittura dei cookie. Poiché la valuta è impostata da *gestoreImposta* sarà proprio questa funzione a creare e scrivere i cookie i cui nomi saranno proprio *valuta* e *fattore*. Per fare questo si utilizza la funzione *creaCookie*, a cui sono passati due argomenti: il nome del cookie e il suo valore.

```
   valutaCorrente = valuta;
   fattoreCorrente = fattore;
   document.cookie = creaCookie("valuta", valutaCorrente);
   document.cookie = creaCookie("fattore", fattoreCorrente);
```

La seconda modifica è relativa alla lettura dei cookie e usare i loro valori per inizializzare le variabili globali *valutaCorrente* e *fattoreCorrente*, oltre che al valore dei campi *valuta* e *fattore*. Tuttavia, poiché i cookie potrebbero non essere stati creati in precedenza, è necessario verificare che il valore restituito dalla funzione *estraiCookie* non sia *undefined*. La modifica apportata alla funzione *gestoreLoad* è la seguente.

```
   var valuta = estraiCookie("valuta", document.cookie);
   if (valuta != undefined) {
      nodoValuta.value = valuta;
      valutaCorrente = valuta;
   }
   var fattore = estraiCookie("fattore", document.cookie);
   if (fattore != undefined) {
      nodoFattore.value = fattore;
      fattoreCorrente = fattore;
   }
```

Il cookie utilizzato per il convertitore è di sessione. Ciò significa che chiudendo il browser il cookie è cancellato e, quindi, la valuta e il fattore di conversione mostrati all'utente saranno quelli che compaiono per prima nell'array *valuteFattore* e non quelli impostati l'ultima volta dall'utente.

14.10 Menu di selezione

La possibilità di impostare la valuta e il fattore di conversione è un aspetto dell'interfaccia utente della pagina web che realizza un convertitore di valuta che può essere migliorato, semplificando il dialogo con l'utente. Nell'ultima versione della pagina web l'utente può scegliere liberamente il codice della valuta e il fattore di conversione. Nella pratica, però, i codici delle valute sono definiti da uno standard e i fattori di conversione sono stabiliti in tempo reale dalle banche centrali. Sarebbe opportuno, quindi, presentare all'utente i codici possibili e recuperare i fattori di conversione mediante un collegamento con un opportuno *servizio web*. Il primo aspetto può essere agevolmente affrontato modificando il documento HTML e il programma JavaScript, il secondo richiede un impegno e competenze tecniche che esulano dagli obiettivi di questo libro.

I codici delle valute e i relativi fattori di conversione si possono rappresentare mediante una variabile globale *valuteFattori*, il cui valore è un array associativo che contiene i nomi delle valute e i corrispondenti fattori di conversione.

```
var valuteFattori = {
                     LIT : 1937.26,
                     USD : 0.95,
                     YEN : 0.13
                    }
```

Per evitare che l'utente debba inserire il nome della valuta in un campo di testo si utilizza un *menu di selezione* in cui compaiono i nomi delle valute conosciute dal convertitore: *LIT* (Lira italiana), *USD* (Dollaro statunitense), *YEN* (Yen giapponese). La modifica del codice HTML riguarda solo il campo *Valuta*.

```
   <select id="valuta">
    <option>LIT</option>
    <option>USD</option>
    <option>YEN</option>
   </select> Valuta
```

L'uso di un menu di selezione modifica sostanzialmente il dialogo con l'utente. Al momento dell'inizializzazione, infatti, il menu presenta la prima opzione. L'utente può iniziare a convertire un importo senza dover impostare la valuta. Questo cambiamento elimina la necessità di controllare l'inizializzazione delle variabili *valutaCorrente* e *fattoreCorrente* nella funzione *gestoreConverti*. Ovviamente, non essendo più necessario, si deve eliminare il pulsante *Imposta*.

L'altra modifica sostanziale riguarda l'impossibilità dell'utente di commettere errori nell'impostazione della valuta. Il nome della valuta sarà scelto tra quelli mostrati nel menu e il fattore di conversione sarà visualizzato dall'interfaccia, senza che l'utente debba inserirlo. Pertanto, non sarà più necessario effettuare i controlli sul campo *Fattore* e visualizzare gli eventuali messaggi in *messaggioImposta*.

La funzione *gestoreImposta* è così modificata, per tener conto delle osservazioni appena riportate.

```
function gestoreImposta () {
   try {
      scriviMessaggio(nodoMessaggioConverti, "");
      var valuta = nodoValuta.value;
      var fattore = valuteFattori[valuta];
      nodoFattore.value = fattore;
      valutaCorrente = valuta;
      fattoreCorrente = fattore;
      document.cookie = creaCookie("valuta", valutaCorrente);
      document.cookie = creaCookie("fattore", fattoreCorrente);
      nodoImporto.value = "";
      nodoRisultato.value = "";
   } catch ( e ) {
      alert("gestoreImposta: " + e);
   }
}
```

La funzione *gestoreConverti* è quasi invariata. Deve essere solo eliminato il riferimento al campo *messaggioImposta*.

La funzione *gestoreLoad* richiede qualche modifica sostanziale. Per prima cosa è necessario eliminare il collegamento tra il nodo del pulsante *Imposta* (eliminato perché inutile) e il gestore degli eventi *gestoreImposta*. Al suo posto dovrà essere creato un collegamento tra il nodo del menu di selezione *Valuta* e *gestoreImposta*. Questa volta l'evento da gestire sarà *change* e l'attributo da usare sarà *onchange*. Infine, per impostare il convertitore al momento della sua attivazione si invoca la funzione *gestoreImposta*.

```
function gestoreLoad () {
   try {
      nodoValuta = document.getElementById("valuta");
      nodoFattore = document.getElementById("fattore");
      nodoImporto = document.getElementById("importo");
      nodoConverti = document.getElementById("converti");
      nodoRisultato = document.getElementById("risultato");
      nodoMessaggioConverti = document.getElementById("messaggioConverti");
      nodoImporto.value = "";
      nodoRisultato.value = "";
      nodoValuta.onchange = gestoreImposta;
      nodoConverti.onclick = gestoreConverti;
      var valuta = estraiCookie("valuta", document.cookie);
      if (valuta != undefined) {
         nodoValuta.value = valuta;
         valutaCorrente = valuta;
      }
      var fattore = estraiCookie("fattore", document.cookie);
      if (fattore != undefined) {
         nodoFattore.value = fattore;
         fattoreCorrente = fattore;
      }
      var nodoTesto1 = document.createTextNode("");
      nodoMessaggioConverti.appendChild(nodoTesto1);
      gestoreImposta();
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
```

14.11 Generazione dinamica del menu di selezione

La soluzione appena presentata ha un difetto strutturale: i contenuti della pagina, ovvero le valute, sono elencate sia in HTML che in JavaScript. Questa duplicazione può essere fonte di errori e imprecisioni in caso di modifica della lista. Per eliminare questo difetto si deve evitare la duplicazione, modificando il menu delle valute presente nel codice HTML.

```
<select id="valuta"></select> Valuta
```

Togliere la lista delle valute dal codice HTML significa avere un campo *select* in cui non compaiono *staticamente* le relative opzioni. Le opzioni dovranno essere create *dinamicamente* al momento del caricamento della pagina da parte del programma JavaScript. Per fare ciò è necessario definire una funzione chiamata *creaSelect* che modifica l'albero DOM della pagina, aggiungendo tanti elementi *option* quanti sono gli elementi di un array associativo. La funzione sarà invocata nel corpo di *gestoreLoad* con i seguenti argomenti: *nodoValuta* e *valuteFattori*.

La funzione *creaSelect* è così definita.

```
function creaSelect (nodoSelect, opzioni) {
   for (var opzione in opzioni) {
      var nodoOpzione = document.createElement("option");
      var nodoTesto = document.createTextNode(opzione);
      nodoOpzione.appendChild(nodoTesto);
      nodoSelect.appendChild(nodoOpzione);
   }
}
```

Il resto del programma JavaScript è invariato.

14.12 Esercizi

1. Definire una pagina web interattiva che realizza una sveglia. La pagina permette di impostare l'orario corrente, l'allarme e di far avanzare l'ora corrente. L'inserimento dell'orario corrente avviene mediante un pulsante e due campi di testo che contengono l'ora e i minuti. L'inserimento dell'orario dell'allarme avviene mediante un pulsante e due campi di testo che contengono l'ora e i minuti. L'ora corrente avanza di un minuto mediante un pulsante. La pagina visualizza in due campi di testo di sola lettura l'orario corrente e l'orario della sveglia. Quando l'orario corrente diventa uguale a quello dell'allarme la pagina visualizza un apposito messaggio.
2. Definire una pagina web interattiva che realizza un distributore automatico di caffè. La pagina permette di aggiungere al distributore un numero arbitrario di capsule, di chiedere l'erogazione di un numero arbitrario di caffè (tenendo conto delle capsule disponibili) e di addebitarle a una persona identificata da un codice univoco, di visualizzare il numero di caffè addebitati finora a una persona mediante il suo codice univoco. Al momento della sua attivazione la pagina non ha conoscenza dei codici delle

persone che utilizzeranno il distributore. L'aggiunta delle capsule avviene mediante un pulsante e un campo di testo che contiene il numero di capsule da aggiungere. L'erogazione dei caffè avviene mediante un pulsante e due campi di testo che contengono il numero di caffè da erogare e il codice a cui questi caffè saranno addebitati. La visualizzazione degli addebiti avviene mediante un pulsante e un campo di testo che contiene il codice di cui si chiedono informazioni.

3. Definire una pagina web interattiva che realizza un catalogo formato da una sequenza di foto a ognuna delle quali è associato un testo esplicativo. Quando l'utente clicca su un'immagine il testo ad essa associato compare nella parte della pagina sottostante alle immagini, sostituendo il testo precedentemente visualizzato.

15 Gestione dei contenuti

Una pagina web visualizza informazioni di varia natura: testuale, grafica, audiovisiva. La gestione di queste informazioni, chiamate anche *contenuti*, diventa sempre più complessa all'aumentare della loro dimensione e varietà. Per affrontare questo problema è necessario ricorrere a tecniche di rappresentazione e di trattamento dei contenuti.

15.1 Il ricettario

In molti casi, i contenuti gestiti da una pagina non possono essere rappresentati da un unico array, soluzione adottata per il convertitore di valuta. Una situazione molto frequente è quella in cui i contenuti sono formati da una lista di *entità*, ognuna delle quali è caratterizzata da proprietà comuni. La rappresentazione di queste entità si può basare su un array in cui ogni elemento è a sua volta un array associativo. Questo approccio, molto generale, può essere seguito in moltissime situazioni.

Si consideri, ad esempio, una pagina interattiva che visualizza un *ricettario* formato da alcune ricette, permettendo all'utente di effettuare alcune semplici attività di ricerca. Il ricettario può essere rappresentato da un array di elementi, uno per ogni ricetta. A sua volta, una ricetta può essere rappresentata mediante un array associativo di quattro elementi: *categoria* (che può assumere i valori *antipasto*, *primo, secondo, contorno, frutta o dolce*), *nome*, *difficoltà* (*bassa, media* o *alta)* e *preparazione* (il numero di minuti previsto).

Il contenuto specifico di una pagina potrebbe essere costituito da due ricette, rappresentate dal seguente array, dove *ricettario* è una variabile globale.

```
var ricettario = [
   {
      categoria    : "Primo",
      nome         : "Gnocchi",
      difficoltà   : "media",
      preparazione : 30
   },
   {
      categoria    : "Secondo",
      nome         : "Cotoletta",
```

```
      difficoltà   : "bassa",
      preparazione : 15
   }
]
```

Il documento HTML che definisce la pagina interattiva del ricettario contiene un elemento *ol* (la lista puntata che conterrà le ricette) il cui identificatore è *ricette*.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="Ricettario.js"></script>
  <title>Ricettario</title>
 </head>
 <body>
  <b>Le mie ricette</b>
  <br />
  <ol id="ricette"></ol>
 </body>
</html>
```

Il file *Ricettario.js* contiene un programma JavaScript che dichiara la variabile *ricettario* e che associa all'evento *load* il gestore di evento *gestoreLoad*. Il gestore inizializza la variabile globale *nodoRicette* e invoca la funzione *calcolaListaDescrizioni* che costruisce un array di stringhe, ognuna delle quali descrive una ricetta. La funzione cre*aLista* ha due parametri, *nodoRicette* e *listaRicette*, che utilizza per modificare l'albero DOM della pagina web.

```
var nodoRicette;
function gestoreLoad () {
   try {
      nodoRicette = document.getElementById("ricette");
      var listaDescrizioni = calcolaListaDescrizioni(ricettario);
      creaLista(nodoRicette, listaDescrizioni);
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
window.onload = gestoreLoad;
```

La funzione *calcolaListaDescrizioni* scandisce l'array *ricette* e per ogni elemento, che rappresenta una ricetta, costruisce una stringa che la descrive. Le descrizioni sono aggiunte a un array, *listaDescrizioni*, che viene restituito come risultato dalla funzione.

```
function calcolaListaDescrizioni(ricette) {
   var listaDescrizioni = [];
   for (var i = 0; i < ricette.length; i++) {
      var ricetta = ricette[i];
      var s = "(" + ricetta.categoria + ") " +
              ricetta.nome + ", difficoltà " +
              ricetta.difficoltà + ", " +
              ricetta.preparazione + " minuti di preparazione";
      listaDescrizioni.push(s);
   }
   return listaDescrizioni;
}
```

La funzione *creaLista* scandisce un array di stringhe e, per ognuna di esse, aggiunge un nuovo elemento al nodo DOM (il primo parametro) della pagina web.

```
function creaLista (nodoLista, elementi) {
   for (var i = 0; i < elementi.length; i++) {
      var elemento = elementi[i];
      var nodoElemento = document.createElement("li");
      nodoLista.appendChild(nodoElemento);
      var nodoTesto = document.createTextNode(elemento);
      nodoElemento.appendChild(nodoTesto);
   }
}
```

15.2 Ricerca esatta

Molto spesso in una pagina web è necessario effettuare delle *ricerche* sui suoi contenuti. La ricerca consiste nell'impostazione, da parte dell'utente, di una *chiave* (o delle chiavi) di ricerca seguita dalla ricerca vera e propria delle informazioni che corrispondono alla chiave (o alle chiavi) e alla visualizzazione del risultato della ricerca. I *motori di ricerca* attualmente disponibili sul web utilizzano questo schema generale.

La prima tecnica di ricerca, chiamata *ricerca esatta*, consiste nell'indicare una chiave, una stringa, e cercare esattamente questa stringa nei contenuti della pagina. Ad esempio, la pagina del ricettario potrebbe avere una funzione di ricerca che permette all'utente di cercare una ricetta con il suo nome. Se la ricetta è presente, il risultato della ricerca sarà la visualizzazione delle informazioni relative alla ricetta trovata, altrimenti si comunicherà all'utente che non esistono ricette con quel nome.

Il documento HTML relativo a questa tecnica di ricerca è il seguente.

```
<!DOCTYPE html>
<html>
```

```
 <head>
  <meta charset="UTF-8">
  <script src="Ricettario.js"></script>
  <title>Ricettario</title>
 </head>
 <body>
  <b>Le mie ricette</b>
  <br />
  <input type="text" id="chiave"/>
  <input type="button" value="Cerca" id="cerca"/>
  <br />
  <br />
  <div id="risultato"></div>
 </body>
</html>
```

Il documento definisce un campo testuale (*chiave*) e un pulsante (*cerca*). L'utente inserisce la chiave nel campo testuale e pigia il pulsante. Il risultato della ricerca è mostrato nell'elemento *risultato*.

Il codice JavaScript che realizza la ricerca esatta è così strutturato. La parte di inizializzazione è simile a quella già vista in precedenza, con l'aggiunta delle variabili globali *nodoChiave* e *nodoCerca*. Per gestire il pulsante *cerca* si definisce la funzione *gestoreCerca*, che gestisce l'evento *onclick*.

```
var nodoChiave;
var nodoCerca;
var nodoRisultato;
function gestoreLoad () {
   try {
      nodoChiave = document.getElementById("chiave");
      nodoCerca = document.getElementById("cerca");
      nodoRisultato = document.getElementById("risultato");
      nodoChiave.value = "";
      nodoCerca.onclick = gestoreCerca;
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
```

La funzione *gestoreCerca* estrae il valore di *nodoChiave* (la stringa di ricerca), invoca la funzione *ricercaEsatta* e poi visualizza l'eventuale ricetta trovata. La visualizzazione riusa le funzioni *calcolaListaDescrizioni* e *creaLista*, leggermente modificata. Se la ricerca ha esito negativo la funzione visualizza un messaggio all'utente.

```
function gestoreCerca () {
   try {
      var chiave = nodoChiave.value;
      var ricette = ricercaEsatta(chiave);
      var listaDescrizioni;
      if (ricette.length != 0) {
         listaDescrizioni = calcolaListaDescrizioni(ricette);
      } else {
         listaDescrizioni = ["Nessuna ricetta trovata"];
      }
      creaLista(nodoRisultato, listaDescrizioni);
   } catch ( e ) {
      alert("gestoreCerca " + e);
   }
}
```

La funzione *ricercaEsatta* scandisce le ricette cercando quella la cui proprietà *nome* è esattamente uguale alla stringa *chiave*.

```
function ricercaEsatta (chiave) {
   var listaRicette = [];
   var i = 0;
   while (i < ricettario.length && ricettario[i].nome != chiave) {
      i++;
   }
   if (i < ricettario.length) {
      listaRicette.push(ricettario[i]);
   }
   return listaRicette;
}
```

La versione modificata di *creaLista* utilizza la funzione *rimuoviFigli* che elimina tutti i nodi figli di un dato nodo.

```
function creaLista (nodoLista, elementi) {
   rimuoviFigli(nodoLista);
   for (var i = 0; i < elementi.length; i++) {
      var elemento = elementi[i];
      var nodoElemento = document.createElement("li");
      nodoLista.appendChild(nodoElemento);
      var nodoTesto = document.createTextNode(elemento);
      nodoElemento.appendChild(nodoTesto);
   }
}
function rimuoviFigli (nodo) {
   while (nodo.childNodes.length > 0) {
      nodo.removeChild(nodo.firstChild);
```

```
    }
}
```

15.3 Ricerca con menu di selezione

Lo svantaggio della ricerca esatta consiste nel fatto che l'utente deve conoscere esattamente la chiave di ricerca per ottenere il risultato voluto. A volte, però, l'utente non conosce il valore della chiave di ricerca ma vuole scegliere tra un insieme di valori possibili. In questo caso la ricerca avviene mediante un menu di selezione che presenta le opzioni possibili. Nel caso del ricettario si potrebbe presentare un *menu* che indica tutte le categorie con cui sono catalogate le ricette e visualizzare tutte le ricette che appartengono alla categoria selezionata dall'utente. Se alla categoria selezionata non appartiene alcuna ricetta, la visualizzazione del risultato indica il fallimento della ricerca.

Il documento relativo a questa tecnica di ricerca è il seguente.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="Ricettario.js"></script>
  <title>Ricettario</title>
 </head>
 <body>
  <b>Le mie ricette</b>
  <br />
  <select id="categorie">
   <option value="Antipasto">Antipasto</option>
   <option value="Primo">Primo</option>
   <option value="Secondo">Secondo</option>
   <option value="Contorno">Contorno</option>
   <option value="Frutta">Frutta</option>
   <option value="Dolce">Dolce</option>
  </select>
  <input type="button" value="Cerca" id="cerca"/>
  <br />
  <div id="risultato"></div>
 </body>
</html>
```

Il documento definisce un menu (c*ategorie*) con sei opzioni, corrispondenti ai possibili valori della proprietà *categoria*. Il pulsante per la ricerca (*cerca*) resta invariato, così come l'elemento (*risultato*) utilizzato per visualizzare l'esito della ricerca.

Il codice JavaScript che realizza la ricerca mediante un menu definito staticamente è molto simile a quello già visto in precedenza. Per prima cosa è necessario aggiungere una variabile globale *nodoCategorie* per il nodo *categorie*, modificando la funzione *gestoreLoad*.

```
var nodoCategorie;
var nodoCerca;
var nodoRisultato;
function gestoreLoad () {
   try {
      nodoCategorie = document.getElementById("categorie");
      nodoCerca = document.getElementById("cerca");
      nodoRisultato = document.getElementById("risultato");
      nodoCerca.onclick = gestoreCerca;
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
```

La funzione *gestoreCerca* recupera la categoria selezionata dall'utente e invoca *ricercaCategoria* ottenendo, come risultato, tutte le ricette che appartengono a quella categoria. La visualizzazione delle eventuali ricette è inalterata.

```
function gestoreCerca () {
   try {
      var categoria = nodoCategorie.value;
      var ricette = ricercaCategoria(categoria);
      var listaDescrizioni;
      if (ricette.length != 0) {
         listaDescrizioni = calcolaListaDescrizioni(ricette);
      } else {
         listaDescrizioni = ["Nessuna ricetta trovata"];
      }
      creaLista(nodoRisultato, listaDescrizioni);
   } catch ( e ) {
      alert("gestoreCerca " + e);
   }
}
```

La ricerca vera e propria è effettuata dalla funzione *ricercaCategoria*. La funzione scandisce tutte le ricette del ricettario e costruisce un array con quelle la cui proprietà *categoria* è uguale al parametro *categoria*.

```
function ricercaCategoria (categoria) {
   var ricette = [];
   for (var i = 0; i < ricettario.length; i++) {
      var ricetta = ricettario[i];
```

```
      if (ricetta.categoria == categoria) {
         ricette.push(ricetta);
      }
   }
   return ricette;
}
```

15.4 Ricerca con menu di selezione generato dinamicamente

Lo svantaggio dell'uso di un menu definito staticamente consiste nel fatto che l'utente può selezionare una categoria per la quale non sono presenti ricette. Per evitare questo inconveniente è possibile creare dinamicamente un menu in cui compaiono solo le categorie alle quali sono associate delle ricette. In questo modo la ricerca non avrà mai esito negativo.

Il documento relativo a questa tecnica di ricerca è il seguente.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="Ricettario.js"></script>
  <title>Ricettario</title>
 </head>
 <body>
  <b>Le mie ricette</b>
  <br />
  <select id="categorie"></select>
  <input type="button" value="Cerca" id="cerca"/>
  <br />
  <div id="risultato"></div>
 </body>
</html>
```

Il documento definisce un menu (*categoria*) che non ha opzioni. Le opzioni sono calcolate dopo aver determinato le categorie effettivamente presenti nelle ricette.

Il codice JavaScript che realizza la ricerca mediante un menu definito dinamicamente è così strutturato. La parte di inizializzazione è leggermente modificata rispetto a quella già vista in precedenza. La modifica riguarda l'aggiunta della creazione del menu, effettuata dalla funzione *creaSelect* che ha come argomenti *nodoCategorie* e *categorie*, un array associativo che contiene le categorie restituito dalla funzione *calcolaCategorie*. Il gestore *gestoreCerca* è simile a quello già visto in precedenza. La gestione della lista vuota delle ricette è stata eliminata perché, per costruzione, questo caso non si può presentare.

```
function gestoreLoad () {
   try {
      nodoCategorie = document.getElementById("categorie");
      nodoCerca = document.getElementById("cerca");
      nodoRisultato = document.getElementById("risultato");
      var categorie = calcolaCategorie();
      creaSelect(nodoCategorie, categorie);
      nodoCerca.onclick = gestoreCerca;
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
```

La funzione *calcolaCategorie* scandisce tutte le ricette e costruisce un array associativo (*categorie*) che contiene tutte le categorie presenti nelle ricette.

```
function calcolaCategorie () {
   var categorie = {};
   for (var i = 0; i < ricettario.length; i++) {
      var ricetta = ricettario[i];
      categorie[ricetta.categoria] = true;
   }
   return categorie;
}
```

La funzione *creaSelect* è una funzione che, utilizzando l'array associativo *opzioni*, crea un menu e lo aggiunge a *nodoSelect*.

```
function creaSelect (nodoSelect, opzioni) {
   rimuoviFigli(nodoSelect);
   for (var opzione in opzioni) {
      var nodoOpzione = document.createElement("option");
      nodoOpzione.value = opzione;
      var nodoTesto = document.createTextNode(opzione);
      nodoOpzione.appendChild(nodoTesto);
      nodoSelect.appendChild(nodoOpzione);
   }
}
```

15.5 Ricerca con chiavi multiple

Le ricerche definite finora utilizzano solo una proprietà delle entità che appartengono al contenuto della pagina. Sarebbe utile effettuare una ricerca in base a più di una proprietà, in modo da rendere più precisa la richiesta dell'utente. In questo caso si tratta di una *ricerca a chiave multipla*, in cui gli elementi cercati devono soddisfare una molteplicità di criteri.

Le ricerche a chiave multipla si suddividono in ricerche in cui gli elementi cercati devono soddisfare i criteri di tutte le chiavi (*ricerca a chiave multipla congiuntiva*) o in cui gli elementi devono soddisfarne almeno uno (*ricerca a chiave multipla disgiuntiva*). Tra questi due estremi ci sono le ricerche per le quali alcune chiavi devono essere presenti e altre possono mancare.

Nel caso delle ricette, oltre alla categoria sarebbe utile scegliere anche il tempo di preparazione, indicato per ogni ricetta dalla proprietà *preparazione*. Nel seguito sarà presentata una ricerca a chiave multipla congiuntiva che usa due chiavi: *categoria* e *preparazione*.

```
 <body>
  <b>Le mie ricette</b>
  <br />
  <select id="categorie"></select>
  <select id="preparazioni"></select>
  <input type="button" value="Cerca" id="cerca"/>
  <br />
  <div id="risultato"></div>
 </body>
```

Il documento definisce un nuovo menu (*preparazioni*) le cui opzioni sono calcolate dopo aver determinato i tempi di preparazione presenti nelle ricette.

Il codice JavaScript che realizza la ricerca mediante due menu è simile a quello visto in precedenza. La parte di inizializzazione è leggermente modificata, con l'aggiunta della creazione del menu relativo alla preparazione.

```
var nodoCategorie;
var nodoPreparazioni;
var nodoCerca;
var nodoRisultato;
function gestoreLoad () {
   try {
      nodoCategorie = document.getElementById("categorie");
      nodoPreparazioni = document.getElementById("preparazioni");
      nodoCerca = document.getElementById("cerca");
      nodoRisultato = document.getElementById("risultato");
      var categorie = calcolaCategorie();
      creaSelect(nodoCategorie, categorie);
      var preparazioni = calcolaPreparazioni();
      creaSelect(nodoPreparazioni, preparazioni);
      nodoCerca.onclick = gestoreCerca;
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
```

La funzione *calcolaPreparazioni* scandisce le ricette e costruisce un array associativo (*preparazioni*) che contiene tutte le categorie presenti nelle ricette.

```
function calcolaPreparazioni () {
   var preparazioni = {};
   for (var i = 0; i < ricettario.length; i++) {
      var ricetta = ricettario[i];
      preparazioni[ricetta.preparazione] = true;
   }
   return preparazioni;
}
```

La ricerca usa il valore delle due chiavi, selezionate dai due menu creati in fase di inizializzazione. Dopo aver estratto questi valori il gestore del pulsante *cerca* (*gestoreCerca*) invoca il metodo *ricercaMultipla* ottenendo un array, possibilmente vuoto, che contene le ricette che soddisfano la ricerca a chiave multipla.

```
function gestoreCerca () {
   try {
      var categoria = nodoCategorie.value;
      var preparazione = Number(nodoPreparazioni.value);
      var ricette = ricercaMultipla(categoria, preparazione);
      var listaDescrizioni;
      if (ricette.length != 0) {
         listaDescrizioni = calcolaListaDescrizioni(ricette);
      } else {
         listaDescrizioni = ["Nessuna ricetta trovata"];
      }
      creaLista(nodoRisultato, listaDescrizioni);
   } catch ( e ) {
      alert ("gestoreCerca " + e);
   }
}
```

La funzione *ricercaMultipla* scandisce tutte le ricette selezionando quelle i cui attributi *categoria* e *preparazione* sono uguali ai valori dei parametri.

```
function ricercaMultipla (categoria, preparazione) {
   var ricette = [];
   for (var i = 0; i < ricettario.length; i++) {
      var ricetta = ricettario[i];
      if (ricetta.categoria == categoria &&
          ricetta.preparazione == preparazione) {
         ricette.push(ricetta]);
      }
   }
```

```
   return ricette;
}
```

15.6 Ricerca senza pulsante

Avendo a disposizione uno o più menu di selezione per specificare i criteri di scelta è possibile eliminare il pulsante e rendere l'interazione con l'utente più semplice e diretta. Le modifiche da fare al ricettario, relativamente all'ultima versione presentata, sono molto limitate: nel documento HTML deve essere eliminato il pulsante *Cerca*, nel codice JavaScript deve essere modificata la funzione *gestoreLoad*. La parte *body* del documento HTML è il seguente.

```
 <body>
  <b>Le mie ricette</b>
  <br />
  <select id="categorie"></select>
  <select id="preparazioni"></select>
  <br />
  <div id="risultato"></div>
 </body>
```

Nella funzione *gestoreLoad* si eliminano i riferimenti a *nodoCerca* e si registra l'evento *onchange* ai menu *categorie* e *preparazioni*. Infine, per visualizzare il risultato relativo al valore iniziale dei due menu si invoca *gestoreCerca*.

```
var nodoCategorie;
var nodoPreparazioni;
var nodoRisultato;
function gestoreLoad () {
   try {
      nodoCategorie = document.getElementById("categorie");
      nodoPreparazioni = document.getElementById("preparazioni");
      nodoRisultato = document.getElementById("risultato");
      var categorie = calcolaCategorie();
      creaSelect(nodoCategorie, categorie);
      var preparazioni = calcolaPreparazioni();
      creaSelect(nodoPreparazioni, preparazioni);
      nodoCategorie.onchange = gestoreCerca;
      nodoPreparazioni.onchange = gestoreCerca;
      gestoreCerca();
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
```

15.7 Selezione a cascata

La soluzione appena presentata ha un inconveniente: al momento del caricamento della pagina viene visualizzato il risultato della ricerca effettuata utilizzando il valore delle due chiavi relative alla categoria e alla preparazione. Nel caso particolare, questi valori sono *Primo* e *15*, a cui non corrisponde alcuna ricetta del ricettario. L'utente è disorientato perché, pur non avendo effettuato alcuna ricerca ne vede visualizzato il risultato.

Per ovviare a questo errore di interazione si possono usare numerose tecniche. Una di queste, chiamata *selezione a cascata*, prevede che l'utente selezioni il valore della prima chiave (la categoria) e successivamente quello della seconda chiave. Solo dopo la selezione della seconda chiave la ricerca viene effettuata. Uno dei vantaggi di questa tecnica è la possibilità di escludere combinazioni di chiavi che danno luogo a una ricerca senza esito.

La prima modifica da apportare è la sostituzione di *gestoreCerca* con due nuove funzioni, *gestoreCategoria* e *gestorePreparazione*, che gestiscono l'evento *onchange* relativo ai menu *categorie* e *preparazioni*. La gestione di questi eventi deve tener conto dell'interazione basata su due menu a cascata tra di loro.

La funzione *gestoreCategoria* è organizzata in due parti. La prima gestisce il menu delle categorie, la seconda quello delle preparazioni. L'uso di due menu a cascata pone un problema relativo alle opzioni del menu delle categorie. Al momento dell'inizializzazione della pagina, il menu visualizza come opzione di scelta la voce fittizia "--- *categoria* ---" che sta a indicare all'utente il fatto che deve, appunto scegliere una categoria. Una volta effettuata la scelta questa voce fittizia deve scomparire dal menu. La prima parte della funzione gestisce questo comportamento. La seconda parte si occupa di filtrare tra tutte le ricette del ricettario quelle che hanno come categoria quella scelta dall'utente e di creare un menu le cui voci corrispondono ai tempi di preparazione di queste ricette. Anche in questo caso il menu deve visualizzare come opzione di scelta la voce fittizia "--- *preparazione* ---", dando all'utente la possibilità di sceglierne una.

```
function gestoreCategoria () {
   try {
      var categoria = nodoCategorie.value;
      var categorie = calcolaCategorie();
      creaSelect(nodoCategorie, categorie);
      nodoCategorie.value = categoria;
      var preparazioni = ricercaPreparazioni(categoria);
      creaSelect(nodoPreparazioni, preparazioni);
      aggiungiPrimaOpzione(nodoPreparazioni, "--- preparazione ---");
   } catch ( e ) {
      alert ("gestoreCategoria " + e);
```

```
    }
}
```

La funzione *gestoreCategoria* invoca due funzioni non definite in precedenza: *ricercaPreparazioni* e *aggiungiPrimaOpzione*. La prima effettua la scansione delle ricette e crea un array associativo che contiene i tempi di preparazione di tutte quelle che appartengono alla categoria selezionata dall'utente. La seconda aggiunge in prima posizione un'opzione a un menu già esistente.

```
function ricercaPreparazioni (categoria) {
   var preparazioni = {};
   for (var i = 0; i < ricettario.length; i++) {
      var ricetta = ricettario[i];
      if (ricetta.categoria == categoria) {
         preparazioni[ricetta.preparazione] = true;
      }
   }
   return preparazioni;
}
function aggiungiPrimaOpzione (nodoSelect, opzione) {
   var nodoOpzione = document.createElement("option");
   nodoOpzione.value = opzione;
   var nodoTesto = document.createTextNode(opzione);
   nodoOpzione.appendChild(nodoTesto);
   nodoSelect.insertBefore(nodoOpzione, nodoSelect.firstChild);
   nodoSelect.value = opzione;
}
```

La funzione *gestorePreparazioni* è organizzata in due parti: la prima individua le voci dei due menu relative alla scelta dell'utente: categorie e preparazione. La seconda cerca le ricette che soddisfano i due criteri di scelta e le visualizza.

```
function gestorePreparazione () {
   try {
      var categoria = nodoCategorie.value;
      var preparazione = Number(nodoPreparazioni.value);
      var preparazioni = ricercaPreparazioni(categoria);
      creaSelect(nodoPreparazioni, preparazioni);
      nodoPreparazioni.value = preparazione;
      var ricette = ricercaMultipla(categoria, preparazione);
      var listaDescrizioni = calcolaListaDescrizioni(ricette);
      creaLista(nodoRisultato, listaDescrizioni);
   } catch ( e ) {
      alert ("gestorePreparazione " + e);
   }
}
```

La funzione *gestoreLoad* è così definita.

```
function gestoreLoad () {
   try {
      nodoCategorie = document.getElementById("categorie");
      nodoPreparazioni = document.getElementById("preparazioni");
      nodoRisultato = document.getElementById("risultato");
      var categorie = calcolaCategorie();
      creaSelect(nodoCategorie, categorie);
      aggiungiPrimaOpzione(nodoCategorie, "--- categoria ---");
      aggiungiPrimaOpzione(nodoPreparazioni, "--- preparazione ---");
      nodoCategorie.onchange = gestoreCategoria;
      nodoPreparazioni.onchange = gestorePreparazione;
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
```

15.8 Esercizi

1. Definire una pagina web che visualizza una lista di film, a ognuno dei quali è associata una lista di cinema presso i quali il film è in programmazione.
2. Definire una pagina web interattiva che visualizza una lista di film, a ognuno dei quali è associata una lista di cinema presso i quali il film è in programmazione. La pagina contiene un menu di selezione, generato dinamicamente, contenente tutti e soli i cinema presenti nella programmazione, un pulsante che visualizza la lista dei film in programmazione presso il cinema selezionato.

16 Pagine web interattive

Questo capitolo presenta alcuni esempi di pagine web interattive, introducendo ulteriori tecniche di programmazione.

16.1 Galleria fotografica

Una *galleria fotografica* è formata da una sequenza di fotografie visualizzate una alla volta. Il passaggio da una foto all'altra avviene in seguito alla richiesta dell'utente di visualizzare la foto successiva o quella precedente.

Le modalità di funzionamento di una galleria fotografica possono essere estese in vari modi. Il passaggio da una foto all'altra può avvenire in maniera automatica. Il cambiamento della foto può essere istantaneo o sfumato. Dopo aver visualizzato l'ultima (la prima) foto della sequenza si può visualizzare la prima (l'ultima). La foto precedente e quella successiva possono essere visualizzate in dimensioni ridotte ai due lati di quella corrente.

La galleria fotografica descritta nel seguito prevede il passaggio automatico, come opzione a scelta dell'utente, e il passaggio dall'ultima (dalla prima) alla prima (all'ultima) foto della sequenza, per dare all'utente un senso di continuità nella visualizzazione delle foto. La soluzione è minimale e non usa fogli di stile.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="Galleria.js"></script>
  <title>Galleria fotografica</title>
 </head>
 <body>
  <input type="button" value="Indietro" id="indietro"/>
  <img id="foto"/>
  <input type="button" value="Avanti" id="avanti"/>
  <br />
  <input type="button" id="startStop"/>
 </body>
</html>
```

Gli elementi interattivi sono i pulsanti *Avanti* e *Indietro* per il passaggio alla foto successiva e a quella precedente, e il pulsante *StartStop* per l'attivazione e la disattivazione della modalità automatica di visualizzazione. La foto è visualizzata nell'elemento con marca *img*, il cui identificatore è *foto*.

Il codice JavaScript è strutturato in maniera standard. La funzione *gestoreLoad* gestisce l'evento *load*, la funzione *gestoreClickAvanti* gestisce l'evento *click* sul pulsante *Avanti*, la funzione *gestoreClickIndietro* gestisce l'evento *click* sul pulsante *Indietro* e la funzione *gestoreClickStartStop* gestisce l'evento *click* sul pulsante *StartStop*.

Le foto della galleria sono contenute nella cartella del file HTML. I loro nomi sono memorizzati nell'array *galleria*. La variabile globale *indiceFoto* contiene l'indice della foto correntemente visualizzata. La variabile globale *automatico* vale *true* se la modalità di visualizzazione automatica è stata scelta dall'utente, *false* altrimenti. La variabile *numeroFoto* indica il numero delle foto della galleria e la costante *RITARDO* il numero di millisecondi che devono trascorrere per passare automaticamente da una foto all'altra. Infine, le variabili globali *nodoAvanti*, *nodoIndietro*, *nodoStartStop* e *nodoFoto* fanno riferimento ai relativi nodi dell'albero DOM.

Il codice della funzione *gestoreLoad*, preceduto dalle dichiarazioni delle costanti e delle variabili globali e seguito dalla registrazione dell'evento *load* è mostrato nel seguito. Da notare l'inizializzazione della proprietà *value* di *nodoStartStop*, fatta in modo che sia l'utente a scegliere esplicitamente la modalità automatica di visualizzazione.

```
const RITARDO = 2000;
var indiceFoto;
var numeroFoto;
var automatico;
var nodoAvanti;
var nodoIndietro;
var nodoStartStop;
var nodoFoto;
function gestoreLoad () {
   try {
      nodoAvanti = document.getElementById("avanti");
      nodoIndietro = document.getElementById("indietro");
      nodoStartStop = document.getElementById("startStop");
      nodoFoto = document.getElementById("foto");
      nodoAvanti.onclick = gestoreClickAvanti;
      nodoIndietro.onclick = gestoreClickIndietro;
      nodoStartStop.onclick = gestoreClickStartStop;
      nodoStartStop.value = "Start";
      numeroFoto = galleria.length;
```

```
      automatico = false;
      indiceFoto = 0;
      cambiaFoto(0);
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
window.onload = gestoreLoad;

var galleria = [
   "foto0.png",
   "foto1.png",
   "foto2.png",
   "foto3.png",
   "foto4.png",
   "foto5.png",
   "foto6.png",
   "foto7.png"
]
```

Il codice dei gestori dell'evento *click* sui pulsanti *Avanti* e *Indietro* usa la funzione *cambiaFoto* per passare da una foto all'altra. Il parametro di questa funzione è un intero che indica di quanto deve cambiare l'indice della foto visualizzata.

```
function gestoreClickAvanti () {
   try {
      if (!automatico) {
         cambiaFoto(+1);
      }
   } catch ( e ) {
      alert("gestoreClickAvanti " + e);
   }
}
function gestoreClickIndietro () {
   try {
      if (1automatico) {
         cambiaFoto(-1);
      }
   } catch ( e ) {
      alert("gestoreClickIndietro " + e);
   }
}
function cambiaFoto (x) {
   indiceFoto += x;
   if (indiceFoto == numeroFoto) {
      indiceFoto = 0;
   }
```

```
   if (indiceFoto < 0) {
      indiceFoto = numeroFoto - 1;
   }
   nodoFoto.setAttribute("src", galleria[indiceFoto]);
}
```

La gestione dell'evento *click* sul pulsante *StartStop* prevede il cambiamento della proprietà *value* di *nodoStartStop* (per informare l'utente che la modalità automatica è stata attivata o disattivata) e l'invocazione della funzione *cambiaFotoInAutomatico*. Questa funzione incrementa di uno l'indice della foto corrente e invoca la funzione predefinita *setTimeout* che ha come parametri il nome di una funzione e un valore numerico che determina dopo quanto tempo la funzione indicata nel primo parametro deve essere invocata. Se durante la visualizzazione automatica delle foto l'utente pigia il pulsante *StartStop* la variabile automatico cambia valore e fa sì che la funzione *cambiaFotoInAutomatico* non invochi *setTimeout*, interrompendo così la modalità automatica. Da notare che nella funzione *cambiaFotoInAutomatico* è stato usato il comando *try catch* perché la funzione compare come primo argomento di *setTimeOut*.

```
function gestoreClickStartStop () {
   try {
      if (automatico) {
         nodoStartStop.value = "Start";
         automatico = false;
      } else {
         nodoStartStop.value = "Stop";
         automatico = true;
         cambiaFotoInAutomatico();
      }
   } catch ( e ) {
      alert("gestoreClickStartStop " + e);
   }
}
function cambiaFotoInAutomatico () {
   try {
      if (automatico) {
         cambiaFoto(+1);
         setTimeout(cambiaFotoInAutomatico, RITARDO);
      }
   } catch ( e ) {
      alert("cambiaFotoInAutomatico " + e);
   }
}
```

16.2 Il gioco del Memory

Il *Memory* è un popolare gioco di carte che richiede concentrazione e memoria. Nel gioco, le carte sono inizialmente mescolate e disposte coperte sul tavolo. I giocatori, a turno, scoprono due carte; se queste formano una "coppia", sono incassate dal giocatore di turno, che può scoprirne altre due; altrimenti, sono nuovamente coperte e rimesse nella loro posizione originale sul tavolo, e il turno passa al prossimo giocatore. Vince il giocatore che riesce a scoprire più coppie. Il Memory può essere anche giocato come solitario, per esempio tenendo conto del numero di carte non corrispondenti scoperte e cercando di incassare tutte le carte nel minor numero possibile di tentativi [Wikipedia, alla voce *Memory*].

Per realizzare il Memory, come solitario, è necessario avere a disposizione un buon numero di immagini da utilizzare come lato posteriore di ogni carta (quello mostrato quando la carta è girata) e un'immagine per il lato anteriore (quello mostrato quando la carta è coperta, uguale per tutte le carte). Per rendere interessante il gioco le immagini scelte per il lato posteriore devono essere almeno otto. Le immagini devono essere memorizzate nella stessa cartella che conterrà il file HTML e quello JavaScript. Per semplificare la gestione delle immagini è conveniente usare un nome formato da un prefisso comune seguito da un numero. La convenzione adottata prevede di usare il prefisso *carta*. La numerazione inizia da zero. Il nome dell'immagine per il lato anteriore è *carta00*.

Il codice HTML è molto semplice ed è relativo a un Memory con sedici carte, ovvero otto coppie di carte uguali. Ogni riga contiene quattro immagini, identificate da un *id* unico (da *t0* a *t3*, per la prima riga). Ogni riga è separata dalla successiva da una marca *br*. Dopo la quarta riga è previsto un pulsante *Nuova partita*, con *id nuovaPartita*, che permette di iniziare il gioco in qualsiasi momento della partita.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="Memory.js"></script>
  <title>Memory</title>
 </head>
 <body>
  <img id="t0"/>
  <img id="t1"/>
  <img id="t2"/>
  <img id="t3"/>
  <br />
  <img id="t4"/>
  <img id="t5"/>
```

```
  <img id="t6"/>
  <img id="t7"/>
  <br />
  <img id="t8"/>
  <img id="t9"/>
  <img id="t10"/>
  <img id="t11"/>
  <br />
  <img id="t12"/>
  <img id="t13"/>
  <img id="t14"/>
  <img id="t15"/>
  <br />
  <input type="button" value="Nuova partita" id="nuovaPartita" />
 </body>
</html>
```

Il codice JavaScript è strutturato in maniera standard. La funzione *gestoreLoad* gestisce l'evento *load* e l'evento *click* sul pulsante *Nuova partita*. I nodi corrispondenti alle 16 marche *img*, una per ogni carta, sono memorizzati nell'array *carte*. A ogni nodo è associata la copertina della carta, mediante l'attributo *src*. Infine per ogni nodo è registrato l'evento *click*, gestito dalla funzione *gestoreClickCarta*. In questo modo, quando l'utente clicca su una carta la funzione *gestoreClickCarta* provvede a gestire l'evento. Per associare a ogni carta un'immagine si utilizza una funzione, *calcolaIndiceCarta*, che calcola in maniera casuale l'indice di una carta a cui non è stata ancora assegnata un'immagine. L'immagine è associata al nodo mediante l'attributo *src1*. Infine, le due variabili globali *primaCarta* e *secondaCarta* sono inizializzate a *null*.

```
const NUMERO_IMMAGINI = 8;
const NUMERO_CARTE = NUMERO_IMMAGINI * 2;
const RITARDO = 800;
const COPERTINA_CARTA = "carta00.png";
var carte;
var primaCarta;
var secondaCarta;
var nodoNuovaPartita;
function gestoreLoad () {
   try {
      nodoNuovaPartita = document.getElementById("nuovaPartita");
      nodoNuovaPartita.onclick = gestoreLoad;
      carte = [];
      for (var i = 0; i < NUMERO_CARTE; i++) {
         var idImmagine = "t" + String(i);
         var nodoCarta = document.getElementById(idImmagine);
         nodoCarta.setAttribute("src", COPERTINA_CARTA);
```

```
         nodoCarta.setAttribute("carta", "coperta");
         nodoCarta.onclick = gestoreClickCarta;
         carte.push(nodoCarta);
      }
      for (var i = 0;  i < NUMERO_IMMAGINI; i++) {
         var fileImmagine = "carta" + i + ".png";
         var i1 = calcolaIndiceCarta();
         carte[i1].setAttribute("src1", fileImmagine);
         carte[i1] = null;
         var i2 = calcolaIndiceCarta();
         carte[i2].setAttribute("src1", fileImmagine);
         carte[i2] = null;
      }
      primaCarta = null;
      secondaCarta = null;
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
window.onload = gestoreLoad;
```

La funzione *gestoreClickCarta* è invocata quando l'utente clicca su una carta. Se la carta è scoperta il gestore ignora l'evento. Se l'utente clicca sulla prima carta selezionata dall'utente il gestore ignora l'evento. Se l'utente clicca su una carta coperta e non ha ancora selezionato la prima carta, il gestore assegna alla variabile *primaCarta* il nodo che ha generato l'evento *click* e cambia l'immagine, rendendo visibile quella associata alla carta. Se l'utente clicca su una carta coperta e non ha ancora selezionato la seconda carta, il gestore assegna alla variabile *secondaCarta* il nodo che ha generato l'evento *click* e cambia l'immagine, rendendo visibile quella associata alla carta. A questo punto il gestore controlla se le immagini delle due pagine selezionate dall'utente sono uguali. In caso affermativo le rende scoperte, assegnando all'attributo *carta* il valore *scoperta* e il valore *null* alle variabili *primaCarta* e *secondaCarta*, altrimenti invoca la funzione *giraCarte* con un ritardo determinato dalla costante *RITARDO*. La funzione *giraCarte* si limita a cambiare l'immagine delle due carte selezionate, ripristinando la copertina.

```
function giraCarte () {
   try {
   primaCarta.setAttribute("src", COPERTINA_CARTA);
   primaCarta = null;
   secondaCarta.setAttribute("src", COPERTINA_CARTA);
   secondaCarta = null;
   } catch ( e ) {
      alert("giraCarte " + e);
```

```
   }
}
function gestoreClickCarta () {
   try {
      if (this.getAttribute("carta") == "scoperta") {
         return;
      }
      if (this == primaCarta) {
         return;
      }
      if (primaCarta == null) {
         primaCarta = this;
         this.setAttribute("src", this.getAttribute("src1"));
         return;
      }
      if (secondaCarta == null) {
         secondaCarta = this;
         this.setAttribute("src", this.getAttribute("src1"));
         if (primaCarta.getAttribute("src") ==
             secondaCarta.getAttribute("src")) {
            primaCarta.setAttribute("carta", "scoperta");
            primaCarta = null;
            secondaCarta.setAttribute("carta", "scoperta");
            secondaCarta = null;
         } else {
            setTimeout(giraCarte, RITARDO);
         }
      }
   } catch ( e ) {
      alert("gestoreClickCarta " + e);
   }
}
```

Il calcolo dell'indice della carta utilizza una funzione, *uniformeRandom*, che genera un numero casuale intero compreso tra zero e *k*. Per fare questo utilizza la funzione predefinita *Math.random* che genera un numero casuale reale compreso tra zero e uno (escluso).

```
function uniformeRandom (k) {
   return Math.trunc(Math.random() * k);
}
function calcolaIndiceCarta () {
   var i = uniformeRandom(NUMERO_CARTE);
   while (carte[i] == null) {
      i = (i + uniformeRandom(NUMERO_CARTE)) % NUMERO_CARTE;
   }
```

```
   return i;
}
```

16.3 Snake

Snake è un videogioco presente in molti telefonini, in particolare quelli prodotti dalla Nokia. Le sue origini risalgono agli anni settanta, con il videogioco arcade *Blockade*. Da allora è stato prodotto in numerose piattaforme e varianti, fino a ritrovare nuova fama negli anni novanta grazie ai cellulari. Snake è un serpente che mangiando quello che appare sul display si allunga facendo guadagnare dei punti al giocatore. Si muove costantemente e deve evitare di andare a sbattere contro gli ostacoli, ma soprattutto contro sé stesso, cosa sempre più difficile man mano che il suo corpo si allunga [Wikipedia, alla voce *Snake*].

La versione proposta di seguito è molto semplificata rispetto al gioco originale e alle sue innumerevoli varianti. Il serpente non si allunga durante la partita e non ha nulla da mangiare. Essendo molto corto non ha il problema di sbattere contro sé stesso, a meno che non cerchi di andare nella direzione opposta. Infine non ha ostacoli da evitare, tranne i bordi dell'area al cui interno si può muovere.

La scelta più importante da fare per realizzare il gioco è la rappresentazione dell'area. Per semplicità l'area è una *matrice bidimensionale* (elemento *table* in HTML, con *id matrice*) in cui le celle hanno una dimensione molto ridotta (un quadrato di tre *pixel* di lato). La matrice è creata dalla funzione *gestoreLoad* e tutte le sue celle hanno lo stesso colore di sfondo (azzurro chiaro). Il pulsante *Inizia* inizia la partita sia dopo che la pagina è stata caricata sia quando il serpente si ferma. Il serpente è lungo quattro celle, colorate di nero, ed è inizialmente posizionato al centro dell'area con la testa rivolta verso destra, direzione che prende quando inizia la partita. L'utente può cambiare la direzione di movimento del serpente pigiando i *tasti freccia* presenti sulla tastiera.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="Snake.js"></script>
  <link rel="stylesheet" href="Snake.css"></link>
  <title>Snake</title>
 </head>
 <body>
  <table id="matrice"></table>
  <br />
  <input type="button" value="Inizia" id="inizia"/>
 </body>
</html>
```

Il foglio di stile è il seguente.

```
table {
   border: 1px solid black;
   border-collapse: collapse;
}
th, td {
   width: 3px;
   height: 3px;
}
```

Per evitare che le celle della tabella siano separate una dall'altra è necessario assegnare alla proprietà di stile *border-collapse* il valore *collapse*.

Anche per questa pagina il codice JavaScript ha una struttura standard. La funzione *gestoreLoad*, le costanti e le variabili globali sono riportate nel seguito.

```
const NUMERO_RIGHE = 50;
const NUMERO_COLONNE = 50;
const BASSO = "ArrowDown";
const ALTO = "ArrowUp";
const DESTRA = "ArrowRight";
const SINISTRA = "ArrowLeft";
const COLORE_SFONDO = "lightblue";
const COLORE_SERPENTE = "black";
const RITARDO = 60;
var nodoMatrice;
var nodoInizia;
var serpente;
var direzione;
var finePartita;
function gestoreLoad () {
   try {
      nodoMatrice = document.getElementById("matrice");
      nodoInizia = document.getElementById("inizia");
      nodoInizia.onclick = gestoreClickInizia;
      document.onkeydown = gestoreKeyDown;
      creaMatrice();
      coloraMatrice(COLORE_SFONDO);
      finePartita = true;
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
window.onload = gestoreLoad;
```

L'evento *keydown* è generato dalla tastiera quando l'utente pigia un tasto. La creazione della matrice e la sua coloratura sono mostrati nel seguito.

```
function creaMatrice () {
   for (var i = 0; i < NUMERO_RIGHE; i++) {
      var nodoRiga = document.createElement("tr");
      nodoMatrice.appendChild(nodoRiga);
      for (var j = 0; j < NUMERO_COLONNE; j++) {
         var nodoCella = document.createElement("td");
         nodoRiga.appendChild(nodoCella);
      }
   }
}
function coloraMatrice (colore) {
   for (var i = 0; i < nodoMatrice.childNodes.length; i++) {
      var nodoRiga = nodoMatrice.childNodes[i];
      for (var j = 0; j < nodoRiga.childNodes.length; j++) {
         var nodoCella = nodoRiga.childNodes[j];
         nodoCella.style.backgroundColor = colore;
      }
   }
}
```

La funzione *gestoreClickInizia* gestisce l'evento *click* sul pulsante *Inizia*. Se la partita è già iniziata, il gestore ignora l'evento, altrimenti colora la matrice in modo da far sparire il serpente fermo da qualche parte, calcola le coordinate della sua testa (usando un numero casuale per la coordinata *x*, ovvero per la colonna della matrice) e inizializza la variabile *serpente* con le coordinate della testa, del corpo e della coda. Invocando la funzione *mostraSerpente* si ottiene la visualizzazione del serpente nella tabella corrispondente alla matrice. La direzione di marcia è inizializzata al valore costante *DESTRA* e, infine, la funzione *muoviSerpente* è invocata con un determinato ritardo, pari a *60* millisecondi. La funzione *mostraSerpente* annerisce le quattro caselle corrispondenti al serpente stesso.

```
function gestoreClickInizia () {
   try {
      if (finePartita) {
         finePartita = false;
         coloraMatrice(COLORE_SFONDO);
         var yTesta = uniformeRandom(NUMERO_COLONNE);
         var xTesta = NUMERO_RIGHE / 2;
         serpente = [{x : xTesta,     y : yTesta},
                     {x : xTesta - 1, y : yTesta},
                     {x : xTesta - 2, y : yTesta},
                     {x : xTesta - 3, y : yTesta}];
         mostraSerpente();
         direzione = DESTRA;
         setTimeout(muoviSerpente, RITARDO);
```

```
      }
   } catch ( e ) {
      alert("gestoreClickInizia " + e);
   }
}
function mostraSerpente () {
   for (var i = 0; i < serpente.length; i++) {
      cambiaColore(serpente[i], COLORE_SERPENTE);
   }
}

function cambiaColore (coord, colore) {
   nodoMatrice.rows[coord.y].cells[coord.x].style.backgroundColor =
      colore;
}
```

La funzione *muoviSerpente* modifica la matrice cambiando il colore della cella corrispondente alla coda e quello della cella che conterrà la testa. Il calcolo delle coordinate di questa cella, chiamata *nuovaTesta*, avviene utilizzando la *direzione* del serpente e le coordinate della testa prima del movimento. Se la nuova testa tocca il bordo della tabella la partita termina. Dopo aver cambiato il colore della cella della coda (che diventa quello dello sfondo) e il colore della cella della nuova testa (che diventa nera) la funzione sposta di uno le coordinate del corpo del serpente e assegna le coordinate della testa al primo elemento dell'array *serpente*. Infine, la funzione *muoviSerpente* è invocata con un determinato ritardo.

```
function muoviSerpente () {
   try {
      if (finePartita) {
         return;
      }
      var testa = serpente[0];
      var coda = serpente[serpente.length - 1];
      var nuovaTesta = {x : testa.x, y : testa.y};
      switch (direzione) {
         case ALTO : {
            nuovaTesta.y--;
            break;
         }
         case BASSO : {
            nuovaTesta.y++;
            break;
         }
         case DESTRA : {
            nuovaTesta.x++;
            break;
```

```
         }
         case SINISTRA : {
            nuovaTesta.x--;
            break;
         }
      }
      if ((nuovaTesta.x < 0 || nuovaTesta.x == NUMERO_COLONNE) ||
          (nuovaTesta.y < 0 || nuovaTesta.y == NUMERO_RIGHE)) {
         finePartita = true;
         return;
      }
      cambiaColore(nuovaTesta, COLORE_SERPENTE);
      cambiaColore(coda, COLORE_SFONDO);
      for (var i = serpente.length - 1; i > 0; i--) {
         serpente[i] = serpente[i - 1];
      }
      serpente[0] = nuovaTesta;
      setTimeout(muoviSerpente, RITARDO);
   } catch ( e ) {
      alert("muoviSerpente " + e);
   }
}
```

Il cambio di direzione del serpente è gestito dalla funzione *gestoreKeyDown*. Per prima cosa è necessario prevenire il comportamento di default associato al pulsante pigiato. Pigiando i tasti freccia, infatti, il browser sposterebbe il fuoco della pagina visualizzata. Dopo aver calcolato la nuova direzione, proprietà *code* del parametro formale *evento*, la funzione controlla che il tasto pigiato sia una delle quattro frecce, altrimenti ignora l'evento. Successivamente, la funzione controlla che la direzione indicata dal tasto freccia pigiato non sia opposta alla direzione corrente del serpente. In caso affermativo la partita termina, altrimenti la funzione aggiorna la variabile *direzione*.

```
function gestoreKeyDown (evento) {
   try {
      evento.preventDefault();
      var nuovaDirezione = evento.code;
      if ((nuovaDirezione != BASSO) &&
          (nuovaDirezione != ALTO) &&
          (nuovaDirezione != DESTRA) &&
          (nuovaDirezione != SINISTRA)) {
         return;
      }
      if ((nuovaDirezione == BASSO && direzione == ALTO) ||
          (nuovaDirezione == ALTO && direzione == BASSO) ||
          (nuovaDirezione == DESTRA && direzione == SINISTRA) ||
```

```
          (nuovaDirezione == SINISTRA && direzione == DESTRA)) {
         finePartita = true;
         return;
      }
      direzione = nuovaDirezione;
   } catch ( e ) {
      alert("gestoreKeyDown " + e);
   }
}
```

16.4 Quiz

Un *quiz* è una domanda o una serie di domande, in forma verbale o scritta, predisposte allo scopo di saggiare la preparazione o la memoria della persona alla quale vengono poste [Wikipedia, alla voce *Quiz*].

Le risposte a un quiz possono essere in *forma aperta* o *chiusa*. La forma aperta permette a chi risponde a una domanda di formulare liberamente la sua risposta. La forma chiusa, invece, richiede che chi risponde debba scegliere la sua risposta tra un elenco prestabilito di risposte.

La pagina web interattiva presentata realizza un quiz in forma chiusa formato da quattro domande ognuna con tre risposte predefinite. La pagina permette all'utente di passare da una domanda alla successiva, dopo aver eventualmente selezionato una risposta. Dopo la risposta all'ultima domanda la pagina riporta il numero di risposte corrette. Il quiz può essere interrotto in qualsiasi momento, facendolo iniziare dalla prima domanda.

La struttura della pagina è molto semplice: la prima riga contiene il numero della domanda corrente, la seconda riga contiene il testo della domanda corrente, le tre righe successive contengono le risposte predefinite, il pulsante *Avanti* permette all'utente di passare alla domanda successiva e il pulsante *Inizia* di iniziare di nuovo il quiz. L'esito del quiz è mostrato in una riga compresa tra i due pulsanti. Il testo delle risposte è preceduto da un elemento interattivo *radio* che permette di sceglierne solo una e di cambiare la scelta prima di passare alla domanda successiva.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="Quiz.js"></script>
  <title>Quiz</title>
 </head>
 <body>
  <span id="numeroDomanda"></span>
```

```
  <br />
  <span id="testoDomanda"></span>
  <br />
  <input type="radio" name="risposte" id="risposta0"/>
  <span id="testoRisposta0"></span>
  <br />
  <input type="radio" name="risposte" id="risposta1"/>
  <span id="testoRisposta1"></span>
  <br />
  <input type="radio" name="risposte" id="risposta2"/>
  <span id="testoRisposta2"></span>
  <br />
  <input type="button" value="Avanti" id="avanti" />
  <br />
  <span id="risultato"></span>
  <br />

  <input type="button" value="Inizia" id="inizia" />
 </body>
</html>
```

Il codice JavaScript ha una struttura standard.

```
var nodoNumeroDomanda;
var nodoTestoDomanda;
var nodoRisposta0;
var nodoTestoRisposta0;
var nodoRisposta1;
var nodoTestoRisposta1;
var nodoRisposta2;
var nodoTestoRisposta2;
var nodoAvanti;
var nodoRisultato;
var nodoInizia;
var numeroDomande;
var numeroDomandaCorrente;
var risposteDate;
function gestoreLoad () {
   try {
      nodoNumeroDomanda = document.getElementById("numeroDomanda");
      nodoTestoDomanda = document.getElementById("testoDomanda");
      nodoRisposta0 = document.getElementById("risposta0");
      nodoTestoRisposta0 = document.getElementById("testoRisposta0");
      nodoRisposta1 = document.getElementById("risposta1");
      nodoTestoRisposta1 = document.getElementById("testoRisposta1");
      nodoRisposta2 = document.getElementById("risposta2");
      nodoTestoRisposta2 = document.getElementById("testoRisposta2");
```

```
      nodoAvanti = document.getElementById("avanti");
      nodoRisultato = document.getElementById("risultato");
      nodoInizia = document.getElementById("inizia");
      nodoAvanti.onclick = gestoreClickAvanti;
      nodoInizia.onclick = gestoreClickInizia;
      numeroDomande = quiz.length;
      nuovoQuiz();
   } catch ( e ) {
      alert ("gestoreLoad " + e);
   }
}
window.onload = gestoreLoad;
```

La funzione *nuovoQuiz* fa partire il quiz dall'inizio azzerando la variabile *numeroDomandaCorrente* e invocando la funzione *aggiornaDomanda* affinché visualizzi la prima domanda. I contenuti del quiz, ovvero le domande, le relative risposte e il numero della risposta esatta, sono codificati nella variabile *quiz*.

```
function nuovoQuiz () {
   numeroDomandaCorrente = 0;
   aggiornaDomanda(numeroDomandaCorrente);
   scriviMessaggio(nodoRisultato, "");
   risposteDate = [];
}
function aggiornaDomanda (i) {
   scriviMessaggio(nodoNumeroDomanda,
                   "Domanda " + (i + 1) + " di " + numeroDomande);
   var parte = quiz[i];
   scriviMessaggio(nodoTestoDomanda, parte.domanda)
   scriviMessaggio(nodoTestoRisposta0, parte.risposte[0]);
   scriviMessaggio(nodoTestoRisposta1, parte.risposte[1]);
   scriviMessaggio(nodoTestoRisposta2, parte.risposte[2]);
   nodoRisposta0.checked = false;
   nodoRisposta1.checked = false;
   nodoRisposta2.checked = false;
}
var quiz = [
   {  // domanda 1
      domanda : "testoDomanda1",
      risposte :  [
                   "testoRisposta11",
                   "testoRisposta12",
                   "testoRisposta13"
                  ],
      rispostaEsatta : 0
   },
   {  // domanda 1
```

```
      domanda : "testoDomanda2",
      risposte :  [
                   "testoRisposta21",
                   "testoRisposta22",
                   "testoRisposta23"
                  ],
      rispostaEsatta : 0
   },
   {  // domanda 2
      domanda : "testoDomanda3",
      risposte :  [
                   "testoRisposta31",
                   "testoRisposta32",
                   "testoRisposta33"
                  ],
      rispostaEsatta : 0
   },
   {  // domanda 3
      domanda : "testoDomanda4",
      risposte :  [
                   "testoRisposta41",
                   "testoRisposta42",
                   "testoRisposta43"
                  ],
      rispostaEsatta : 0
   }
];
```

La funzione *scriviMessaggio* è una versione leggermente modificata di quella presentata negli esempi precedenti.

```
function scriviMessaggio (nodo, messaggio) {
   var nodoTesto = document.createTextNode(messaggio);
   if (nodo.childNodes.length == 0) {
      nodo.appendChild(nodoTesto);
   } else {
      nodo.replaceChild(nodoTesto, nodo.firstChild);
   }
}
```

Il gestore dell'evento *click* sul pulsante *Avanti*, la funzione *gestoreClickAvanti*, controlla se il quiz è terminato e in caso affermativo ignora l'evento. Altrimenti identifica la risposta selezionata dall'utente e memorizza il suo numero nell'array *risposteDate*, nella posizione corrispondente alla domanda corrente. Dopo aver incrementato di uno il numero della domanda corrente, la funzione verifica

se il quiz è terminato e, in caso affermativo, invoca *calcolaEsito* e visualizza il risultato.

```
function gestoreClickAvanti () {
   try {
      if (numeroDomandaCorrente == numeroDomande) {
         return;
      }
      if (nodoRisposta0.checked) {
         risposteDate[numeroDomandaCorrente] = 0;
      } else if (nodoRisposta1.checked) {
         risposteDate[numeroDomandaCorrente] = 1;
      } else if (nodoRisposta2.checked) {
         risposteDate[numeroDomandaCorrente] = 2;
      } else {
         return;
      }
      numeroDomandaCorrente++;
      if (numeroDomandaCorrente == numeroDomande) {
         var esito = calcolaEsito();
         var s;
         if (esito == 1) {
            s = "1 risposta esatta su " + numeroDomande;
         } else {
            s = esito + " risposte esatte su " + numeroDomande;
         }
         scriviMessaggio(nodoRisultato, s);
      } else {
         aggiornaDomanda(numeroDomandaCorrente);
      }
   } catch ( e ) {
      alert ("gestoreClickAvanti " + e);
   }
}
function calcolaEsito () {
   var numeroRisposteEsatte = 0;
   for (var i = 0; i < quiz.length; i++) {
      var parte = quiz[i];
      if (parte.rispostaEsatta == risposteDate[i]) {
         numeroRisposteEsatte++;
      }
   }
   return numeroRisposteEsatte;
}
```

Per iniziare un nuovo quiz l'utente deve pigiare il pulsante *Inizia*, a cui è associato la funzione *gestoreClickInizia*.

```
function gestoreClickInizia () {
   try {
      nuovoQuiz();
   } catch ( e ) {
      alert ("gestoreClickInizia " + e);
   }
}
```

16.5 Questionario

Un *questionario* è uno strumento sfruttato nella ricerca sociale (psicologia, sociologia, economia, studi di mercato, indagini di opinione) per raccogliere informazioni in modo standardizzato e su *campioni* più o meno grandi [Wikipedia alla voce *Questionario*]. Come i quiz, anche un questionario è formato da un gruppo di domande le cui risposte possono essere in forma aperta o chiusa. Se la dimensione del campione, ovvero le persone a cui il questionario è somministrato, si riduce a un'unità allora il questionario diventa uno strumento di analisi individuale. Esempi di questo tipo, chiamati anche *test di personalità*, sono molto comuni e si trovano su riviste, quotidiani e pagine web. Molto spesso la loro scientificità è dubbia, essendo formulati per fini ludici. Nel seguito saranno presi in considerazione questionari destinati a un uso individuale.

A differenza dei quiz, per ogni domanda non è definita una risposta corretta ma tutte le risposte concorrono a determinare un *profilo* di chi compila il questionario. Per semplicità si assume che il numero di profili sia determinato a priori. Al termine della compilazione del questionario l'analisi delle risposte determina il profilo che più si avvicina alle caratteristiche di chi l'ha compilato.

Per definire un questionario è necessario definire il *dominio* al cui interno sono determinati i profili. Un dominio può essere un qualunque argomento, purché sufficientemente complesso. Ad esempio, il questionario potrebbe essere sui colori, sui personaggi di un'opera letteraria, sul cibo, sul cinema e così via. La scelta del dominio e la determinazione dei profili sono due passi fondamentali da svolgere prima di procedere nelle fasi successive.

Dopo aver determinato il dominio e i profili, si procede alla stesura delle domande e delle relative risposte. Per evitare che la compilazione del questionario richieda troppo tempo, il numero delle domande deve essere limitato e non superare la decina. Come regola generale si assume che per ogni domanda ci sia sempre lo stesso numero di risposte. Anche per le risposte è bene non eccedere. Un valore accettabile è compreso tra *3* e *5*.

Ogni domanda deve essere formulata in modo da individuare un *aspetto* dei profili. Per questa ragione, ogni risposta deve essere formulata in modo da descrivere in maggior dettaglio l'aspetto relativo a non più di uno o due profili. Ad

esempio, se il dominio è relativo ai personaggi di un'opera letteraria, una domanda potrebbe essere basata sulle loro caratteristiche fisiche e, pertanto, ogni risposta deve fare riferimento alle caratteristiche fisiche di uno o al massimo due personaggi. Per esprimere questi concetti si usa un *vettore di affinità*, un array numerico la cui dimensione è pari al numero dei profili. Ogni elemento del vettore di affinità è un numero decimale compreso tra *0* e *1*, che indica quanto il profilo associato all'elemento è affine alla risposta fornita. Per convenzione, *0* significa nessuna affinità e *1* massima affinità. La somma dei valori di un vettore di affinità deve essere pari a *1*. Per ogni domanda sono definiti tanti *vettori di affinità delle risposte* quante sono le possibili risposte. La risposta scelta permette di identificare il *vettore di affinità della domanda*.

Al termine della compilazione del questionario, si sommano i vettori di affinità delle domande per ottenere il *vettore di affinità del questionario*. Per mantenere il vincolo sulla somma degli elementi, dopo avere effettuato la somma è necessario normalizzare il vettore di affinità del questionario, ovvero dividere il valore di ogni suo elemento per il numero delle domande. La ricerca del massimo del vettore di affinità del questionario permette di identificare il profilo da comunicare a chi l'ha compilato.

La seguente pagina web realizza un generico questionario formato da quattro domande, ognuna delle quali prevede tre risposte che permettono di determinare uno di quattro possibili profili.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="LibreriaQuestionario.js"></script>
  <script src="Questionario.js"></script>
  <script src="DatiQuestionario.js"></script>
  <title>Questionario</title>
 </head>
 <body>
  <span id="numeroDomanda"></span>
  <br />
  <span id="testoDomanda"></span>
  <br />
  <input type="radio" name="risposte" id="risposta0"/>
  <span id="testoRisposta0"></span>
  <br />
  <input type="radio" name="risposte" id="risposta1"/>
  <span id="testoRisposta1"></span>
  <br />
  <input type="radio" name="risposte" id="risposta2"/>
  <span id="testoRisposta2"></span>
```

```
  <br />
  <input type="button" value="Avanti" id="avanti" />
  <br />
  <span id="risultato"></span>
  <br />
  <input type="button" value="Inizia" id="inizia" />
 </body>
</html>
```

Come si può notare, il codice HTML è praticamente uguale a quello del quiz.

Il codice JavaScript è organizzato in tre file: *LibreriaQuestionario.js*, *Questionario.js* e *DatiQuestionario.js*. Il primo contiene alcune funzioni di supporto, il secondo contiene le funzioni che gestiscono l'interattività della pagina web, il terzo contiene i dati relativi alle domande, alle risposte e ai profili.

Il file *Questionario.js* contiene funzioni definite per il quiz, eventualmente leggermente modificate.

```
function scriviMessaggio (nodo, messaggio) { … }

function nuovoQuestionario () {
   numeroDomandaCorrente = 0;
   aggiornaDomanda(numeroDomandaCorrente);
   scriviMessaggio(nodoRisultato, "");
   risposteDate = [];
   matriceAffinita = [];
}

function aggiornaDomanda (i) {
   scriviMessaggio(nodoNumeroDomanda,
                   "Domanda " + (i + 1) + " di " + numeroDomande);
   var parte = questionario[i];
   scriviMessaggio(nodoTestoDomanda, parte.domanda)
   scriviMessaggio(nodoTestoRisposta0, parte.risposte[0]);
   scriviMessaggio(nodoTestoRisposta1, parte.risposte[1]);
   scriviMessaggio(nodoTestoRisposta2, parte.risposte[2]);
   nodoRisposta0.checked = false;
   nodoRisposta1.checked = false;
   nodoRisposta2.checked = false;
}

function gestoreClickAvanti () { … }


function gestoreClickInizia () {
   try {
      nuovoQuestionario();
```

```
   } catch ( e ) {
      alert ("gestoreClickInizia " + e);
   }
}
var nodoNumeroDomanda;
var nodoTestoDomanda;
var nodoRisposta0;
var nodoTestoRisposta0;
var nodoRisposta1;
var nodoTestoRisposta1;
var nodoRisposta2;
var nodoTestoRisposta2;
var nodoAvanti;
var nodoRisultato;
var nodoInizia;

var numeroDomande;
var numeroDomandaCorrente;
var risposteDate;
var matriceAffinita;

function gestoreLoad () {
   try {
      nodoNumeroDomanda = document.getElementById("numeroDomanda");
      nodoTestoDomanda = document.getElementById("testoDomanda");
      nodoRisposta0 = document.getElementById("risposta0");
      nodoTestoRisposta0 = document.getElementById("testoRisposta0");
      nodoRisposta1 = document.getElementById("risposta1");
      nodoTestoRisposta1 = document.getElementById("testoRisposta1");
      nodoRisposta2 = document.getElementById("risposta2");
      nodoTestoRisposta2 = document.getElementById("testoRisposta2");
      nodoAvanti = document.getElementById("avanti");
      nodoRisultato = document.getElementById("risultato");
      nodoInizia = document.getElementById("inizia");
      nodoAvanti.onclick = gestoreClickAvanti;
      nodoInizia.onclick = gestoreClickInizia;
      numeroDomande = questionario.length;
      creaAffinita();
      nuovoQuestionario();
   } catch ( e ) {
      alert ("gestoreLoad " + e);
   }
}
window.onload = gestoreLoad;
```

Le differenze rispetto al codice del quiz sono le seguenti. La variabile globale *quiz* diventa *questionario*; la funzione *nuovoQuiz* diventa *nuovoQuestionario*; viene introdotta la variabile globale *matriceAffinità* e la funzione *creaAffinità*.

Il file *LibreriaQuestionario.js* contiene le funzioni specifiche per la gestione del questionario.

```
function creaAffinita () {
   for (var i = 0; i < questionario.length; i++) {
      var parte = questionario[i];
      var punteggiRisposte = parte.punteggi;
      var affinitaRisposte = [];
      for (var j = 0; j < punteggiRisposte.length; j++) {
         var punteggiRisposta = punteggiRisposte[j];
         var sommaPunteggi = 0;
         for (var k = 0; k < punteggiRisposta.length; k++) {
            sommaPunteggi += punteggiRisposta[k];
         }
         var affinitaRisposta = [];
         for (var k = 0; k < punteggiRisposta.length; k++) {
            affinitaRisposta[k] = punteggiRisposta[k] / sommaPunteggi;
         }
         affinitaRisposte[j] = affinitaRisposta;
      }
      questionario[i].affinita = affinitaRisposte;
   }
}

function calcolaEsito () {
   for (var i = 0; i < questionario.length; i++) {
      var parte = questionario[i];
      var rispostaData = risposteDate[i];
      matriceAffinita[i] = parte.affinita[rispostaData];
   }
   var affinitaQuestionario = [];
   for (var i = 0; i < profili.length; i++) {
      affinitaQuestionario[i] = 0;
   }
   for (var i = 0; i < matriceAffinita.length; i++) {
      var vettoreAffinita = matriceAffinita[i];
      for (var j = 0; j < vettoreAffinita.length; j++) {
         affinitaQuestionario[j] += vettoreAffinita[j];
      }
   }
   var indiceMax = 0;
   for (var i = 0; i < affinitaQuestionario.length; i++) {
      if (affinitaQuestionario[i] > affinitaQuestionario[indiceMax]) {
```

```
            indiceMax = i;
        }
    }
    return indiceMax;
}
```

La funzione *creaAffinità* scandisce il questionario e, per ogni parte, calcola un array di vettori di affinità, uno per ogni risposta. Per ogni risposta viene prima calcolata la somma di tutti i punteggi e poi, utilizzando questo valore, viene effettuata la normalizzazione. La funzione *calcolaEsito* è organizzata in quattro passi. Il primo passo consiste nell'inizializzazione della variabile *matriceAffinità*, in cui sono riportati i vettori di affinità delle risposte date al questionario. Il secondo passo consiste nell'inizializzazione del vettore di affinità del questionario, la variabile *affinitaQuestionario*. Il terzo passo consiste nella somma dei vettori di affinità delle risposte date al questionario. Il quarto passo consiste nella ricerca del massimo del vettore di affinità del questionario. La funzione restituisce l'indice del massimo individuato nell'ultimo passo.

Il file *DatiQuestionario.js* contiene i dati del questionario.

```
var profili = [
    "Profilo 1",
    "Profilo 2",
    "Profilo 3"
]
var questionario = [
    {  // domanda 1
       domanda  :  "testoDomanda1",
       risposte :  [
                      "testoRisposta11",
                      "testoRisposta12",
                      "testoRisposta13"
                   ],
       punteggi :  [
                      [0, 2, 8],
                      [9, 1, 2],
                      [2, 9, 0]
                   ]
    },
    {  // domanda 1
       domanda : "testoDomanda2",
       risposte :  [
                      "testoRisposta21",
                      "testoRisposta22",
                      "testoRisposta23"
                   ],
```

```
      punteggi :  [
                     [1, 4, 9],
                     [0, 8, 2],
                     [9, 0, 0]
                  ]
   },
   {  // domanda 2
      domanda : "testoDomanda3",
      risposte :  [
                     "testoRisposta31",
                     "testoRisposta32",
                     "testoRisposta33"
                  ],
      punteggi :  [
                     [8, 3, 3],
                     [0, 2, 8],
                     [1, 9, 1]
                  ]
   },
   {  // domanda 3
      domanda : "testoDomanda4",
      risposte :  [
                     "testoRisposta41",
                     "testoRisposta42",
                     "testoRisposta43"
                  ],
      punteggi :  [
                     [1, 1, 6],
                     [6, 1, 2],
                     [3, 5, 0]
                  ]
   }
]
```

La variabile globale *profili* contiene la descrizione dei profili. La variabile globale questionario ha la stessa struttura della variabile globale quiz. L'unica differenza è la mancanza della *rispostaEsatta*, sostituita dai *punteggi*, un array di punteggi della stessa dimensione delle risposte. A sua volta ogni punteggio è costituito da un array di interi compresi tra 0 e 10, uno per ogni profilo. Da notare che l'array di punteggi viene normalizzato e diventa un vettore di affinità.

16.6 Un semplice drag and drop

L'operazione di *drag and drop* consiste nel puntare un elemento grafico presente in una pagina web e spostarlo in un altro punto della pagina. L'operazione inizia con il puntamento dell'elemento, continua con lo spostamento durante il

quale l'elemento resta agganciato al cursore, termina con il rilascio dell'elemento nella posizione prescelta *sganciandolo* dal cursore. Per ogni passaggio si generano degli eventi specifici: *dragstart* sull'elemento puntato da spostare, *dragover* sull'area in cui si vuole spostare l'elemento, *dragdrop* quando si sgancia l'elemento.

La seguente pagina web interattiva permette di effettuare un'operazione di drag and drop. La pagina contiene un elemento grafico chiamato *rettangolo* e un'*immagine* che può essere spostata all'interno del rettangolo. L'immagine è associata al file *immagine.png*, il valore della proprietà *draggable* è posto a *true*, per indicare che può essere spostata.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="DragDrop.css"></link>
  <script src="DragDrop.js"></script>
  <title>Un semplice Drag and Drop</title>
 </head>
 <body>
  <div id="rettangolo"></div>
  <img id="immagine" src="immagine.png" draggable="true"/>
 </body>
</html>
```

Il file *DragDrop.css* contiene una direttiva di stile per il rettangolo.

```
#rettangolo {
    width: 200px;
    height: 100px;
    border: 1px solid black;
}
```

Il codice JavaScript ha una struttura standard. La funzione *gestoreLoad* inizializza le variabili globali e registra l'evento *dragstart* sul nodo corrispondente all'immagine e gli eventi *dragover* e *drop* sul nodo corrispondente al rettangolo. Registra l'evento *mouseover* sul nodo corrispondente all'immagine per far sì che il cursore cambi quando passa sopra l'immagine.

```
var nodoImmagine;
var nodoRettangolo;
function gestoreLoad () {
   try {
      nodoImmagine = document.getElementById("immagine");
      nodoRettangolo = document.getElementById("rettangolo");
      nodoImmagine.ondragstart = gestoreDragStart;
```

```
      nodoRettangolo.ondragover = gestoreDragOver;
      nodoRettangolo.ondrop = gestoreDrop;
      nodoImmagine.onmouseover = gestoreCursore;
   } catch ( e ) {
      alert("gestoreLoad " + e);
   }
}
window.onload = gestoreLoad;
```

La funzione *gestoreDragStart* viene invocata quando l'utente seleziona l'immagine e inizia a spostarla. L'unico comando eseguito da questa funzione ha il compito di associare all'evento, in una sua struttura dati chiamata *dataTransfer*, l'identificatore dell'elemento che viene spostato. Se l'utente seleziona e sposta l'immagine, il valore dell'identificatore sarà proprio la stringa *immagine*.

```
function gestoreDragStart (evento) {
   try {
      evento.dataTransfer.setData("text", this.id);
   } catch ( e ) {
      alert("gestoreDragStart " + e);
   }
}
```

La funzione *gestoreDragOver* viene invocata quando l'utente sposta l'immagine sul rettangolo. Per evitare che lo spostamento sia interpretato dal browser come uno spostamento di testo è necessario disabilitare il comportamento standard, mediante la funzione predefinita *preventDefault*, già vista in precedenza.

```
function gestoreDragOver (evento) {
   try {
      evento.preventDefault();
   } catch ( e ) {
      alert("gestoreDragOver " + e);
   }
}
```

Infine, la funzione *gestoreDrop* viene invocata quando l'utente rilascia l'immagine all'interno del rettangolo. Anche in questo caso è necessario invocare la funzione predefinita *preventDefault* prima di gestire l'evento. I dati relativi all'immagine spostata sono recuperati da dataTransfer e, in particolare, da questa struttura si ricava l'identificatore dell'immagine spostata. Dopo aver ottenuto la stringa relativa all'identificatore si procede a recuperare il nodo DOM corrispondente all'identificatore stesso. Successivamente si appende questo nodo come primo figlio del nodo relativo al rettangolo, in modo da far sì che l'immagine sia posizionata al suo interno.

```
function gestoreDrop (evento) {
   try {
      evento.preventDefault();
      var dati = evento.dataTransfer.getData("text");
      this.appendChild(document.getElementById(dati));
   } catch ( e ) {
      alert("gestoreDrop " + e);
   }
}
```

Per far cambiare il cursore quando passa sopra l'immagine è sufficiente cambiare lo stile *cursor*, assegnando il valore *pointer*.

```
function gestoreCursore () {
   try {
      nodoImmagine.style.cursor = "pointer";
   } catch ( e ) {
      alert("gestoreCursore " + e);
   }
}
```

Questo esempio di *drag and drop* è molto semplice. Lo schema presentato può essere facilmente esteso per casi più complicati. Ad esempio, si potrebbe definire un altro rettangolo che contiene inizialmente l'immagine e permettere all'utente di spostarla da un rettangolo all'altro.

16.7 Immagine interattiva

Una immagine è *interattiva* quando ad alcune sue aree sono associati a uno o più eventi. Le immagini interattive sono usate in molte pagine web per consentire all'utente di individuare e selezionare al loro interno parti significative.

Per realizzare una pagine interattiva è necessario definire le aree in termini di *poligoni* i cui *vertici* sono specificati da *coordinate spaziali*. Per fare ciò in HTML si associa all'immagine un attributo *usemap*, il cui valore è un identificatore preceduto dal simbolo `#`. L'identificatore, a sua volta, è assegnato a un elemento *map*, al cui interno sono definite le aree. Ogni area ha un identificatore, l'attributo *shape* con il valore *poly* e l'attributo *coords* il cui valore è una lista di coordinate spaziali espresse in pixel. Per ottenere i valori delle coordinate spaziali che corrispondono ai vertici di un poligono si suggerisce di usare una delle tante applicazioni disponibili su internet.

L'immagine usata per questo esempio rappresenta *Tuttomondo*, un grande murale realizzato da *Keith Haring* nel *1989* sulla parete esterna della canonica della *chiesa di Sant'Antonio Abate* a Pisa. La superficie della parete misura circa *180* metri quadri (*10* metri di altezza per *18* metri di larghezza). Si tratta del più

grande murale mai realizzato in Europa, l'ultima opera pubblica dell'artista statunitense, nonché l'unica pensata per essere permanente [Wikipedia, alla voce *Tuttomondo*]. *Elvira Mercatanti* ha definito dodici aree nell'immagine.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script src="ImmagineInterattiva.js"></script>
  <title>Immagine interattiva</title>
 </head>
 <body>
  <img id="tuttomondo" src="Tuttomondo.jpg" usemap="#mappa" />
  <map name="mappa">
   <!-- Croce di Pisa -->
   <area id="0" shape="poly"
         coords="291,199,385,174,397,151,450,161,466,207,
                 534,220,505,274,415,265,459,316,455,348,
                 390,335,321,302,304,275,297,237"/>
   <!-- Artista -->
   <area id="1" shape="poly"
         coords="414,610,436,531,486,572,460,609,487,694,
                 365,694,359,648" />
   <!-- Uomo-TV -->
   <area id="2" shape="poly"
         coords="391,359,456,363,463,452,477,527,437,529,
                 409,524,381,470,390,413,394,413,390,413,
                 389,415,391,358" />
   <!-- Razze umane -->
   <area id="3" shape="poly"
         coords="637, 14,703, 44,727,127,752, 90,780,137,
                 739,167,743,197,776,166,789,226,748,239,
                 768,281,785,316,765,318,732,359,705,342,
                 669,247,573,250,671,131,645, 90,539, 18,
                 590, 16" />
   <!-- Serpente e forbici -->
   <area id="4" shape="poly"
         coords="443,117,500, 18,651,122,517,202,584,207,
                 654,160,487,199" />
   <!-- Maternità -->
   <area id="5" shape="poly"
         coords="561,322,605,313,645,393,634,495,590,497,
                 565,437,544,398" />
   <!-- Delfino -->
   <area id="6" shape="poly"
         coords="333, 24,393, 42,435, 22,435, 55,460, 70,
                 435, 77,402, 77,435,142,378,137,360,146,
```

```
                357,175,363,194,337,186,312,174,301,140,
                258,107,277, 71,261, 52,271, 29" />
   <!-- Infinito -->
   <area id="7" shape="poly"
         coords="178, 48,237, 38,265, 68,245,101,250,130,
                281,147,285,184,253,219,197,224,163,202,
                161,170,151,141,121,128,101,120,144, 74" />
   <!-- Cooperatività -->
   <area id="8" shape="poly"
         coords="288,530,332,550,415,625,355,629,335,694,
                289,694,214,605,253,544" />
   <!-- Lupo -->
   <area id="9" shape="poly"
         coords="162,311,235,311,251,383,297,391,296,426,
                323,448,337,485,301,487,273,529,251,503,
                221,471,213,421,215,335" />
   <!-- Uomo scala -->
   <area id="10" shape="poly"
         coords="293,374,298,284,269,237,149,241,128,142,
                103,245, 55,268,  6,246, 44,294,258,306,
                246,372" />
   <!-- Gemelli -->
   <area id="11" shape="poly"
         coords="281,510,359,459,422,564,380,592,349,555" />
   <!-- Scimmia -->
   <area id="12" shape="poly"
         coords=" 13,279,116,349, 69,406,  9,367" />
  </map>
  <br />
  <br />
  <span id="messaggio"></span>
  <h3>La definizione delle aree è di Elvira Mercatanti</h3>
 </body>
</html>
```

Il codice JavaScript è molto semplice. La funzione *gestoreLoad* registra l'evento *click* su ogni area dell'immagine e prepara un nodo per segnalare il nome dell'area quando l'utente ci clicca sopra. La variabile globale *simboliNascosti* è un array che contiene i nomi dei simboli rappresentati in ogni area.

```
var nodoMessaggio;
function gestoreLoad () {
   try {
      var nodiArea = document.getElementsByTagName("area");
      for (var i = 0; i < nodiArea.length; i++){
         nodiArea[i].onclick = gestoreClickArea;
      }
```

```
      nodoMessaggio = document.getElementById("messaggio");
      var nodoTesto = document.createTextNode("");
      nodoMessaggio.appendChild(nodoTesto);
      } catch ( e ) {
      alert ("gestoreLoad " + e);
   }
}
window.onload = gestoreLoad;

// contenuti
var simboliNascosti = [
   "Croce pisana",
   "Keith Haring",
   "Uomo TV",
   "Tre razze umane",
   "Forbici che tagliano serpente",
   "Maternità",
   "Uomo con delfino",
   "Infinito",
   "Cooperazione",
   "Cane",
   "Uomo scala",
   "Gemelli",
   "Scimmia"
];
```

La funzione *gestoreClickArea* si limita a scrivere un messaggio.

```
function gestoreClickArea () {
   try {
      scriviMessaggio(nodoMessaggio, simboliNascosti[this.id]);
   } catch ( e ) {
      alert ("gestoreClickArea " + e);
   }
}
```

16.8 Immagine interattiva scalabile

L'immagine interattiva descritta in precedenza ha un difetto: se in seguito a un'azione dell'utente le dimensioni della finestra che la contiene cambiano può accadere che una parte dell'immagine resti fuori dalla finestra stessa. Per eliminare questo difetto è necessario rendere l'immagine *scalabile*, mantenendo al tempo stesso la posizione delle sue aree.

La soluzione richiede una modifica al file HTML, in particolare alla marca *img*, alla quale viene aggiunto l'id *tuttomondo* e l'attributo di stile *max-width*

con il valore *100%*. In questo modo l'immagine diventa scalabile in seguito al variare delle dimensioni della finestra che la contiene.

```
<img id="tuttomondo" src="Tuttomondo.jpg"
     style="max-width:100%" usemap="#mappa" />
```

La seconda modifica riguarda le coordinate di ogni area, che devono essere riportate su una sola riga, senza andare a capo. Questo accorgimento è necessario per consentirne la modifica dinamica.

Il programma JavaScript è molto diverso da quello precedente. L'idea alla base della nuova soluzione consiste nel registrare l'evento *resize*, associato alla finestra, a un gestore che calcola le nuove coordinate delle aree utilizzando un fattore moltiplicativo (la variabile *ratio*) pari al rapporto tra l'attuale dimensione della foto (la variabile *attualeDimensioneFoto*) e la dimensione della foto prima della sua scalatura (la variabile *dimensioneFoto*). Per realizzare questa tecnica è necessario memorizzare le coordinate di ogni area in un array (la variabile *coordinateAree*) e aggiornare questo array ogni volta che la foto viene scalata.

La funzione *gestoreLoad* è così modificata.

```
const LARGHEZZZA_FOTO = 803;
var nodoTuttomondo;
var nodoMessaggio;
var nodiArea;
var coordinateAree;
var dimensioneFoto;
function gestoreLoad () {
   try {
      nodiArea = document.getElementsByTagName("area");
      nodoTuttomondo = document.getElementById("tuttomondo");
      nodoMessaggio = document.getElementById("messaggio");
      coordinateAree = [];
      for (var i = 0; i < nodiArea.length; i++){
         var nodoArea = nodiArea[i];
         nodoArea.onclick = gestoreClickArea;
         nodoArea.onmouseover = gestoreCursore;
         coordinateAree[i] = nodoArea.coords.split(',');
      }
      dimensioneFoto = LARGHEZZZA_FOTO;
      var nodoTesto = document.createTextNode("");
      nodoMessaggio.appendChild(nodoTesto);
      window.onresize = gestoreResize;
      gestoreResize();
   } catch ( e ) {
      alert ("gestoreLoad " + e);
```

```
   }
}
window.onload = gestoreLoad;
```

La funzione gestoreResize è così definita.

```
function gestoreResize () {
   try {
      var attualeDimensioneFoto = nodoTuttomondo.width;
      var ratio = attualeDimensioneFoto / dimensioneFoto;
      for (var i = 0; i < nodiArea.length; i++) {
         for (var j = 0; j < coordinateAree[i].length; j++) {
            coordinateAree[i][j] *= ratio;
         }
         nodiArea[i].coords = coordinateAree[i].join(',');
      }
      dimensioneFoto = attualeDimensioneFoto;
   } catch ( e ) {
      alert("gestoreResize " + e);
   }
}
```

Infine, la funzione gestoreCursore fa in modo che il cursore cambi forma quando il mouse passa sopra un'area.

```
function gestoreCursore () {
      try {
            this.style.cursor = "pointer";
      } catch ( e ) {
            alert("gestoreCursore " + e);
      }
}
```

17 Progetto didattico

Il *progetto didattico* previsto dal corso di *Progettazione e programmazione web* è un'attività individuale che consiste nella realizzazione di un sito web basato su un argomento liberamente scelto dallo studente. Il capitolo indica le linee guida e le regole per lo svolgimento di questa attività.

17.1 Il sito

La scelta dell'*argomento* del sito è lasciata completamente allo studente che può, in questo modo, esprimere al meglio le sue capacità creative e tecniche. Il sito si rivolge a una *platea di destinatari* ben definita, potenzialmente interessati a visitarlo.

La complessità e il livello di approfondimento dell'argomento richiedono una valutazione attenta e consapevole. Un argomento semplice può dar luogo a un sito che non valorizza le competenze e le conoscenze dello studente. Viceversa, un argomento troppo complesso può comportare un periodo di tempo eccessivo per la realizzazione del sito, spesso superiore a quello previsto per il progetto didattico.

L'argomento scelto può richiedere la raccolta di *contenuti* di varia natura (testi, immagini, filmati). Molto spesso accade che parte di questi contenuti sia stato realizzato da altre persone o istituzioni. Utilizzarli è consentito, a patto di citare puntualmente le fonti: i siti da cui sono state raccolte le informazioni, i libri, gli articoli, il materiale multimediale. Inoltre, è buona regola indicare esplicitamente le persone che hanno dato un contributo alla realizzazione del sito, specificando esattamente ciò che hanno fatto.

Occorre prestare molta attenzione alla dimensione dei contenuti non originali presentati nel sito. Ad esempio, non è consentito copiare interi paragrafi da altri siti o da libri perché ciò costituisce una violazione del diritto d'autore, anche se le fonti sono citate.

I contenuti raccolti sono presentati nel sito per renderne efficace la comunicazione e la fruizione da parte dei destinatari. La natura intrinsecamente multimediale e ipertestuale del sito richiede sinteticità, precisione e incisività. Ad esempio, è bene evitare che una pagina contenga testi lineari lunghi e poco strutturati.

Il sito è strutturato in più pagine, legate tra loro da un *tema grafico* comune. La pagina principale, chiamata *home page*, contiene il nome del sito e ne descrive sinteticamente le finalità. A partire da questa pagina si raggiungono le altre, coordinate tra loro, in cui l'argomento è sviluppato in maggior dettaglio. Il numero complessivo delle pagine varia da un minimo di *4* a un massimo (consigliato) di *10*.

Il sito contiene elementi di interattività che consentono all'utente di effettuare azioni che causano effetti visibili, secondo un dialogo gradevole e intuitivo.

Il *documento dei requisiti* descrive il progetto didattico nei suoi aspetti principali. La sua redazione inizia il più presto possibile, per consentire allo studente di riflettere sull'argomento da proporre, sui contenuti da raccogliere, sulle scelte di progettazione e su quelle di programmazione. Il modello del documento dei requisiti è disponibile sul sito web dei docenti.

Appena pronto, il documento è inviato per posta elettronica ai docenti che ne prendono visione e lo discutono durante gli incontri con lo studente. Anche se la redazione del documento dei requisiti non è una condizione necessaria per il superamento della verifica finale, l'esperienza acquisita finora dimostra che i progetti corredati da questo documento sono di buona qualità.

17.2 Le regole

Il progetto didattico si svolge secondo regole definite a priori, il cui rispetto è essenziale per il superamento della verifica finale. Le regole si dividono in due gruppi: *requisiti*, *suggerimenti*. La violazione di almeno un requisito comporta l'esclusione dalla prova d'esame. I suggerimenti sono indicazioni che lo studente può ignorare senza che ciò comporti l'esclusione dalla prova d'esame.

Requisiti

1. I file HTML sono redatti secondo lo standard HTML5 e validati con il validatore del W3C.
2. I file CSS sono validati con il validatore del W3C.
3. Lo stile è separato dal contenuto mediante l'uso di fogli di stile.
4. Il sito è *responsive* o almeno fluido per sfruttare al meglio lo spazio a disposizione su schermi di diversa dimensione.
5. Il sito ha una pagina con le generalità dell'autore, i riferimenti ai contenuti non originali utilizzati, le persone che hanno partecipato alla sua realizzazione.
6. I collegamenti ipertestuali presenti nel sito funzionano anche quando i file sono spostati altrove, ad esempio su un *web server*.

7. Il caricamento dei file JavaScript non genera errori sintattici.
8. Le pagine del sito non generano errori dinamici.
9. I file HTML non contengono codice JavaScript.
10. Il codice JavaScript non legge file esterni.
11. Il codice JavaScript non usa la proprietà *innerHTML*.
12. I singoli comandi JavaScript non sono commentati.
13. I gestori degli eventi contengono il comando *try catch*.
14. Il codice JavaScript ripreso integralmente da questo libro è opportunamente evidenziato.

Suggerimenti

1. La prima pagina del sito, la cosiddetta *home page*, si chiama *index.html*.
2. Il sito è visualizzato correttamente sui browser recenti.
3. I contenuti sono corretti dal punto di vista ortografico, senza errori di battitura.
4. Le immagini sono di buona qualità, non deformate e di dimensioni contenute.
5. Il codice JavaScript è correttamente indentato.
6. I contenuti gestiti dal codice JavaScript sono dichiarati in un file con estensione *js*, separato dagli altri file con la stessa estensione.
7. Le variabili globali sono inizializzate nel gestore dell'evento *load*.
8. Ad ogni funzione JavaScript è associato un commento che la descrive sommariamente.

17.3 La verifica finale

La consegna del progetto è effettuata secondo le modalità definite sul sito web dei docenti, entro il termine previsto per la registrazione indicato sul sito ufficiale dell'università. La mancata consegna del progetto entro questo termine comporta l'esclusione dalla prova d'esame.

I progetti presentati sono pubblicati su una pagina web del corso (chiamata *mappa dei progetti*) con un'indicazione dello stato (*pubblicato*) in cui si trovano. Dopo la pubblicazione sulla mappa dei progetti e prima della prova d'esame i docenti verificano il soddisfacimento dei requisiti e stabiliscono quali progetti didattici sono ammessi.

Dopo il superamento positivo della prova d'esame i progetti cambiano stato (*concluso*) e rimangono sulla mappa dei progetti.

La prova d'esame consiste in un orale durante il quale lo studente presenta il sito realizzato, illustra le scelte progettuali effettuate e, a richiesta dei docenti, spiega in dettaglio il codice (HTML, CSS, JavaScript), dimostrando di aver acquisito le conoscenze e le competenze previste dal programma del corso. Se necessario, lo studente motiva le ragioni che l'hanno portato a non seguire i suggerimenti previsti per il progetto didattico. Come indicato nei requisiti, il codice JavaScript ripreso integralmente da questo libro deve essere opportunamente evidenziato. Una tecnica per evidenziare il codice JavaScript è mostrata nel seguito.

```
/**/function gestoreCursore () {
/**/   try {
/**/      this.style.cursor = "pointer";
/**/   } catch ( e ) {
/**/      alert("gestoreCursore " + e);
/**/   }
/**/}
```

La valutazione del progetto è in trentesimi ed è determinata collegialmente dai docenti, tenendo in considerazione la preparazione sul programma del corso e, per quanto riguarda il progetto didattico, l'argomento scelto, i contenuti raccolti, la gradevolezza dell'interfaccia, il livello e la qualità dell'interattività, le caratteristiche del codice, l'osservanza dei suggerimenti e, infine, l'originalità e la creatività del sito.

Nel passato si sono verificati alcuni casi di *plagio*, ovvero di progetti realizzati da soggetti estranei al corso oppure copiati da materiale reperito su internet, spesso proprio dalla mappa dei progetti. Queste prassi sono assolutamente proibite e pregiudicano, senza alcuna eccezione, il superamento della prova d'esame. Nei casi più gravi, i docenti procedono come previsto dal *Codice etico della comunità accademica* (articoli 11 e 22).

18 Grammatica di JavaScript

La seguente grammatica non copre in modo esaustivo la sintassi di JavaScript, ma riporta le regole sintattiche introdotte nel libro. Il lettore interessato può trovare la sintassi completa in un qualsiasi manuale di JavaScript.

18.1 Parole riservate

```
false, true, null

this, new

var, const, function

return, break

if, else

switch, case, default

while, for, in

try, catch, finally, throw
```

18.2 Caratteri

```
<Lettera>  ::= a | b | c | d | e | f | g | h | i
             | j | k | l | m | n | o | p | q | r
             | s | t | u | v | w | x | y | z
             | A | B | C | D | E | F | G | H | I
             | J | K | L | M | N | O | P | Q | R
             | S | T | U | V | W | X | Y | Z

<Cifra>    ::= 0
             | <CifraNz>

<CifraNz>  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Speciale> ::= Space[18]
             | ! | " | # | $ | % | & | ' | ( | )
             | * | + | , | - | . | / | : | ; | <
             | = | > | ? | @ | [ | \ | ] | ^ | _
             | ` | { | | | } | ~
```

18 Carattere di spaziatura.

18.3 Identificatore

```
<Identificatore> ::= <CarIniziale>
                   |  <CarIniziale> <Caratteri>

<CarIniziale>    ::= <Lettera>
                   |  _
                   |  $

<Caratteri>      ::= <CarNonIniziale>
                   |  <CarNonIniziale> <Caratteri>

<CarNonIniziale> ::= <Lettera>
                   |  <Cifra>
                   |  _
                   |  $
```

18.4 Letterale

```
<Letterale>    ::= <Numero>
                 |  <Booleano>
                 |  <Stringa>
                 |  null

<Numero>       ::= <Intero>
                 |  <Intero>.<Cifre>
                 |  <Intero>E<Esponente>
                 |  <Intero>.<Cifre>E<Esponente>

<Intero>       ::= <Cifra>
                 |  <CifraNZ> <Cifre>

<Cifre>        ::= <Cifra>
                 |  <Cifra> <Cifre>

<Esponente>    ::= <Intero>
                 |  + <Intero>
                 |  - <Intero>

<Booleano>     ::= true
                 |  false

<Stringa>      ::= ""
                 |  "<CaratteriStr>"
                 |  ''
                 |  '<CaratteriStr>'

<CaratteriStr> ::= <CarattereStr>
                 |  <CarattereStr><CaratteriStr>

<CarattereStr> ::= <Lettera>
                 |  <Cifra>
                 |  <Speciale>
```

18.5 Espressione

```
<Espressione> ::= <Letterale>
                |  <Identificatore>
                |  (<Espressione>)
                |  <UnOp> <Espressione>
                |  <Espressione> <BinOp> <Espressione>
                |  this
                |  <Espressione>.<Identificatore>
                |  <Espressione>.<Chiamata>
                |  <Espressione>[<Espressione>]
                |  <Espressione> in <Espressione>
                |  <Chiamata>
                |  new <Identificatore>()
                |  new <Identificatore>(<Espressioni>)
                |  []
                |  [<Espressioni>]
                |  {}
                |  {<Coppie>}

<UnOp>         ::= -  | +  | !
<BinOP>        ::= -  | +  | *  | /  | %
                |  && | || |
                |  <  | <= | >  | >= | == | !=

<Espressioni> ::= <Espressione>
                |  <Espressione>, <Espressioni>

<Chiamata>     ::= <Identificatore>()
                |  <Identificatore>(<Espressioni>)

<Coppie>       ::= <Coppia>
                |  <Coppia>, <Coppie>
<Coppia>       ::= <Identificatore> : <Espressione>
                |  <Letterale> : <Espressione>
```

18.6 Programma, dichiarazione, comando, blocco

```
<Programma>        ::= <Comandi>

<Comandi>          ::= <Comando>
                     |  <Comando> <Comandi>

<Comando>          ::= <Dichiarazione>
                     |  <ComandoSemplice>
                     |  <ComandoComposto>

<Dichiarazione>    ::= <DicCost>
                     |  <DicVar>
                     |  <DicFun>
                     |  <DicCostr>

<ComandoSemplice>  ::= <Assegnamento>
                     |  <Invocazione>
                     |  <Return>
                     |  <Break>
                     |  <Throw>

<ComandoComposto>  ::= <Blocco>
                     |  <If>
                     |  <Switch>
                     |  <For>
                     |  <While>
                     |  <Try>

<Blocco>           ::= {<Comandi>}
```

18.7 Dichiarazione

```
<DicCost>      ::= const <Identificatore> = <Espressione>;

<DicVar>       ::= var <Identificatore>;
                |  var <Identificatore> = <Espressione>;

<DicFun>       ::= function <Identificatore>()
                      <Blocco>
                |  function <Identificatore>(<Parametri>)
                      <Blocco>

<Parametri>    ::= <Identificatore>
                |  <Identificatore>, <Parametri>

<DicCostr>     ::= function <Identificatore>()
                      {<BloccoCostr>}
                |  function <Identificatore>(<Parametri>)
                      {<BloccoCostr>}

<BloccoCostr>  ::= <PropMet>
                |  <PropMet> <BloccoCostr>

<PropMet>      ::= this.<Identificatore> = <Espressione>;
                |  this.<Identificatore> = <FunLet>

<FunLet>       ::= function ()
                      <Blocco>
                |  function (<Parametri>)
                      <Blocco>
```

18.8 Comando semplice

```
<Assegnabile>  ::= <Identificatore>
                 |  <Assegnabile>[<Espressione>]
                 |  this.<Assegnabile>
                 |  <Identificatore>.<Assegnabile>

<Assegnamento> ::= <Assegnabile> = <Espressione>;
                 |  <Assegnabile> += <Espressione>;
                 |  <Assegnabile> -= <Espressione>;
                 |  <Assegnabile> *= <Espressione>;
                 |  <Assegnabile> /= <Espressione>;
                 |  <Assegnabile> %= <Espressione>;
                 |  <Assegnabile>++;
                 |  <Assegnabile>--;

<Invocazione>  ::= <Assegnabile> ();
                 |  <Assegnabile> (<Espressioni>);

<Return>       ::= return <Espressione>;

<Break>        ::= break;

<Throw>        ::= throw <Espressione>;
```

18.9 Comando composto

```
<If>          ::= if (<Espressione>)
                  <Blocco>
                | if (<Espressione>)
                  <Blocco>
                  else <Blocco>

<Alternativa> ::= case <Letterale>: <Comandi>

<Alternative> ::= <Alternativa>
                | <Alternativa> <Alternative>

<Switch>      ::= switch (<Espressione>)
                  {<Alternative>}
                | switch (<Espressione>)
                  {<Alternative> default: <Comandi>}

<For>         ::= for (<Comando>; <Espressione>; <Comando>)
                  <Blocco>
                | for (var <Identificatore> in <Espressione>)
                  <Blocco>

<While>       ::= while (<Espressione>)
                  <Blocco>
<Try>         ::= try {<Blocco>}
                  catch (<Identificatore>) {<Blocco>}
                | try {<Blocco>}
                  catch (<Identificatore>) {<Blocco>}
                  finally {<Blocco>}
```

Indice analitico