Università di Pisa
Dipartimento di Informatica

Programmazione in JavaScript

Vincenzo Ambriola

Versione 3.1 ~~~ 5 dicembre 2012

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.

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 invita-

to a risolvere non solo con carta e matita ma mediante un calcolatore. Gli strumenti necessari sono alla portata di tutti: un browser di nuova generazione è più che sufficiente. La soluzione degli 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 sarebbe stato praticamente impossibile capire 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 parlato che usiamo tutti i giorni 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 ambigui, perché il significato delle *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 in cui sono inserite. 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 può 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 si trovano.

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 $A = \{a, b, c\}$ 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 A definito in precedenza si possono formare le frasi *aa*, *abba*, *caab*.

Dato un alfabeto 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* A. Sempre usando l'esempio precedente, è possibile formare l'insieme delle frasi di 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 A, un *linguaggio* L *su* A è un sottoinsieme delle frasi di A. Questa definizione è diversa da quella data in precedenza: non tutte le frasi di A sono, infatti, anche frasi di L.

Abbiamo fatto riferimento a "un" linguaggio e non "del" linguaggio su un alfabeto perché può esistere più di un linguaggio su un dato alfabeto. 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;
- *regole*: definiscono le relazioni tra i simboli terminali e i non-terminali; mediante la loro applicazione si ottengono le frasi del linguaggio.

Un linguaggio L le cui frasi rispettano le regole della grammatica G si dice *generato da* G.

2.3 Backus-Naur Form

La *Backus-Naur Form* (*BNF*) è uno dei formalismi più usati per definire le grammatiche. In accordo con la definizione 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 regole per la formazione delle frasi sono di due tipi:

- $X ::= S$

  si legge "*X* può essere sostituito con *S*" o "*X* produce *S*" (da cui il nome di *produzioni* dato alle regole). *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, ..., o S_n". Anche in questo caso *X* è un simbolo non-terminale, mentre S_1, S_2, ..., S_n sono sequenze di simboli terminali e non-terminali.

Il linguaggio *L* generato da una grammatica *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 regole di *G*.

Applicare una regola *R* a una sequenza *s* di simboli significa sostituire in *s* un'occorrenza del simbolo non-terminale definito da *R* con una sequenza della parte sinistra di *R* .

Per rendere concreti i concetti presentati finora, mostriamo la grammatica 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*> — regola di <*S*>
- <*A*> ::= *a* | *aa* — regola di <*A*>
- <*B*> ::= *b* | *bb* — regola di <*B*>

Il linguaggio generato da 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 G_2:

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

Il linguaggio infinito generato da 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 regola 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 cui appartengono le regole di derivazione.

A titolo di esempio, consideriamo la sequenza di derivazione della frase *aab* appartenente al linguaggio generato da G_2:

<S>

=> **Regola** <S> ::= <A> <B>

<A> <B>

=> **Regola** <A> ::= *a* | *a* <A> seconda opzione

a <A> <B>

=> **Regola** <A> ::= *a* | *a* <A> prima opzione

a a <B>

=> **Regola** <B> ::= *b* | *b* <B> prima opzione

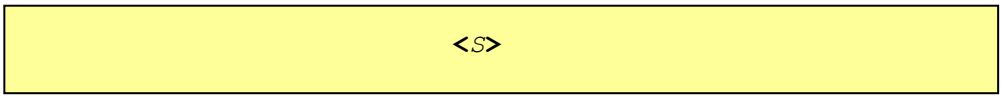
a a b

2.5 Alberi di derivazione

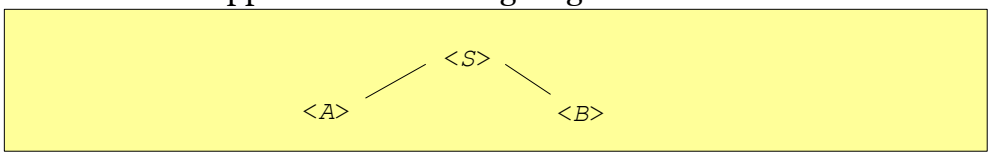
Un metodo alternativo per rappresentare il processo di derivazione si basa sugli *alberi di derivazione*. Per costruire l'*albero sintattico* di una frase si parte dal simbolo iniziale della grammatica, che ne costituisce la *radice*. Successivamente, per ogni *foglia* che è un simbolo non-terminale, si sceglie una regola 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 regole di derivazione a partire dal simbolo iniziale, ma la sua struttura finale non dipende dall'ordine di applicazione delle regole. 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.

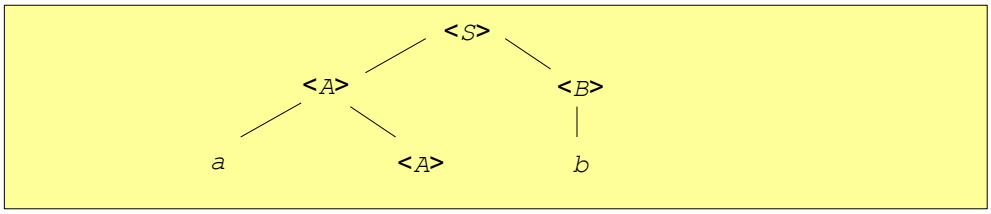
Ecco un esempio di costruzione dell'albero sintattico per la frase *aab* appartenente al linguaggio generato da G_2:



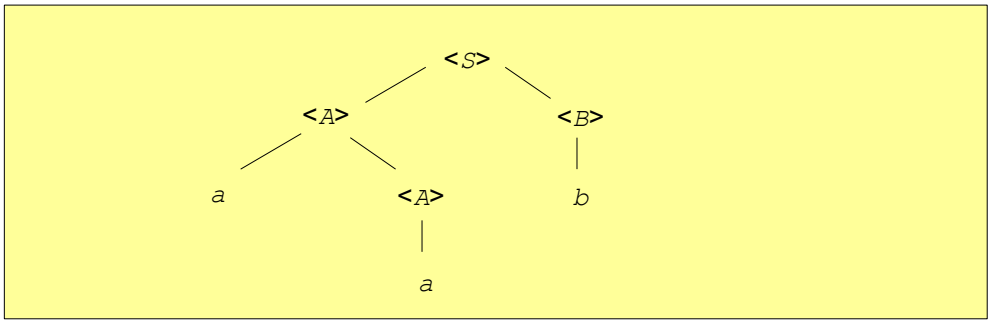
Applichiamo la regola <*S*> ::= <*A*> <*B*> al nodo radice dell'albero di derivazione. L'applicazione della regola genera due nuovi nodi.



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



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



3 Programmi, comandi e costanti

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 dinamiche e interattive le pagine web 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 è anche possibile aggiungere effetti grafici o accedere alle funzionalità del browser in cui le pagine sono visualizzati (lanciare una stampa, aprire una nuova finestra, ridimensionare o spostare sullo schermo una finestra di visualizzazione).

Con l'evoluzione dei browser, JavaScript ha subito numerose modifiche, acquisendo progressivamente nuove funzionalità. Per evitare possibili confusioni, il libro fa riferimento a JavaScript 1.8.2, la versione del linguaggio accettata dal browser *Firefox* (versione 15.0 e successive).

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 presentare gli aspetti di base del linguaggio senza dover introdurre la complessità della programmazione dinamica delle pagine web, argomento trattato nella seconda parte. Per rendere concreta la presentazione del linguaggio, il libro fa riferimento a un *ambiente di programmazione* chiamato *EasyJS*[1]. Questo ambiente consiste in

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

una pagina web in cui è possibile 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>
```

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 Costanti numeriche e logiche

In JavaScript una costante può essere 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.

```
<Costante>  ::= <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 *costanti numeriche* sono i seguenti:

- *12*
- *159.757*
- *23E5*

- *51E+2*
- *630E−6*
- *2.321E4*
- *3.897878E+7*
- *9.7777E−1*

La costante *2* denota il valore *due*, la costante *159.757* denota il valore *centocinquantanove virgola settecentocinquantasette*. Le altre costanti, in *rappresentazione esponenziale* possono essere trasformate 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*

La costante *23E5* denota il valore *duemilionitrecentomila*, la costante *51E+2* denota il valore *cinquemilacento*, mentre la costante *630E−6* denota il valore *zero virgola zerozerozeroseicentotrenta*. Il valore denotato dalle altre costanti è facilmente desumibile.

Le *costanti booleane*, dette anche *costanti logiche*, sono due. La costante *true* denota il valore di verità *vero*, la costante *false* denota il valore di verità *falso*.

3.3 Costanti 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 *costante* 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).

```
<Costante>       ::= <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, le costanti stringa devono essere scritte 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(<Costante>);
```

L'esecuzione di un comando di stampa ha l'effetto di scrivere sulla finestra inferiore di *EasyJS* la *costante* racchiusa tra *parentesi tonde*.

Ad esempio, il seguente programma stampa tre costanti.

```
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*. Una costante è un'espressione semplice, il valore della costante 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> ::= <Costante>
                | (<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 prevede che una stringa *a* è minore di una stringa *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* che rappresenta 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*, che rappresenta 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 costanti 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, default, else, false, for, function, if, in,
new, null, return, switch, this, true, 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>   ::= <DicVariabile>

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

5.1 Dichiarazione di costante

Una convenzione molto diffusa prevede che le costanti definite in un programma abbiano un identificatore formato solo da lettere maiuscole, da numeri e dal carattere _. Ciò permette di distinguerle immediatamente dalle variabili. La costante π (*pi greco*), il cui valore approssimato alle prime dieci cifre decimali è 3,1415926535, può essere dichiarata come una variabile con un valore iniziale.

```
 var 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 += i*, il cui significato è: applica l'operatore + alla variabile *x* e all'espressione alla destra del segno = e assegna il risultato 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 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 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> ::= <DicFunzione>

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

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

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 parametro 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 di 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 a tutti i parametri formali che non hanno un corrispondente parametro attuale, ovvero agli ultimi della lista;
- se il numero di 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. È possibile invocare una funzione nelle espressioni o assegnare

a una variabile il risultato dell'invocazione di una funzione. Per restituire un valore una funzione deve eseguire il comando *return*.

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

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

Come esempio, definiamo una funzione che restituisce la somma dei suoi parametri.

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

La funzione può essere invocata in un qualunque punto di un programma in cui è lecito usare un'espressione, ad esempio in un comando di assegnamento.

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

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].

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

6.1 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

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.

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.

6.2 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*.

6.3 Esercizi

1. 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*.
2. 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*.
3. 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*.

7 Comandi condizionali

Finora abbiamo definito la sintassi di JavaScript che permette di scrivere programmi molto semplici, formati da una sequenza di dichiarazioni di variabili, di assegnamenti e di comandi di stampa. Con questi programmi 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*.

7.1 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>

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

Il comando inizia con la parola riservata *if*, seguita da un'espressione tra parentesi tonde, detta *condizione* o *guardia*. 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;
}
```

7.2 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 <Costante>: <Comandi>

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

<ComandoSemplice> ::= break;
```

L'espressione (chiamata anche *selettore*) è valutata e il risultato è confrontato con quello delle costanti 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 al comando 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.

7.3 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));
}
```

7.4 Esercizi

1. 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*.

2. 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*.

3. 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*.

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 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>
```

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

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 *1,41421356 23*.

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 compositi*, 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 omogenei tra loro[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 solo elemento, *12*. Nel terzo caso un array con quattro elementi, i caratteri *f*, *u*, *n*, ed *e*.

6 Come vedremo nella seconda parte del libro, in JavaScript gli array possono essere disomogenei. Per semplicità di presentazione, nella prima parte assumiamo che siano omogenei.

Una volta creato 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]);
```

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]);
```

Per scandire tutti gli elementi di un array in JavaScript si usa il comando di iterazione determinata. La funzione *stampaElementi* 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 (i in a) {
      print(a[i]);
   }
}
var a1 = [2, 3, 5];
stampaElementi(a1);
var a2 = [0, -9, 17, 4, 100];
stampaElementi(a2);
```

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*.

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

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 un elemento a un array 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. Per scandire tutti gli elementi di un array associativo si usa una forma particolare di iterazione determinata che prevede un indice e un array. L'indice assume tutti i valori utilizzati per definire l'array[7].

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

Nell'esempio che segue l'indice *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);
}
```

9.5 Stringhe di caratteri

Un caso particolare di array è costituito dalle stringhe di caratteri che in JavaScript sono oggetti predefiniti con alcune 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);
```

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]);
```

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

```
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 un carattere di una stringa che, pertanto, è trattata in JavaScript come una costante.

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]);
   }
}
```

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 ((i < a.length) && (a[i] != k)) {
      i++;
   }
   return i;
}
```

Anche lo schema di ricerca lineare incerta si basa su un'iterazione indeterminata e da una guardia è 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. Un problema che può essere risolto usando questo schema è il seguente: definire una funzione che calcola e restituisce il valore dell'indice di un elemento di *a* che vale *k*, senza sapere a priori che un tale elemento appartiene all'array.

```
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 l'array *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 è possibile utilizzare 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 di numeri può essere ordinato in senso crescente o decrescente utilizzando un *algoritmo di ordinamento*. Il più semplice di questi algoritmi si basa sulla ricerca successiva del minimo (ordinamento crescente) o del massimo (ordinamento decrescente).

```
function ordina(a) {
   for (var i = 0; i < a.length; 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 pari 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, 2, 6, 8, 31].

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 6

1. 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));
```

2. 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.

3. 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 7

1. 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 = 0;
   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;
      case  1: n += giorno;
   }
   return n;
}
print(numeroGiorno(1957, 4, 25));
print(numeroGiorno(2004, 11, 7));
print(numeroGiorno(2000, 12, 31));
print(numeroGiorno(2012, 2, 29));
```

2. 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) {
 // da definire
}
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));
```

3. 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));
```

10.5 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) {
   var 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);
```

10.6 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 pari 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, 2, 6, 8, 31].

```
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. Abbiamo già visto questo concetto quando abbiamo parlato di grammatiche formali: una regola 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. Passiamo, quindi, ad affrontare un problema che può essere risolto con una funzione ricorsiva.

11.1 Fattoriale

Il *fattoriale* di *n* (indicato con !*n*) è il prodotto dei primi *n* numeri interi positivi minori o eguali 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) {
   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 trovare 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 nuove coppie nate si comportano in modo analogo; le coppie fertili, dal secondo mese di vita, danno alla luce una coppia di conigli al mese.

Vediamo in azione la definizione della successione. Partendo con una singola coppia, questa dopo un mese sarà fertile. Dopo due mesi avremo due coppie, di cui una sola fertile. Nel mese seguente avremo (*2 + 1 = 3*) tre coppie, perché solo la coppia fertile ha 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.

Definiamo 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);
   }
}
```

Definiamo un'altra funzione equivalente, 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 contenga lo *zero* e il successore di ogni proprio elemento coincide con l'intero insieme dei numeri naturali (assioma dell'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 l'addizione e la moltiplicazione sui numeri naturali sono definite per *induzione* utilizzando esclusivamente l'uguaglianza, la costante *zero* e l'operatore unario *successore*[8]*:*

$$x+0=x$$
$$x+y=x+y-1+1=(x+(y-1))+1$$
$$x\cdot 0=0$$
$$x\cdot y=x\cdot(y-1+1)=x\cdot(y-1)+x$$

In JavaScript è possibile definire ricorsivamente le funzioni che realizzano questi due operatori utilizzando la costante *0*, l'operatore binario di uguaglianza (==), 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);
   }
}
```

Gli operatori aritmetici di *sottrazione*, *divisione* e *resto*, *potenza* possono essere definiti per induzione sui numeri naturali utilizzando la costante *zero*, l'operatore binario di uguaglianza (==), l'operatore bi-

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

nario di disuguaglianza (*!=*), l'operatore di incremento unitario (*+1*), l'operatore di decremento (*-1*) e gli operatori di *addizione* e *moltiplicazione* definiti sopra.

La *sottrazione* è così definita per induzione:

$$x-0=x$$
$$x-y=x-y-1+1=(x-1)-(y-1)\ se\ x\sim y$$

Per definire la *divisione* è necessario dare una definizione induttiva dell'operatore di relazione *<:*

$$x<0\Rightarrow false$$
$$0<y\Rightarrow true\ se\ y>0$$
$$x<y\Rightarrow x-1<y-1\ se\ x>0\ e\ y>0$$

La *divisione* è così definita per induzione:

$$\frac{x}{y}=0\ se\ x<y$$
$$\frac{x}{y}=\frac{x-y+y}{y}=\frac{x-y}{y}+\frac{y}{y}=\frac{x-y}{y}+1\ se\ x\sim y$$

L'operatore non è definito se $y=0$.

Il *resto* della divisione è così definito per induzione:

$$x\ r\ y=x\ se\ x<y$$
$$x\ r\ y=(x-y)\ r\ y\ se\ x\geq y$$

Anche in questo caso l'operatore non è definito se $y=0$.

La *potenza* è così definita per induzione:

$$x^0=1\ se\ x>0$$
$$x^y=x\cdot x^{(y-1)}\ se\ x>0$$
$$0^y=0\ se\ y>0$$

Da notare che 0^0 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 Oggetti

In JavaScript, oltre agli oggetti predefiniti presentati nella prima parte (stringhe e array) è possibile usare *oggetti personalizzati*, con proprietà e metodi specifici.

Per creare un oggetto personalizzato è necessario dichiarare una *funzione costruttrice* (o *costruttore*) che ne definisce le proprietà e i metodi.

```
<fun_let>       ::= function ()
                    <blocco>
                  | function (<parametri>)
                    <blocco>

<prop_met>      ::= this.<identificatore> = <espressione>;
                  | this.<identificatore> = <fun_let>

<blocco_costr>  ::= <prop_met>
                  | <prop_met> <blocco_costr>

<dic_costr>     ::= function <identificatore>()
                    {<blocco_costr>}
                  | function <identificatore>(<parametri>)
                    {<blocco_costr>}
```

La dichiarazione di costruttore è simile alla dichiarazione di funzione. Anche il costruttore ha un'intestazione con una lista di parametri formali, eventualmente vuota, e un blocco che costituisce la definizione vera e propria del costruttore. A differenza della dichiarazione di funzione, tuttavia, nel blocco del costruttore sono ammesse solo dichiarazioni di proprietà e di metodi. Inoltre, nel corpo di un costrut-

tore può essere usata la parola riservata *this*[9], che fa riferimento all'oggetto in fase di costruzione.

La dichiarazione di una proprietà è simile alla dichiarazione di una variabile, con l'unica differenza che il nome della proprietà deve essere preceduto dalla parola riservata *this* e da un punto. La dichiarazione di un metodo è simile alla dichiarazione di una funzione. Anche in questo caso il nome della funzione deve essere preceduto dalla parola riservata *this* e da un punto. Ciò che cambia è l'intestazione in cui manca il nome della funzione, precedentemente indicato nella dichiarazione.

Per convenzione, i nomi dei costruttori iniziano con una lettera maiuscola, al contrario delle variabili e delle funzioni il cui nome inizia sempre con una lettera minuscola.

```
function Automobile (mar, mod, col, cil){
   this.marca = mar;
   this.modello = mod;
   this.colore = col;
   this.cilindrata = cil;
}
```

Il costruttore *Automobile* definisce un oggetto che ha quattro proprietà: *marca*, *modello*, *colore*, *cilindrata*. Il costruttore non ha metodi associati.

Il valore iniziale delle proprietà di un oggetto è stabilito al momento della sua creazione che può essere effettuata con le stesse modalità viste per la creazione degli array, utilizzando cioè la parola riservata *new* seguita dal nome del costruttore e da una lista di parametri.

```
<espressione> ::= new <identificatore>()
                | new <identificatore>(<espressioni>)
```

Dopo aver definito il costruttore *Automobile* è possibile utilizzarlo per creare alcuni oggetti.

9 JavaScript non segnala errori se la parola riservata *this* è utilizzata al di fuori della definizione di un costruttore. Questo uso è riservato a situazioni particolari e può causare comportamenti talvolta poco comprensibili.

```
var a1 = new Automobile('Ford',   'Focus', 'Grigio', 1.8);
var a2 = new Automobile('Fiat',   'Panda', 'Rosso',  1.1);
var a3 = new Automobile('Volvo',  'V70',   'Nero',   2.4);
var a4 = new Automobile('Lancia', 'Libra', 'Blu',    1.9);
```

Dopo aver creato un oggetto, è possibile accedere alle sue proprietà per utilizzarne o modificarne il valore. Anche per le proprietà degli oggetti si utilizza la notazione a punti vista nella prima parte.

```
var descrizione = a1.marca + ' ' + a1.modello;
a2.modello = 'Punto';
a2.colore = 'Nero';
a2.cilindrata = 1.3;
```

Il primo comando dichiara la variabile *descrizione* a cui è assegnato come valore iniziale una stringa ottenuta concatenando i valori delle proprietà *marca* e *modello* dell'oggetto *a1*. I tre comandi successivi trasformano l'oggetto *a2*, facendolo diventare una *Punto nera* di *1.3* di cilindrata. Il valore della proprietà *marca* resta invariato.

12.1 Metodi

Nella dichiarazione di un costruttore è possibile definire, oltre alle proprietà, anche i metodi, funzioni che possono utilizzare e modificare il valore delle proprietà dell'oggetto stesso. Per questo motivo, come nel caso dei costruttori, nel corpo dei metodi può essere utilizzata la parola riservata *this*, che fa riferimento all'oggetto su cui il metodo è invocato.

```
function Automobile (mar, mod, col, cil){
   this.marca = mar;
   this.modello = mod;
   this.colore = col;
   this.cilindrata = cil;
   this.serbatoio = 0;
   this.rifornisci =
      function(c){
         this.serbatoio += c;
      }
   this.descrivi =
      function() {
```

```
        return this.marca + ' ' +
               this.modello + ' ' +
               this.colore + ' ' +
               this.cilindrata;
    }
}
```

Al costruttore *Automobile* è stata aggiunta una nuova proprietà, *serbatoio*, inizializzata a zero al momento della creazione dell'oggetto.

Oltre alla nuova proprietà, il costruttore definisce due metodi: *rifornisci* e *descrivi*. Il primo incrementa il valore della proprietà *serbatoio*, simulando così il rifornimento effettuato presso una stazione di servizio. Il secondo restituisce una stringa che descrive l'oggetto.

```
a1.rifornisci(50);
var descrizione = a1.descrivi();
```

L'invocazione del metodo *rifornisci* provoca la modifica della proprietà *serbatoio* che passa dal valore zero, assegnato al momento della creazione dell'oggetto, al valore *50*. L'invocazione del metodo *descrivi* restituisce la stringa *Ford Focus Grigio 1.8*.

12.2 Array

In JavaScript gli array sono oggetti predefiniti. Come per gli altri oggetti è possibile creare un nuovo array usando il costruttore *Array*, i cui argomenti specificano il numero e il valore degli elementi dell'array. Più precisamente, sono previsti i seguenti casi:

- senza parametri: il costruttore crea un array vuoto, un array che contiene zero elementi;
- con un solo parametro di tipo numerico (un intero maggiore o uguale a zero): il costruttore crea un array con un numero di elementi pari al valore dell'argomento numerico; ogni elemento creato è inizializzato al valore *undefined*;
- con un solo parametro di tipo numerico (un intero minore di zero o un numero reale): il costruttore segnala il verificarsi di un errore;

- con un solo parametro di tipo non numerico: il costruttore crea un array con un solo elemento il cui valore è proprio quello del parametro;
- con più parametri: il costruttore crea un array con tanti elementi quanti sono i parametri; agli elementi dell'array sono assegnati i valori passati come parametri nell'ordine in cui sono trasmessi; all'elemento di indice zero (il primo) è assegnato il valore del primo parametro, all'elemento di indice uno il valore del secondo parametro e così via.

Vediamo alcuni esempi di dichiarazione di variabili inizializzate con un array.

```
var a1 = new Array();
var a2 = new Array(3);
var a3 = new Array(3.5);
var a4 = new Array(true);
var a5 = new Array('Mario');
var a6 = new Array(10, 20, 30, 40);
```

Nel primo caso si ottiene un array vuoto. Nel secondo, si ottiene un array di tre elementi, tutti inizializzati con il valore *undefined*. Nel terzo caso si ottiene un errore perché, se il costruttore è invocato con un solo parametro numerico, questo deve avere come valore un intero maggiore o uguale a zero. Nel quarto e quinto caso si ottengono, rispettivamente, un array con un solo elemento il cui valore è *true* e un array con un solo elemento contenente la stringa *Mario*. Nell'ultimo caso si ottiene un array di quattro elementi con i seguenti valori:

- l'elemento di indice 0 ha come valore l'intero *10*;
- l'elemento di indice *1* ha come valore l'intero *20*;
- l'elemento di indice *2* ha come valore l'intero *30*;

l'ultimo elemento, di indice *3*, ha come valore l'intero *40*.

12.3 Stringhe

In JavaScript le stringhe sono oggetti predefiniti e, come tali, hanno proprietà e metodi che permettono di manipolarle. Come per gli ar-

ray, le stringhe hanno la proprietà *length*, il cui valore è pari al numero di caratteri di una stringa.

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

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.

12.4 Il convertitore di valuta

Un *convertitore di valuta* permette di calcolare il valore di un importo di denaro espresso in una valuta (ad esempio in Euro) nel corrispondente importo espresso in un'altra valuta (ad esempio in Lire). Nel seguito mostriamo il codice JavaScript di un costruttore personalizzato per oggetti che rappresentano *convertitori di valuta*.

L'oggetto ha i seguenti metodi:

- *imposta (valuta, fattore)*: imposta la sigla della *valuta* e il *fattore* di conversione da €;
- *converti (importo)*: converte l'*importo*, utilizzando il fattore di conversione, e restituisce il valore calcolato preceduto dalla sigla della valuta.

```
function Convertitore () {
   this.valutaCorrente;
   this.fattoreCorrente;
   this.imposta =
      function (valuta, fattore) {
         this.valutaCorrente = valuta;
         this.fattoreCorrente = fattore;
      }
   this.converti =
      function (importo) {
         return this.valutaCorrente + " " +
                importo * this.fattoreCorrente;
      }
}
```

Per verificare il corretto funzionamento dell'oggetto, definiamo una funzione che:

- crea un convertitore di valuta,
- imposta la valuta in lire con un fattore di conversione pari a 1936.27,
- converte 100.00 €.

```
function foo() {
   var c = new Convertitore();
   c.imposta("Lira", 1936.27);
   print(c.converti(100.00));
}
```

12.5 Insiemi

Un *insieme* [Wikipedia, alla voce *Insieme*] è una collezione di oggetti chiamati *elementi dell'insieme*. Il concetto di insieme è molto intuitivo, caratterizzato dalle seguenti proprietà:

- un elemento può appartenere o non appartenere a un insieme;
- un elemento può comparire solo una volta in un insieme;
- gli elementi di un insieme non hanno un ordine di comparizione;
- due insiemi coincidono se e solo se hanno gli stessi elementi.

La creazione di un oggetto che rappresenta un insieme vuoto avviene mediante l'invocazione del costruttore *Insieme*.

Gli elementi di un insieme sono memorizzati in una proprietà dell'oggetto, un array associativo chiamato *elementi*, inizializzata con un array vuoto al momento della sua creazione.

```
this.elementi = {};
```

L'oggetto *Insieme* ha i seguenti metodi: *cardinalità*, *appartiene*, *aggiungi*, *intersezione*, *unione*, *visualizza*.

Il metodo *cardinalità* restituisce il numero di elementi dell'insieme. Il ciclo che conta gli elementi è necessario perché la proprietà *length* non è definita per gli array associativi.

```
this.cardinalità =
   function() {
      var c = 0;
      for (var i in this.elementi) {
         c++;
      }
      return c;
   }
```

Il metodo *appartiene* è un predicato che verifica se un elemento appartiene a un insieme.

```
this.appartiene =
   function(x) {
      return (x in this.elementi);
   }
```

Il metodo *aggiungi* estende l'insieme con un elemento. Il metodo sfrutta le caratteristiche degli array associativi.

```
this.aggiungi =
   function(k) {
      this.elementi[k] = k;
   }
```

Il metodo *intersezione* ha un parametro (un altro insieme) e restituisce un insieme formato dagli elementi in comune ai due insiemi.

```
this.intersezione =
   function(x) {
      var n = new Insieme();
      for (var i in this.elementi) {
         if (x.appartiene(i)) {
             n.aggiungi(i);
         }
      }
      return n;
   }
```

Il metodo *unione* ha un parametro (un altro insieme) e restituisce un insieme formato dagli elementi dei due insiemi. Un elemento comune ai due insiemi compare solo una volta nel nuovo insieme.

```
this.unione =
   function(x) {
      var n = new Insieme();
      for (var i in this.elementi){
         n.aggiungi(i);
      }
      for (var j in x.elementi){
         n.aggiungi(j);
      }
      return n;
   }
```

Il metodo *visualizza* restituisce una stringa formata dagli elementi dell'insieme separati da virgole. Conformemente alla notazione standard degli insiemi, la stringa è delimitata da parentesi graffe.

```
this.visualizza =
   function() {
      var s = "{";
      var b = true;
      for (var i in this.elementi) {
         if (b) {
            s += i;
            b = false;
         } else {
            s += ", " + i;
         }
      }
      s += "}";
      return s;
   }
}
```

La seguente funzione

- crea l'insieme *alfa*
- aggiunge due elementi ad *alfa*
- calcola la cardinalità di *alfa*
- visualizza *alfa*
- verifica l'appartenenza di un numero ad *alfa*
- crea l'insieme *beta*
- aggiunge due elementi a *beta*
- calcola l'intersezione tra *alfa* e *beta*
- calcola l'unione tra *alfa* e *beta*.

```
function foo() {
   var alfa = new Insieme();
   alfa.aggiungi(10);
   alfa.aggiungi(20);
   print(alfa.cardinalità());
   print(alfa.visualizza());
   var x = 10;
   if (alfa.appartiene(x)) {
      print (x + " appartiene ad alfa");
```

```
   } else {
      print (x + " non appartiene ad alfa");
   }
   var beta = new Insieme();
   beta.aggiungi(10);
   beta.aggiungi(30);
   print(alfa.intersezione(beta).visualizza());
   print(alfa.unione(beta).visualizza());
}
```

12.6 Tabelle

Una *tabella* è un insieme di *coppie* (*chiave*, *valore*). Le tabelle si utilizzano per memorizzare l'informazione (valore) associate a una chiave. In una tabella una chiave può comparire solo una volta in una coppia.

Per realizzare una tabella si definisce un oggetto, *Tabella*, che ha una proprietà (*mappa*) e i seguenti metodi: *appartiene*, *aggiungi*, *valore*, *visualizza*. Le coppie di una tabella sono memorizzate nella proprietà *mappa*, un array associativo, inizializzata con un array vuoto al momento della sua creazione.

```
this.mappa = {};
```

Il metodo *appartiene* è un predicato che verifica se nella tabella esiste una coppia corrispondente alla chiave *c*.

```
this.appartiene =
      function (c) {
         return (c in this.mappa);
      }
```

Il metodo *aggiungi* ha due parametri: una chiave *c* e un valore *v*. Se la chiave *c* non è presente nella tabella il metodo aggiunge una nuova coppia, altrimenti sostituisce il vecchio valore della coppia con il nuovo valore *v*.

```
this.aggiungi =
   function (c, v) {
```

```
      this.mappa[c] = v;
   }
```

Il metodo *valore* ha un parametro, la chiave *c* da cercare nella tabella. Il metodo restituisce il valore associato alla chiave, se questa è presente nella tabella, *undefined* altrimenti.

```
this.valore =
   function (c) {
      return this.mappa[c];
   }
```

Il metodo *visualizza* restituisce una stringa che rappresenta il contenuto della tabella. Ogni coppia è rappresentata da una chiave seguita dal segno = e dal valore della chiave delimitato da doppi apici.

```
this.visualizza =
   function () {
      var s = '';
      for (var i in this.mappa) {
         s += ' ' + i + '="' + this.mappa[i] + '"';
      }
      return s;
   }
```

La seguente funzione

- crea una tabella
- aggiunge alcune coppie
- cerca il valore delle coppie
- visualizza la tabella.

```
function foo() {
   var t = new Tabella();
   var c1 = "pippo";
   t.aggiungi(c1, 12);
   if (t.appartiene(c1)) {
      print (c1 + '="' + t.valore(c1) + '"');
   } else {
      print (c1 + ' non è in tabella');
   }
```

```
   var c2 = "pluto";
   t.aggiungi(c2, 13);
   if (t.appartiene(c2)) {
      print (c2 + '="' + t.valore(c2) + '"');
   } else {
      print (c2 + ' non è in tabella');
   }
   var c3 = "paperino";
   if (t.appartiene(c3)) {
      print (c3 + '="' + t.valore(c3) + '"');
   } else {
      print (c3 + ' non è in tabella');
   }
   print(t.visualizza());
   t.aggiungi(c2, 3);
   if (t.appartiene(c2)) {
      print (c2 + '="' + t.valore(c2) + '"');
   } else {
      print (c2 + ' non è in tabella');
   }
   print(t.visualizza());
}
```

12.7 Esercizi

1. Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano *sveglie*.

   L'oggetto ha i seguenti metodi:

   - *oraCorrente* (*h*, *m*): imposta l'ora corrente. Se *h* è maggiore di 23 o *m* è maggiore di 59, i rispettivi valori sono impostati a zero;
   - *allarme* (*h*, *m*): imposta l'allarme. Se *h* è maggiore di 23 o *m* è maggiore di 59, i rispettivi valori sono impostati a zero;
   - *tic* (): fa avanzare di uno i minuti dell'ora corrente. Se i minuti arrivano a 60, azzera i minuti e fa avanzare di uno le ore. Se le ore arrivano a 24, azzera le ore. Se l'ora corrente è uguale all'allarme impostato, restituisce il valore *true*, *false* altrimenti.

Definire una funzione che

- crea una sveglia,
- imposta l'ora corrente alle 13:00,
- imposta l'allarme alle 13:02,
- fa avanzare l'ora corrente di due minuti.

2. Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano *distributori automatici di caffè*.

   L'oggetto ha i seguenti metodi:

   - *carica* (*n*): aggiunge *n* capsule per erogare *n* caffè;
   - *eroga* (*n*, *c*): verifica se è possibile erogare *n* caffè e, in caso positivo, li eroga addebitandoli al codice *c*;
   - *rapporto* (*c*): restituisce il numero di caffè addebitati al codice *c* e il numero di capsule disponibili.

   Definire una funzione che:

   - crea un distributore automatico di caffè,
   - carica 20 capsule,
   - eroga 12 caffè per il codice *Carlo*,
   - genera il rapporto per il codice *Carlo*.

13 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)*]. In questo capitolo presentiamo alcuni oggetti per la rappresentazione e la manipolazione di alberi.

13.1 Alberi binari

Un *albero binario* è un albero i cui nodi possono avere al massimo due *sotto-alberi* o *figli*. Un nodo senza figli è anche chiamato *foglia*. Un nodo che non ha un *nodo padre* è chiamato *radice*.

In JavaScript è possibile realizzare gli alberi binari mediante l'oggetto *AlberoBinario*. Il suo costruttore ha tre parametri: un valore *v*, un albero sinistro *sx*, un albero destro *dx*. Per memorizzare questi valori si utilizzano le proprietà *valore*, *sinistro* e *destro*.

```
this.valore   = v;
this.sinistro = sx;
this.destro   = dx;
```

La seguente funzione crea un albero binario formato da una radice e da due foglie. La creazione dell'albero inizia dalle foglie e procede verso la radice. Il valore *null* è utilizzato per indicare l'assenza di un figlio in un albero.

```
function foo () {
   var foglia1 = new AlberoBinario(2, null, null);
   var foglia2 = new AlberoBinario(3, null, null);
   var radice = new AlberoBinario(1, foglia1, foglia2);
}
```

Un albero può essere visualizzato mostrando i suoi valori in sequenza. La costruzione della sequenza dei valori è chiamata *visita* e, nel caso degli alberi binari, può essere effettuata 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 visita anticipata del figlio sinistro e dalla visita anticipata del figlio destro. La visita anticipata dell'albero binario creato nell'esempio precedente è *1 2 3*.

Il metodo *visitaAnticipata* costruisce una stringa che rappresenta la visita anticipata di un albero binario.

```
this.visitaAnticipata =
   function() {
      var vs;
      if (this.sinistro != null) {
         vs = this.sinistro.visitaAnticipata();
      } else {
         vs = "";
      }
      var vd;
      if (this.destro != null) {
         vd = this.destro.visitaAnticipata();
      } else {
         vd = "";
      }
      return this.valore + " " + vs + vd;
}
```

La visita differita costruisce una sequenza in cui prima compaiono la visita anticipata del figlio sinistro e dalla visita anticipata del figlio destro e, alla fine, il valore della radice. La visita differita dell'albero binario creato nell'esempio precedente è *2 3 1*.

Il metodo *visitaDifferita* costruisce una stringa che rappresenta la visita differita di un albero binario.

```
this.visitaDifferita =
   function() {
      var vs;
      if (this.sinistro != null) {
```

```
         vs = this.sinistro.visitaDifferita();
      } else {
         vs = "";
      }
      var vd;
      if (this.destro != null) {
         vd = this.destro.visitaDifferita();
      } else {
         vd = "";
      }
      return vs + vd + " " + this.valore;
   }
```

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 creato nell'esempio precedente è *2 1 3*.

Il metodo *visitaSimmetrica* costruisce una stringa che rappresenta la visita simmetrica di un albero binario.

```
this.visitaSimmetrica =
   function() {
      var vs;
      if (this.sinistro != null) {
         vs = this.sinistro.visitaSimmetrica();
      } else {
         vs = "";
      }
      var vd;
      if (this.destro != null) {
         vd = this.destro.visitaSimmetrica();
      } else {
         vd = "";
      }
      return vs + " " + this.valore + vd;
   }
```

L'*altezza* di un albero è definita come la lunghezza del massimo cammino dalla radice a una foglia. In base a questa definizione, l'altezza

di un albero formato solo da un nodo radice è uguale a zero. L'altezza dell'albero binario creato nell'esempio precedente è *1*.

Il metodo *altezza* restituisce l'altezza di un albero binario.

```
this.altezza =
   function() {
      if (this.sinistro == null &&
          this.destro == null) {
         return 0;
      }
      var as;
      if (this.sinistro != null) {
         as = this.sinistro.altezza();
      } else {
         as = 0;
      }
      var ad;
      if (this.destro != null) {
         ad = this.destro.altezza();
      } else {
         ad = 0;
      }
      if (as > ad) {
          return as + 1;
      } else {
          return ad + 1;
      }
   }
```

La *frontiera* di un albero è la sequenza dei valori delle foglie visitate da sinistra verso destra. La frontiera dell'albero binario creato nell'esempio precedente è *2 3*.

Il metodo *frontiera* restituisce la frontiera di un albero binario.

```
this.frontiera =
   function() {
      if (this.sinistro == null &&
          this.destro == null) {
         return this.valore + " ";
      }
      var fs;
```

```
      if (this.sinistro != null) {
         fs = this.sinistro.frontiera();
      } else {
         fs = "";
      }
      var fd;
      if (this.destro != null) {
         fd = this.destro.frontiera();
      } else {
         fd = "";
      }
      return fs + fd;
   }
```

La seguente funzione

- crea un albero binario *alfa*
- effettua la visita anticipata di *alfa*
- effettua la visita differita di *alfa*
- effettua la visita simmetrica di *alfa*
- calcola l'altezza di *alfa*
- costruisce la frontiera di *alfa*.

```
function foo() {
   var alfa = new AlberoBinario(1,
                  new AlberoBinario(2,
                     new AlberoBinario(4, null, null),
                     new AlberoBinario(5, null, null)),
                  new AlberoBinario(3, null, null));
   print(alfa.visitaAnticipata());
   print(alfa.visitaDifferita());
   print(alfa.visitaSimmetrica());
   print(alfa.altezza());
   print(alfa.frontiera());
}
```

13.2 Alberi di ricerca

Un *albero di ricerca* è un albero binario la cui visita simmetrica genera una sequenza ordinata di valori. Ci possono essere due tipi di albe-

ri di ricerca: con ripetizioni o senza ripetizioni. Nel primo caso l'ordinamento prevede anche l'uguaglianza (ad esempio non decrescente), nel secondo caso l'uguaglianza è esclusa (ad esempio crescente).

La ricerca di un valore in un albero di ricerca è molto efficiente: partendo dalla radice è possibile escludere dalla ricerca metà dell'albero, perché il valore cercato non può appartenere a quella metà. Anche l'inserzione di un valore in un albero di ricerca è un'operazione efficiente, perché si basa su una ricerca e sull'eventuale aggiunta di una nuova foglia.

Il costruttore dell'oggetto *AlberoDiRicerca* è simile a quello dell'oggetto *AlberoBinario* in quanto ha un parametro e tre proprietà: valore, figlio sinistro, figlio destro.

Il metodo *aggiungi* ha un parametro *k* e aggiunge un nodo all'albero di ricerca, mantenendo l'ordinamento crescente. Il metodo è definito ricorsivamente.

```
this.aggiungi =
   function (k) {
      if (k < this.valore) {
         if (this.sinistro == null) {
            this.sinistro = new AlberoDiRicerca(k);
         } else {
            this.sinistro.aggiungi(k);
         }
      }
      if (k > this.valore) {
         if (this.destro == null) {
            this.destro = new AlberoDiRicerca (k);
         } else {
            this.destro.aggiungi(k);
         }
      }
   }
```

Il metodo *cerca* è un predicato con un parametro *k* che verifica se esiste un nodo il cui valore è uguale a *k*. Anche questo metodo è definito ricorsivamente.

```
this.cerca =
   function (k) {
      if (k == this.valore) {
         return true;
      }
      if (k < this.valore) {
         if (this.sinistro == null) {
            return false;
         } else {
            return this.sinistro.cerca(k);
         }
      }
      if (k > this.valore) {
         if (this.destro == null) {
            return false;
         } else {
            return this.destro.cerca(k);
         }
      }
   }
```

Infine, il metodo *visita* restituisce una stringa ottenuta visitando l'albero con una visita simmetrica. Questo metodo è uguale al metodo *visitaSimmetrica* definito per gli alberi binari.

La seguente funzione

- crea l'albero di ricerca *alfa* con il valore 1
- aggiunge *5*, *3* ed *8* ad *alfa*
- effettua una visita simmetrica di *alfa*
- cerca *1* e *4* in *alfa*.

```
function foo() {
   var a = new AlberoDiRicerca(1);
   a.aggiungi(5);
   a.aggiungi(3);
   a.aggiungi(8);
   print(a.visita());
   print(a.cerca(1));
   print(a.cerca(4));
}
```

13.3 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. Un'altra differenza con gli alberi binari consiste nel fatto che sono definite solo due visite: anticipata e differita. La visita simmetrica, infatti, non avrebbe senso.

Come per gli alberi binari, i nodi di un albero n-ario hanno un valore associato. Il costruttore *AlberoNArio* ha un solo parametro *v,* memorizzato nella proprietà *valore*. I figli di un albero n-ario sono memorizzati nella proprietà *figli*, inizializzata con un array vuoto.

```
this.valore = v;
this.figli = [];
```

Il metodo *aggiungiFiglio* ha un parametro *n*, un nodo da aggiungere in coda ai figli di un albero n-ario.

```
this.aggiungiFiglio =
   function (n) {
      this.figli.push(n);
   }
```

La visita anticipata di un albero n-ario è calcolata dal metodo *visitaAnticipata* che si comporta analogamente allo stesso metodo definito per gli alberi binari. La scansione dei figli è effettuata mediante un'iterazione determinata.

```
this.visitaAnticipata =
   function() {
      var visita = "";
      for (var i in this.figli) {
         visita += this.figli[i].visitaAnticipata() + " ";
      }
      return this.valore + " " + visita;
   }
```

Anche la visita differita, calcolata dal metodo *visitaDifferita*, è simile all'analoga visita definita per gli alberi binari.

```
this.visitaDifferita =
   function() {
      var visita = "";
      for (var i in this.figli) {
        visita += this.figli[i].visitaDifferita() + " ";
      }
      return visita + this.valore + " ";
   }
```

L'altezza di un albero n-ario è calcolata dal metodo *altezza*, che usa l'algoritmo per il calcolo del massimo, già definito per gli array.

```
this.altezza =
   function() {
      if (this.figli.length == 0) {
         return 0;
      }
      var maxAltezza = this.figli[0].altezza();
      for (var i = 1; i < this.figli.length; i++) {
         var altezzaFiglio = this.figli[i].altezza();
         if (altezzaFiglio > maxAltezza) {
            maxAltezza = altezzaFiglio;
         }
      }
      return maxAltezza + 1;
   }
```

L'ultimo metodo, *frontiera*, calcola la frontiera di un albero n-ario. Anche questo metodo è simile a quello definito per gli alberi binari.

```
this.frontiera =
   function() {
      if (this.figli.length == 0) {
         return this.valore + " ";
      }
      var f = "";
      for (var i in this.figli) {
         f += this.figli[i].frontiera();
      }
      return f;
   }
```

La seguente funzione

- crea l'albero n-ario *a1* composto da sette nodi
- effettua la visita anticipata e differita di *a1*
- calcola l'altezza e la frontiera di *a1*.

```
function foo() {
   var a1 = new AlberoNArio(1);
   var a2 = new AlberoNArio(2);
   var a3 = new AlberoNArio(3);
   var a4 = new AlberoNArio(4);
   var a5 = new AlberoNArio(5);
   var a6 = new AlberoNArio(6);
   var a7 = new AlberoNArio(7);
   a1.aggiungiFiglio(a2);
   a1.aggiungiFiglio(a3);
   a1.aggiungiFiglio(a4);
   a2.aggiungiFiglio(a5);
   a2.aggiungiFiglio(a6);
   a3.aggiungiFiglio(a7);
   print(a1.visitaAnticipata());
   print(a1.visitaDifferita());
   print(a1.frontiera());
   print(a1.altezza());
}
```

13.4 Alberi n-ari con attributi

Un albero n-ario può essere esteso associando uno o più attributi ad ogni nodo. Si ottiene così un *albero n-ario con attributi*. Il costruttore *AlberoNArioAttr* differisce dal costruttore *AlberoNArio* per la presenza di una proprietà *attributi* il cui valore è una tabella, inizialmente vuota. Le proprietà del costruttore sono tre: *valore*, *figli*, *attributi*.

```
this.valore = v;
this.figli = [];
this.attributi = new Tabella();
```

L'unico metodo nuovo da definire è *aggiungiAttributo*, con due parametri: il nome dell'attributo *n*, il valore dell'attributo *v*.

```
this.aggiungiAttributo =
   function (n, v) {
      this.attributi.aggiungi(n, v);
   }
```

La seguente funzione

- crea l'albero n-ario con attributi *a1* composto da due nodi
- aggiunge l'attributo *x* associandogli il valore *100*.

```
function foo() {
   var a1 = new AlberoNArioAttr(1);
   var a2 = new AlberoNArioAttr(2);
   var a3 = new AlberoNArioAttr(3);
   a1.aggiungiFiglio(a2);
   a1.aggiungiFiglio(a3);
   a1.aggiungiAttributo("x", 100);
}
```

13.5 Esercizi

1. Un'*espressione aritmetica* può essere rappresentata mediante un albero binario i cui nodi sono *operatori* (addizione, sottrazione, moltiplicazione, divisione) e le foglie sono valori numerici.

   Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano espressioni aritmetiche.

   L'oggetto ha i seguenti metodi:

   - *calcola* (): restituisce il valore dell'espressione,
   - *visualizza* (): restituisce una stringa che rappresenta l'espressione.

   Definire una funzione che

   - crea l'espressione *2 * 3 + 5 – 6 / 2 + 1*
   - stampa la rappresentazione dell'espressione
   - calcola e stampa il valore dell'espressione.
2. L'*albero genealogico* è una rappresentazione (parziale) che mostra i rapporti familiari tra gli antenati di un individuo. Abi-

tualmente un albero genealogico è realizzato utilizzando delle caselle, quadrate per i maschi e circolari per le femmine, contenenti i nomi di ciascuna persona, spesso corredate di informazioni aggiuntive, quali luogo e data di nascita e morte, occupazione o professione. Questi simboli, disposti dall'alto verso il basso in ordine cronologico, sono connessi da vari tipi di linee che rappresentano matrimoni, unioni extra coniugali, discendenza [Wikipedia, alla voce *Albero genealogic*o].

Un *albero genealogico patrilineare* è un particolare tipo di albero genealogico in cui sono rappresentati tutti e soli i discendenti per via paterna.

Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano alberi genealogici patrilineari.

Il costruttore ha un parametro: *persona*.

L'oggetto ha i seguenti metodi:

- *aggiungiConiuge* (*persona*, *coniuge*): aggiunge, nell'albero genealogico, il *coniuge* a una *persona*;
- *aggiungiDiscendente* (*persona*, *discendente*): aggiunge, nell'albero genealogico, un *discendente* diretto a una *persona*;
- *visualizzaPersona* (*persona*): restituisce una stringa contenente le informazioni relative a una *persona*;
- *grado*(*persona*): calcola il grado di parentela tra una *persona* e il suo antenato più anziano nell'albero genealogico;
- *linea* (*persona*): restituisce una stringa contenente la linea di discendenza tra una *persona* e il suo antenato più anziano nell'albero genealogico.

Definire una funzione che

- crea un albero genealogico con queste caratteristiche: *Bruno*, *Carlo* e *Daniela* sono figli di *Aldo*, *Enzo* e *Fabio* sono figli di *Bruno*, *Giacomo* è figlio di *Carlo*, *Alessia* è la moglie di *Aldo*, *Beatrice* è la moglie di *Bruno*, *Cecilia* è la moglie di *Carlo*;

- visualizza le informazioni di *Aldo* ed *Enzo*;
- calcola il grado di parentela di *Bruno*, *Giacomo* e *Fabio*;
- visualizza la linea di discendenza di *Enzo* e *Giacomo*.

14 HTML

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

14.1 Marche

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. Una marca è un elemento inserito nel testo di un documento per indicarne la struttura o la formattazione. 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 del 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 del tag stesso. 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 HTML è formato da un testo racchiuso dalla marca *html* e si divide in due parti: un'*intestazione* e un *corpo*. Nell'intestazione, racchiusa dalla marca *head*, è riportato il *titolo* del documento, mostrato sulla *barra del browser* quando il documento è visualizzato come una pagina. Il corpo rappresenta il documento vero e proprio e contiene sia il testo da visualizzare nella pagina, sia le marche di formattazione. Il corpo del documento è racchiuso dalla marca *body*.

Prima della marca *html* può esserci un'ulteriore marca che riporta alcune informazioni sul tipo di documento e sulla versione di HTML utilizzata per descriverlo. Non tratteremo questa marca, ma è importante sapere della sua esistenza perché alcuni strumenti per la scrittura di documenti HTML la inseriscono automaticamente. Ecco un esempio di documento HTML:

```
<html>
  <head>
  <title>Titolo del documento</title>
  </head>
  <body>
  Contenuto del documento
  </body>
</html>
```

14.2 Eventi

Un *evento* è qualcosa che accade in un documento HTML in seguito a un'azione dell'utente che lo sta visualizzando mediante un browser. Poiché il numero degli eventi è grande e in continua crescita, conviene raggrupparli in base ad alcuni aspetti comuni:

- eventi attivati dai tasti del mouse,
- eventi attivati dai movimenti del mouse,
- eventi attivati dal trascinamento del mouse,
- eventi attivati dalla tastiera,
- eventi attivati dalle modifiche della pagina,
- eventi legati al *focus*,
- eventi attivati dal caricamento degli oggetti,

- eventi attivati dai movimenti delle finestre,
- eventi legati a particolari bottoni.

Nel seguito presentiamo una parte degli eventi, quelli più comuni. Il lettore interessato può trovare l'elenco completo degli eventi consultando la letteratura su HTML.

Gli eventi attivati dai tasti del mouse sono:

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

Gli eventi *mouseDown* e *mouseUp* sono attivati 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, attiva in successione *mouseDown*, *mouseUp*, *click*.

Gli eventi attivati dai movimenti del mouse sono:

- *mouseOver*: attivato quando il mouse si muove su un oggetto;
- *mouseOut*: attivato quando il mouse si sposta da un oggetto;

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

Gli eventi attivati dalla tastiera sono:

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

Gli eventi legati al caricamento degli oggetti sono:

- *load*: evento attivato quando si carica un oggetto, ad esempio una pagina o un'immagine;
- *unload*: è l'opposto del precedente ed è attivato quando si lascia una finestra per caricarne un'altra o anche per ricaricare la stessa (col tasto *refresh*).

14.3 Gestione degli eventi

Quando un evento si verifica il browser determina l'azione da compiere in risposta. La situazione più comune si verifica con l'evento *click*: l'utente punta un elemento della pagina e clicca il pulsante sinistro; il browser riconosce l'elemento puntato dal mouse nel codice HTML e verifica se tale elemento può "sentire" l'evento *click*. In caso affermativo, il browser esegue il codice JavaScript associato all'evento.

Per far sì che un elemento HTML possa sentire un evento è necessario definire un gestore dell'evento, come attributo della marca di apertura dell'elemento. L'attributo ha un nome determinato convenzionalmente dal nome dell'evento preceduto da *on*. Ad esempio, l'attributo per l'evento *click* si chiama *onclick*. Il valore dell'attributo è una stringa che rappresenta un frammento di codice JavaScript, che sarà eseguito quando si verificherà l'evento sull'elemento stesso.

Consideriamo un semplice documento HTML che contiene una lista di tre elementi:

- italiano
- inglese
- messaggio.

Quando l'utente clicca sulla parola *messaggio* il browser apre una *finestra di alert* con il messaggio *Ciao mondo!*. All'elemento corrispondente alla parola messaggio (in questo caso la marca *li*) è stato associato l'attributo *onclick* e un frammento di programma JavaScript che apre la finestra di *alert*. Il codice è memorizzato nel file *CiaoMondo.html*.

```
<html>
  <head>
  <title>Gestione evento click</title>
  </head>
  <body>
  <ul>
    <li>italiano</li>
    <li>english</li>
    <li onclick="alert('Ciao mondo!');">messaggio</li>
  </ul>
  </body>
</html>
```

14.4 Script

Un programma JavaScript può essere inserito in un documento HTML tramite la marca *script*, che può comparire sia nell'intestazione (la parte del documento racchiusa dalla marca *head*), sia nel corpo del documento. In questo libro useremo la marca *script* solo nell'intestazione. Tra i vari attributi della marca *script*, quelli che ci interessano direttamente sono due: *type*, che definisce il tipo di script, e *src*, descritto nel seguito. Per usare JavaScript all'interno di un documento HTML è necessario assegnare il valore *text/javascript* all'attributo *type*.

Per inserire un programma JavaScript in un documento HTML si possono usare due tecniche:

- scrivere il programma tra la marca *script* di apertura e quella di chiusura;
- scrivere il programma in un file separato e indicare il nome del file come valore dell'attributo *src*.

Gli esempi presentati in questo libro usano la seconda tecnica. Ciò permette di organizzare i programmi JavaScript in *librerie*, ovvero file contenenti programmi utilizzati frequentemente, scritti una volta sola e usati in più occasioni. Alcuni dei vantaggi legati all'utilizzo delle librerie sono:

- un programma contenuto in una libreria non è mescolato alle marche ed è più semplice da leggere;
- un documento HTML è più leggibile perché non contiene programmi JavaScript;
- la manutenzione di un programma contenuto in una libreria è semplificata perché le modifiche al programma possono essere effettuate una sola volta nella libreria, per riflettersi automaticamente sui documenti che includono il programma.

Un file che contiene un programma JavaScript ha, per convenzione, l'estensione ".js". Il file deve contenere esclusivamente un programma JavaScript perché la presenza di altri elementi, come ad esempio marche HTML, provocherebbe un errore e farebbe sì che il suo contenuto sia ignorato.

14.5 Caricamento di una pagina

Quando il browser termina il caricamento di una pagina HTML viene generato l'evento *load*. Questo evento può essere associato all'oggetto *window*, che rappresenta la finestra in cui compare la pagina appena caricata. Questa associazione viene effettuata nel programma JavaScript associato al documento HTML mediante la marca *script*. In pratica, la proprietà *onload* di *window* viene associata al nome di una funzione senza parametri che sarà invocata quando si verificherà l'evento *load*.

Riprendiamo l'esempio visto in precedenza e definiamo un programma JavaScript che gestisce l'evento *load*. La funzione associata si chiama *inizializza*. Modifichiamo il comportamento dinamico della pagina aggiungendo la gestione dell'evento click agli elementi corrispondenti alle parole *italiano* e *english*. Vogliamo far sì che dopo aver cliccato sulla parola *italiano* la stringa che comparirà nella finestra di *alert* associata a *messaggio* sia in italiano (*Ciao mondo!*) mentre dopo aver cliccato sulla parola *english* la stringa sia in inglese (*Hello world!*). Per ottenere questo comportamento definiamo una variabile globale *linguaggio*, che può avere come valore la stringa *italiano* o la stringa *english*. Per rendere più leggibile il codice definiamo del programma JavaScript due funzioni, *inItaliano* e *inEnglish*, che invo-

chiamo nei gestori dell'evento *click* delle parole *italiano* e *english*. Definiamo una terza funzione, *messaggio*, che invochiamo nel gestore dell'evento *click* della parola *messaggio*.

Il codice HTML della pagina è il seguente.

```
<html>
  <head>
  <title>Gestione evento click</title>
  <script type="text/javascript"
          src="CiaoMondo.js">
  </script>
  </head>
  <body>
  <ul>
    <li onclick="inItaliano();">italiano</li>
    <li onclick="inEnglish();">english</li>
    <li onclick="messaggio();">messaggio</li>
  </ul>
  </body>
</html>
```

Il programma JavaScript è memorizzato nel file *CiaoMondo.js*.

```
var linguaggio;
window.onload = inizializza;
function inizializza() {
   linguaggio = "italiano";
}
function inItaliano() {
   linguaggio = "italiano";
}
function inEnglish() {
   linguaggio = "english";
}
function messaggio() {
   if (linguaggio == "italiano") {
      alert("Ciao mondo!");
   }
   if (linguaggio == "english") {
      alert("Hello world!");
```

```
    }
}
```

14.6 Esercizi

1. Definire in HTML una pagina relativa a un ristorante. La pagina deve contenere una descrizione e un menu. Il menu deve essere diviso in più parti: antipasti, primi, secondi, contorni, dolci. Cliccando sul nome di un piatto deve comparire il suo prezzo. Passando il mouse sul nome del proprietario, indicato nella descrizione, deve comparire un breve messaggio. Un doppio clic sul nome del ristorante, indicato nella descrizione, deve far comparire un numero di telefono.

15 Document Object Model

Il *Document Object Model* (spesso abbreviato come *DOM*), letteralmente modello a oggetti del documento, è una forma di rappresentazione dei documenti strutturati. DOM è lo standard ufficiale del *World Wide Web Consortium* (spesso abbreviato come *W3C*) per la rappresentazione di documenti strutturati in maniera da essere neutrali sia per la lingua che per la piattaforma. DOM è inoltre la base per una vasta gamma delle interfacce di programmazione delle applicazioni [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. L'applicazione più importante del DOM la troviamo in Internet, per la rappresentazione e la visualizzazione delle pagine web. In pratica, un documento in formato XML (o HTML) è rappresentato mediante un albero DOM.

Per semplicità presenteremo solo alcuni aspetti del DOM, quelli che ci serviranno per illustrare i concetti di base della programmazione web. Il lettore interessato può trovare in linea una documentazione molto ampia per approfondire le sue conoscenze.

15.1 Struttura e proprietà

Un albero DOM 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 DOM 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 DOM. I nodi attributo definiscono gli attributi dei nodi elemento. La radice dell'albero DOM associato a una pagina HTML corrisponde all'oggetto *document*.

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

- *nodeType*
- *nodeName*
- *nodeValue*
- *childNodes*
- *attributes*.

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

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

Il nome di un nodo, definito dalla proprietà *nodeName*, ha un significato che dipende dal tipo del nodo. Il nome di un nodo elemento corrisponde alla marca del corrispondente documento XML. Il nome di un nodo attributo corrisponde al nome dell'attributo. Il nome di un nodo testo è predefinito ed è sempre *#text*, così come il nome di nodo documento è sempre *#document*.

Il valore di un nodo, definito dalla proprietà *nodeValue*, ha anch'esso un significato che dipende dal tipo del nodo. Il valore di un nodo elemento è *null*. Il valore di un nodo attributo è il valore dell'attributo Il valore di un nodo testo è il testo associato al nodo. Il valore di un nodo documento è *null*.

Il valore della proprietà *childNodes* è un array che contiene i figli di un nodo elemento o di un nodo documento. Se un nodo non ha fi-

gli, 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*.

15.2 Navigazione

Il modello DOM prevede diverse tecniche per visitare gli alberi DOM e per muoversi (navigare) tra i suoi nodi. Queste tecniche si basano sul valore della proprietà childNodes e di altre proprietà che collegano un nodo ai suoi nodi vicini:

- *firstChild*
- *lastChild*
- *firstSibling*
- *lastSibling*
- *previousSibling*
- *nextSibling*
- *parentNode*.

Queste proprietà, quando sono definite, hanno i seguenti valori: *firstChild* è il primo figlio di un nodo, *lastChild* è l'ultimo figlio di un nodo, *firstSibling* è il primo fratello di un nodo, *lastSibling* è l'ultimo fratello di un nodo, *previousSibling* è il fratello precedente di un nodo, *nextSibling* è il fratello successivo di un nodo, *parentNode* è il padre di un nodo. Se una di queste proprietà non è definita il suo valore è *null*.

La tecnica più comune per visitare i figli di un nodo si basa su un'iterazione determinata che fa uso della proprietà *childNodes*.

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

Un'altra tecnica di visita si basa su un'iterazione indeterminata e sulle proprietà *firstSibling* e *nextSibling*.

```
var nodoFiglio = n.firstSibling;
while (nodoFiglio != null) {
   nodoFiglio = nodoFiglio.nextSibling;
}
```

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

```
var nodoFiglio = n.lastSibling;
while (nodoFiglio != null) {
   nodoFiglio = nodoFiglio.previousSibling;
}
```

Un'altra tecnica permette di visitare un albero partendo da un nodo fino ad arrivare alla sua radice. In questo caso si usa la proprietà *parentNode*. Nell'esempio che segue la visita parte dalla radice, procede scegliendo il primo figlio e poi torna indietro alla radice.

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

15.3 Ricerca

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

- *getElementsByTagName*
- *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 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 DOM. L'iterazione determinata[10] scandisce la collezione così ottenuta.

```
var l = document.getElementsByTagName("br");
for (var i = 0; i < l.length; i++) {
   alert (l[i].nodeName);
}
```

Il metodo *getElementById* ha un unico argomento *v*, 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 restituisce il primo nodo il cui attributo *id* ha un valore uguale a *v*, *null* altrimenti.

Nell'esempio che segue il valore della variabile *n* è un nodo elemento il cui attributo *id* vale la stringa *alfa*. Se un tale nodo non è presente il valore della variabile *n* è *null*.

```
var n = document.getElementById("alfa");
if (n != null) {
   alert (n.nodeName);
}
```

15.4 Creazione e modifica

Per modificare un albero DOM si può procedere in due modi diversi: creare un nuovo nodo e aggiungerlo all'albero, eliminare un nodo esistente. La creazione di un nodo può essere effettuata 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

10 La scansione deve essere effettuata indicando esplicitamente la lunghezza della collezione.

nodo testo che si vuole creare. Questi due metodi devono essere invocati sull'oggetto *document*.

```
var n1 = document.createElement("br");
var n2 = document.createTextNode("alfa");
```

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

- *appendChild*
- *insertBefore*.

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 n1 = document.createElement("br");
n.appendChild(n1);
```

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 n2 = document.createTextNode("alfa");
n.insertBefore(n2, n.firstSibling);
```

La rimozione di un nodo, e di tutto l'albero di cui tale nodo è la radice, può essere effettuata 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.

```
n.removeChild(n.firstSibling);
```

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 n1 = document.createElement("br");
n.replaceChild(n1, n.firstChild);
```

15.5 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 = n.getAttribute("a");
```

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*.

```
n.setAttribute("a", 123);
n.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.

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

15.6 Eventi

Nel capitolo precedente abbiamo affrontato il problema della gestione degli eventi. Per reagire a un evento generato dalla pagina definiamo una funzione specifica. A volte è necessario conoscere l'elemento che ha generato un evento, per far sì che la funzione possa navigare a partire dal nodo relativo a quell'elemento. Questa informazione è disponibile utilizzando il valore dell'oggetto *this*. Ad esempio, supponiamo di aver definito una lista di elementi a cui abbiamo associato,

in HTML, un attributo *alfa* con valori distinti per ogni elemento. L'evento *onclick* è gestito dalla funzione *foo*.

```
<ol>
  <li alfa="1" onclick="foo();">Uno</li>
  <li alfa="2" onclick="foo();">Due</li>
  <li alfa="3" onclick="foo();">Tre</li>
</ol>
```

La funzione *foo* può accedere al valore dell'attributo *alfa* dell'elemento che è stato cliccato.

```
function foo() {
   alert(this.getAttribute("alfa"));
}
```

15.7 Proprietà innerHTML

La proprietà *innerHTML* non è definita nello standard *W3C* ma, di fatto, è gestita dalla maggior parte dei browser attualmente disponibili. Il valore di questa proprietà è una stringa che rappresenta il codice HTML corrispondente all'albero radicato nel nodo corrispondente. La proprietà può essere modificata, assegnando un valore che deve essere una stringa corrispondente a una sequenza corretta di marche HTML. L'effetto dell'assegnamento consiste nella modifica dinamica dell'albero DOM, con l'inserzione dell'albero corrispondente alla stringa HTML.

Un esempio di modifica della proprietà *innerHTML* è il seguente:

```
n.innerHTML = "<ul><li>alfa</li> <li>beta</li></ul>";
```

15.8 Generazione dinamica

Una pagina può contenere un modulo per la raccolta di informazioni, definito dalla marca *form*. All'interno del modulo sono definiti elementi per l'inserimento di dati, pulsanti per l'interazione con l'utente, menu e altri elementi grafici. La definizione di una pagina con queste caratteristiche richiede un attento lavoro di analisi e di progettazione grafica.

In alcuni casi è necessario generare dinamicamente questi elementi, a partire da informazioni che possono variare a seconda del contesto di uso della pagina. Ad esempio, le voci di un menu possono cambiare in seguito all'inserimento di un valore in un campo numerico. Ciò rende la pagina più semplice e più intuitiva da usare perché capace di adattarsi alle esigenze e alle aspettative dell'utente.

Per ottenere questo comportamento è necessario utilizzare appieno le potenzialità offerte da HTML e da JavaScript, combinandole insieme secondo schemi efficaci e sperimentati. Nel seguito definiremo alcuni problemi e presenteremo le relative soluzioni. Il trattamento di questa tematica non può essere esaustivo ma copre uno spettro abbastanza ampio di casi.

Supponiamo di dovere definire un menu formato da un numero limitato (ad esempio, inferiore a dieci) di voci. Supponiamo, inoltre che la scelta effettuata dall'utente debba essere convalidata dalla pressione di un pulsante. Conoscendo a priori le voci è possibile definire un modulo formato da un elemento menu e da un pulsante. Il codice HTML della pagina è il seguente.

```
<html>
<head>
<script type="text/javascript" src="Menu1.js"></script>
</head>
<body>
 <form>
  <select id="selezione">
   <option value="alfa">Uno</option>
   <option value="beta">Due</option>
   <option value="gamma">Tre</option>
  </select>
  <input type="button" value="Conferma"
         onclick="conferma();"/>
 </form>
</body>
</html>
```

Il programma JavaScript associato alla pagina è il seguente.

```
var nodoSelect;
function inizializza() {
  nodoSelect = document.getElementById("selezione");
}
function conferma() {
   alert(nodoSelect.value);
}
window.onload = inizializza;
```

Per rendere dinamica la generazione della pagina definiamo gli array *selezioni* e *valori*. L'array *selezioni* è inizializzato con le tre voci che compaiono nel menu (le stringhe *Uno*, *Due*, *Tre*). L'array *voci* è inizializzato con i valori associati alle voci del menu (le stringhe *alfa*, *beta*, *gamma*). Utilizzando questi array è possibile generare dinamicamente una stringa equivalente al codice che definisce l'elemento *select* e che compare nella pagina HTML. Assegnando la stringa alla proprietà *innerHTML* di un elemento HTML si ottiene dinamicamente lo stesso effetto ottenuto in maniera statica.

La pagina è così definita.

```
<html>
<head>
<script type="text/javascript" src="Menu2.js"></script>
</head>
<body>
 <form>
  <select id="selezione"></select>
  <input type="button" value="Conferma"
         onclick="conferma();"/>
 </form>
</body>
</html>
```

Il programma JavaScript associato alla pagina è il seguente.

```
var nodoSelect;
function inizializza() {
   var selezioni = ["Uno", "Due", "Tre"];
   var valori = ["alfa", "beta", "gamma"];
   var s = "";
```

```
   for (var i in selezioni) {
      s += '<option value="' + valori[i] + '">' +
           selezioni[i] + '</option>';
   }
   nodoSelect = document.getElementById("selezione");
   nodoSelect.innerHTML = s;
}
function conferma() {
   alert(nodoSelect.value);
}
window.onload = inizializza;
```

Come ultimo passaggio, eliminiamo anche il codice JavaScript associato all'attributo *onclick*, gestendo questa associazione nella funzione *inizializza*. Vedremo che questa tecnica permette di gestire in maniera efficace gli eventi di una pagina.

La pagina è così definita.

```
<html>
<head>
<script type="text/javascript" src="Menu3.js"></script>
</head>
<body>
 <form>
  <select id="selezione"></select>
  <input type="button" value="Conferma" id="conferma"/>
 </form>
</body>
</html>
```

Il programma JavaScript associato alla pagina è il seguente.

```
var nodoInput;
var nodoSelect;
function inizializza() {
   nodoInput = document.getElementById("conferma");
   nodoInput.onclick = conferma;
   var selezioni = ["Uno", "Due", "Tre"];
   var valori = ["alfa", "beta", "gamma"];
   var s = "";
   for (var i in selezioni) {
```

```
      s += '<option value="' + valori[i] + '">' +
           selezioni[i] + '</option>';
   }
   nodoSelect = document.getElementById("selezione");
   nodoSelect.innerHTML = s;
}
function conferma() {
   alert(nodoSelect.value);
}
window.onload = inizializza;
```

Un problema più complesso consiste nella generazione dinamica di una pagina che contiene un numero variabile di pulsanti ognuno dei quali è associato a una casella di testo. Quando l'utente clicca su un pulsante l'evento invoca una funzione che ha come argomento il valore della casella di testo associata al pulsante premuto.

Come per il problema precedente, definiamo una pagina completamente statica, che contiene due pulsanti e due caselle di testo. La pagina è così definita.

```
<html>
<head>
<script type="text/javascript" src="Pulsanti1.js"></script>
</head>
<body><form>
 <input type="text" name="testo1">
 <input type="button" value="Uno"
        onclick="pulsante1(testo1.value);"/>
 <br/>
 <input type="text" name="testo2">
 <input type="button" value="Due"
        onclick="pulsante2(testo2.value);"/>
</form></body>
</html>
```

Il programma JavaScript associato è il seguente.

```
function pulsante1(s) {
   alert(s);
}
function pulsante2(s) {
```

```
   alert(s);
}
```

Per rendere dinamica la generazione della pagina definiamo un array, *valori*, che contiene le stringhe che definiscono l'attributo *value* dell'elemento *input* che definisce un pulsante. L'array è inizializzato con le stringhe *Uno* e *Due*. Utilizzando questo array è possibile generare il codice HTML corrispondente agli elementi. Poiché ad ogni pulsante è associato una casella di testo, la generazione del codice tiene conto anche di questo elemento.

La pagina è così definita.

```
<html>
<head>
<script type="text/javascript" src="Pulsanti2.js"></script>
</head>
<body>
 <form id="modulo"></form>
</body>
</html>
```

Il programma JavaScript è il seguente.

```
var nodoForm;
function inizializza() {
   var valori = ["Uno", "Due"];
   var s = "";
   for (var i = 0; i < valori.length; i++) {
      s += '<input type="text" name="testo' + i + '">';
      s += '<input type="button" value="' + valori[i] + '"';
      s += 'onclick="pulsante' + (i + 1) +
           '(testo' + i + '.value);"/>';
         if (i < valori.length - 1) {
            s += '<br/>';
         }
      }
   nodoForm = document.getElementById("modulo");
   nodoForm.innerHTML = s;
}
function pulsante1(s) {
   alert(s);
```

```
}
function pulsante2(s) {
   alert(s);
}
window.onload = inizializza;
```

La soluzione appena presentata assume che ad ogni pulsante sia associata una funzione specifica. Per generalizzare il problema a un numero variabile di pulsanti è necessario definire un'unica funzione che ha come parametro il numero del pulsante che ha generato l'evento click. Al suo interno la funzione gestisce opportunamente questo valore. Il codice HTML resta invariato, mentre cambia il programma JavaScript.

```
var nodoForm;
function inizializza() {
   var valori = ["Uno", "Due", "Tre"];
   var s = "";
   for (var i = 0; i < valori.length; i++) {
      s += '<input type="text" name="testo' + i + '">';
      s += '<input type="button" value="' + valori[i] + '"';
      s += 'onclick="pulsante' + '(' + (i + 1) +
           ', testo' + i + '.value);"/>';
      if (i < valori.length - 1) {
         s += '<br/>';
      }
   }
   nodoForm = document.getElementById("modulo");
   nodoForm.innerHTML = s;
}
function pulsante(i, s) {
   alert("Pulsante" + i + ": " + s);
}
window.onload = inizializza;
```

15.9 Esercizi

1. Definire in HTML una pagina che contiene una lista numerata di nomi di animali. Cliccando sul nome di un animale, il nome viene spostato in fondo alla lista.

2. Definire in HTML una pagina che contiene un'area di testo e un pulsante. Cliccando sul pulsante la pagina visualizza la lista delle parole (senza ripetizioni) contenute nell'area di testo.

16 XML

XML è un linguaggio per la rappresentazione strutturata di dati. L'acronimo XML significa e**X**tensible **M**arkup **L**anguage. Come HTML, anche XML è un linguaggio di marcatura ma, a differenza di HTML, le marche non sono predefinite. La loro definizione è a carico di chi è interessato a rappresentare i propri dati. Un'altra differenza rispetto ad HTML consiste nel fatto che XML non è stato progettato per visualizzare documenti ma per rappresentarli ai fini della loro memorizzazione e trasmissione.

16.1 Un documento XML

Un esempio molto semplice di documento XML è il seguente.

```
<?xml version="1.0"?>
<ricettario anno="2012">
 <ricetta categoria="Primo">
  <nome>Gnocchi</nome>
  <difficolta>media</difficolta>
  <preparazione>30</preparazione>
 </ricetta>
 <ricetta categoria="Secondo">
  <nome>Cotoletta</nome>
  <difficolta>bassa</difficolta>
  <preparazione>15</preparazione>
 </ricetta>
</ricettario>
```

Il documento rappresenta un *ricettario* del 2012 che contiene due ricette: *Gnocchi* e *Cotoletta*. Per ogni ricetta è indicata la *categoria*, il grado di *difficoltà* e il tempo di *preparazione*. Le marche utilizzate sono: *ricettario*, *ricetta*, *nome*, *difficolta*[11], *preparazione*. Gli attributi

11 Nelle marche XML non si possono usare segni diacritici.

utilizzati sono: *anno* e *categoria*. Il documento è memorizzato nel file *Ricettario.xml*.

16.2 Il parser XML

Il contenuto di un documento XML può essere letto mediante un *parser*, un programma che riconosce la struttura del documento espressa mediante le marche e gli attributi associati e costruisce l'albero DOM corrispondente al documento. L'invocazione del parser può essere effettuata in modi diversi, che dipendono dal particolare browser utilizzato. Nel seguito presentiamo la funzione *caricaXML* che funziona correttamente con i principali browser attualmente disponibili.

```
function caricaXML(nomeFile) {
   var xmlhttp;
   if (window.XMLHttpRequest) {
      // IE7+, Firefox, Chrome, Opera, Safari
      xmlhttp = new XMLHttpRequest();
   } else {
      // IE6, IE5
      xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
   }
   xmlhttp.open("GET", nomeFile, false);
   xmlhttp.send();
   return xmlhttp.responseXML;
}
```

La funzione restituisce l'albero DOM che rappresenta il documento XML memorizzato nel file indicato nel parametro *nomeFile*.

Per leggere il contenuto del file *Ricettario.xml*, che contiene il documento XML descritto in precedenza, si invoca il parser con il seguente comando. Per ragioni di sicurezza, il parser può accedere solo a file memorizzati nella stessa cartella da cui si attiva *EasyJS* o in quella in cui è memorizzato il file HTML a cui è associato il programma JavaScript che contiene l'invocazione del parser.

```
var d = caricaXML("Ricettario.xml");
```

Per visualizzare il contenuto del file XML usiamo una pagina HTML così definita.

```
<html>
<head>
<script type="text/javascript"
        src="Ricettario.js">
</script>
</head>
<body>
<b>Le mie ricette</b>
<br/>
<ol id="lista"></ol>
</body>
</html>
```

Il file *Ricettario.js* contiene un programma JavaScript che carica il file *Ricettario.xml* utilizzando la funzione *caricaXML* vista in precedenza e modifica dinamicamente la pagina.

```
function caricaXML() { … }

function crea(l) {
   var s = "";
   for (var i = 0; i < l.length; i++) {
      var ln = l[i].getElementsByTagName("nome");
      var n = ln[0].firstChild.nodeValue;
      var ld = l[i].getElementsByTagName("difficolta");
      var d = ld[0].firstChild.nodeValue;
      var lp = l[i].getElementsByTagName("preparazione");
      var p = lp[0].firstChild.nodeValue;
      var r = n + ", difficoltà " +
              d + ", " +
              p + " minuti di preparazione";
      s += "<li>" + r + "</li>";
   }
   return s;
}
function inizializza() {
   var nodo = caricaXML("Ricettario.xml");
   var l = nodo.getElementsByTagName("ricetta");
   var lista = document.getElementById("lista");
   lista.innerHTML = crea(l);
```

```
}
window.onload = inizializza;
```

16.3 Creazione di oggetti definiti mediante XML

In molte situazioni è necessario effettuare dei calcoli sui dati caricati dinamicamente da un file XML. Questi calcoli sono richiesti dall'utente mediante l'interfaccia grafica di una pagina realizzata mediante campi di testo, pulsanti e altri elementi di interazione previsti da HTML.

Una tecnica per rendere agevole l'esecuzione di questi calcoli consiste nella creazione di oggetti a partire dal contenuto di un file XML. Nel caso del ricettario, per esempio, è immediato riconoscere gli oggetti *Ricettario* e *Ricetta*, definiti mediante le marche con lo stesso nome. L'oggetto *Ricettario* avrà una proprietà *anno*, mentre l'oggetto Ricetta avrà le proprietà *categoria*, *nome*, *difficoltà* e *preparazione*. In JavaScript i due costruttori saranno così (parzialmente) definiti.

```
function Ricettario (a) {
   this.anno = a;
   this.lista = [];
}
function Ricetta (c, n, d, p) {
   this.categoria = c;
   this.nome = n;
   this.difficoltà = d;
   this.preparazione = p;
}
```

Dopo aver caricato il file *Ricettario.xml* si procede a creare un oggetto *Ricettario* e tanti oggetti *Ricetta*. La creazione di questi oggetti è guidata dalla struttura dell'albero DOM costruito dal parser XML che ha letto il file *Ricettario.xml*.

La creazione dell'oggetto *Ricetta* è effettuata dalla funzione *creaRicetta*, che ha come parametro un nodo dell'albero DOM creato dal parser XML il cui nome è *ricetta*.

```
function creaRicetta(nodo) {
   var c = nodo.getAttribute("categoria");
```

```
   var ln = nodo.getElementsByTagName("nome");
   var n = ln[0].firstChild.nodeValue;
   var ld = nodo.getElementsByTagName("difficolta");
   var d = ld[0].firstChild.nodeValue;
   var lp = nodo.getElementsByTagName("preparazione");
   var p = lp[0].firstChild.nodeValue;
   return new Ricetta(c, n, d, p);
}
```

La creazione dell'oggetto *Ricettario* è effettuata dalla funzione *creaRicettario*, che ha come parametro la radice dell'albero DOM creato dal parser XML.

```
function creaRicettario(radice) {
   var nodo = radice.getElementsByTagName("ricettario")[0];
   var anno = nodo.getAttribute("anno");
   var l = nodo.getElementsByTagName("ricetta");
   var a = [];
   for (var i = 0; i < l.length; i++) {
      a.push(creaRicetta(l[i]));
   }
   var r = new Ricettario(anno);
   r.lista = a;
   return r;
}
```

La creazione del ricettario avviene mediante l'invocazione della funzione *creaRicettario*, dopo l'invocazione della funzione *caricaXML*. La variabile globale *ilRicettario* è usata per memorizzare l'oggetto creato.

```
var ilRicettario;
function inizializza() {
   var nodo = caricaXML("Ricettario.xml");
   ilRicettario = creaRicettario(nodo);
}
window.onload = inizializza;
```

Una volta creato l'oggetto *Ricettario* e gli oggetti *Ricetta* è possibile definire i metodi che generano la stringa HTML da assegnare alla

proprietà *innerHTML* dell'elemento lista, per modificare dinamicamente la pagina.

```
this.visualizza =
   function() {
      var s = "";
      for (var i in this.lista) {
         s += this.lista[i].visualizza();
      }
      return s;
   }

this.visualizza =
   function() {
      return "<li>" +
             this.nome + ", difficoltà " +
             this.difficoltà + ", " +
             this.preparazione + " minuti di preparazione" +
             "</li>";
   }
```

La versione definitiva della funzione inizializza è la seguente.

```
function inizializza() {
   var nodo = caricaXML("Ricettario.xml");
   ilRicettario = creaRicettario(nodo);
   var lista = document.getElementById("lista");
   lista.innerHTML = ilRicettario.visualizza();
}
```

La tecnica appena presentata può essere migliorata trasformando la funzione *creaRicettario* nel metodo *inizializza* dell'oggetto *Ricettario*. Anche la funzione *creaRicetta* può essere trasformata nel metodo *inizializza* dell'oggetto *Ricetta*. In questo modo la creazione degli oggetti è completamente gestita dai metodi degli oggetti stessi.

```
this.inizializza =
   function (r) {
      var nodo = r.getElementsByTagName("ricettario")[0];
      this.anno = nodo.getAttribute("anno");
      var l = nodo.getElementsByTagName("ricetta");
      for (var i = 0; i < l.length; i++) {
```

```
         var ri = new Ricetta();
         ri.inizializza(l[i]);
         this.lista.push(ri);
      }
   }
this.inizializza =
   function(nodo) {
      var c = nodo.getAttribute("categoria");
      this.categoria = c;
      var ln = nodo.getElementsByTagName("nome");
      var n = ln[0].firstChild.nodeValue;
      this.nome = n;
      var ld = nodo.getElementsByTagName("difficolta");
      var d = ld[0].firstChild.nodeValue;
      this.difficoltà = d;
      var lp = nodo.getElementsByTagName("preparazione");
      var p = lp[0].firstChild.nodeValue;
      this.preparazione = p;
   }
```

Anche i due costruttori devono essere opportunamente modificati.

```
function Ricettario () {
   this.anno;
   this.lista = [];
}
function Ricetta () {
   this.categoria;
   this.nome;
   this.difficoltà;
   this.preparazione;
}
```

La funzione *inizializza* deve essere modificata per tener conto delle modifiche effettuate.

```
function inizializza() {
   var nodo = caricaXML("Ricettario.xml");
   ilRicettario = new Ricettario();
   ilRicettario.inizializza(nodo);
   var lista = document.getElementById("lista");
```

```
   lista.innerHTML = ilRicettario.visualizza();
}
```

16.4 Esercizi

1. Definire in JavaScript una funzione che calcola e restituisce una stringa che rappresenta un documento XML. La funzione ha come argomento l'albero DOM creato dal parser XML a partire dal documento XML stesso. La funzione assume che nel documento XML le parti testuali siano solo all'interno di coppie di marche terminali (che non hanno figli, cioè). Invocare la funzione utilizzando il file *Ricettario.xml* dopo averlo letto mediante la funzione *caricaXML*.
2. Definire in JavaScript una funzione che calcola e restituisce una stringa che rappresenta un documento XML. La funzione ha come argomento l'albero DOM creato dal parser XML a partire dal documento XML stesso. La funzione assume che nel documento XML le parti testuali siano solo all'interno di coppie di marche terminali (che non hanno figli, cioè). Infine, la funzione visualizza anche il valore degli attributi XML, rispettandone la sintassi.

   Invocare la funzione utilizzando il file *Ricettario.xml* dopo averlo letto mediante la funzione *caricaXML*.
3. Definire in HTML una pagina che visualizza un menu a tendina contenente i nomi degli autori (senza ripetizioni) dei libri contenuti in un documento XML.

17 Un esempio completo

In questo capitolo presentiamo un esempio che ci serve per mettere in pratica i concetti presentati finora. Abbiamo scelto di mostrare la realizzazione di una rubrica sia per ragioni storiche sia perché con questo esempio possiamo affrontare una buona parte dei problemi che si presentano quando si vuole rendere dinamica una pagina HTML mediante un programma JavaScript.

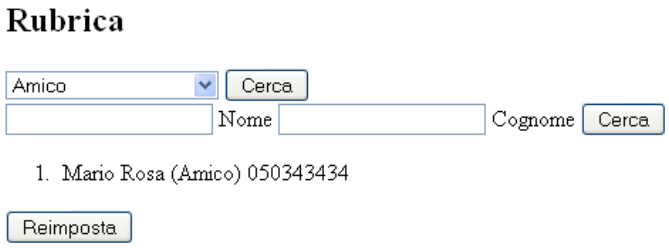
17.1 Il problema

Una *rubrica* è un elenco di *voci*, ognuna delle quali contiene informazioni su una persona: nome e cognome, numeri di telefono, indirizzi, data di nascita, tanto per citarne alcune. Tradizionalmente realizzate su carta, le rubriche si sono trasformate in oggetti digitali che troviamo su cellulari, calcolatori, telefoni. Questo esempio affronta il problema di realizzare una rubrica che permette di cercare le voci in base a due criteri di ricerca. Per semplicità, per ogni voce la rubrica memorizza il nome, il cognome, il numero di telefono e il gruppo di appartenenza di una persona.

17.2 Il codice HTML

La rubrica è realizzata mediante una pagina HTML. La ricerca per gruppo utilizza un menu a tendina mediante il quale si seleziona uno dei gruppi di appartenenza delle persone. Il pulsante alla destra del menu effettua la ricerca delle voci che saranno poi elencate nella parte bassa della pagina. La ricerca per nome o per cognome utilizza due campi di testo nei quali si inseriscono i valori da usare per la ricerca. Anche in questo caso, il pulsante alla destra dei due campi effettua la ricerca. L'esito negativo della ricerca è segnalato da una finestra di

alert. Il pulsante *Reimposta* cancella il contenuto delle caselle e aggiorna la scelta corrente del menu.



La figura mostra il risultato della ricerca delle voci che appartengono al gruppo *Amico*.

Il codice completo del documento *Rubrica.html* è il seguente.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
 <head>
  <meta http-equiv="Content-type"
        content="charset=windows-1252"/>
  <title>Rubrica</title>
  <script type="text/javascript"
          src="Rubrica.js"></script>
 </head>
 <body>
  <h2>Rubrica</h2>
  <form>
   <select id="selectGruppo" size="1"></select>
   <input type="button"
          value="Cerca"
          id="cercaPerGruppo"/>
   <br/>
   <input type="text"
          id="nome"/> Nome
   <input type="text"
          id="cognome"/> Cognome
   <input type="button"
          value="Cerca"
         id="cercaPerNomeCognome"/>
```

```
   <br/>
   <ol id="elenco"></ol>
   <input type="reset"
          id="reimposta"/>
  </form>
 </body>
</html>
```

La pagina contiene un modulo al cui interno sono dichiarati un menu, tre pulsanti, due campi di testo, una lista di elementi puntati. Il menu e la lista sono generati dinamicamente, come vedremo nel seguito. Il codice JavaScript è contenuto nel file *Rubrica.js*, come indicato dal tag *script*.

17.3 Gli oggetti Rubrica e Voce

Le informazioni relative alla rubrica sono mantenute in due oggetti: *Voce* e *Rubrica*. L'oggetto *Voce* memorizza le informazioni relative a una voce. L'oggetto *Rubrica* gestisce tutti gli aspetti relativi alla rubrica vera e propria e contiene un array di oggetti *Voce*.

Il costruttore dell'oggetto *Voce* ha quattro proprietà: *nome*, *cognome*, *telefono*, *gruppo*.

```
function Voce() {
   this.nome;
   this.cognome;
   this.telefono;
   this.gruppo;
}
```

Il costruttore dell'oggetto *Rubrica* ha due proprietà *voci* e *gruppi*, inizializzate entrambi con un array vuoto. La proprietà *voci* contiene le voci della rubrica, la proprietà *gruppi* contiene i gruppi di appartenenza delle voci.

```
function Rubrica() {
   this.voci = [];
   this.gruppi = [];
}
```

17.4 Caricamento e inizializzazione

La creazione della rubrica, inizialmente vuota, viene effettuata dal gestore dell'evento *load* (evento generato al termine del caricamento della pagina HTML) che invoca il metodo *inizializza* del costruttore *Rubrica*, il cui unico parametro è il nome del file (*Rubrica.xml*) che contiene la descrizione del contenuto della rubrica.

```
<?xml version="1.0"?>
<rubrica>
 <voce gruppo="Amico">
  <nome>Mario</nome>
  <cognome>Rosa</cognome>
  <telefono>050343434</telefono>
 </voce>
 <voce gruppo="Parente">
  <nome>Carlo</nome>
  <cognome>Neri</cognome>
  <telefono>050434343</telefono>
 </voce>
</rubrica>
```

Il gestore dell'evento *load* è il seguente.

```
var r;

function inizializza() {
   r = new Rubrica();
   r.inizializza("Rubrica.xml");
}

window.onload = inizializza;
```

Il metodo *inizializza* dell'oggetto *Rubrica* è il seguente.

```
this.inizializza =
   function(nomeFile) {
      var doc = caricaXML(nomeFile);
      var l = doc.getElementsByTagName("voce");
      for (i in l) {
         var v = new Voce();
         v.inizializza(l[i]);
```

```
         this.voci.push(v);
         var j = 0;
         while ((j < this.gruppi.length) &&
                (this.gruppi[j].gruppo != v.gruppo)) {
            j++;
         }
         if (j == this.gruppi.length) {
            this.gruppi.push(v.gruppo);
         }
      }
   }
```

Questo metodo utilizza il metodo *inizializza* dell'oggetto *Voce*.

```
this.inizializza =
   function(nodo) {
      var l;
      l = nodo.getElementsByTagName("nome");
      this.nome = l[0].firstChild.nodeValue;
      l = nodo.getElementsByTagName("cognome");
      this.cognome = l[0].firstChild.nodeValue;
      l = nodo.getElementsByTagName("telefono");
      this.telefono = l[0].firstChild.nodeValue;
      this.gruppo = nodo.getAttribute("gruppo");
   }
```

17.5 Gestione degli eventi

La creazione dinamica del menu di selezione per la ricerca basata sul nome del gruppo è effettuata dal metodo *creaSelect* dell'oggetto *Rubrica*.

```
this.creaSelect =
   function() {
      var s = "";
      s += '<option value="" selected="selected">' +
           'Seleziona un gruppo</option>'
      var l = r.gruppi;
      for (i in l) {
         s += '<option value="' + l[i] + '">' +
              l[i] + '</option>';
      }
```

```
      return s;
   }
```

La gestione degli eventi *click* (eventi generati premendo uno dei tre pulsanti presenti nella pagina) viene effettuata estendendo la definizione della funzione *inizializza*.

```
function inizializza() {
   var p1 = document.getElementById("cercaPerGruppo");
   p1.onclick = cercaPerGruppo;
   var p2 = document.getElementById("cercaPerNomeCognome");
   p2.onclick = cercaPerNomeCognome;
   var p3 = document.getElementById("reimposta");
   p3.onclick = reimposta;
   r = new Rubrica();
   r.inizializza("Rubrica.xml");
   var nodo = document.getElementById("selectGruppo");
   nodo.innerHTML = r.creaSelect();
}
```

17.6 Ricerca di voci

Le funzioni di ricerca previste per la rubrica sono attivate dai due pulsanti che affiancano il menu e le caselle di testo. Per ogni pulsante è stata definita una funzione, associata al relativo evento *click*. Per la ricerca basata sul nome del gruppo la funzione si chiama *cercaPerGruppo*, per la ricerca basata sul nome o sul cognome la funzione si chiama *cercaPerNomeCognome*.

```
function cercaPerGruppo() {
   var s = document.getElementById("selectGruppo");
   var i = 0;
   while ((i < s.options.length) &&
          !s.options[i].selected) {
      i++;
   }
   var l = r.cercaGruppo(s.options[i].value);
   var nodo = document.getElementById("elenco");
   if (l == "") {
      nodo.innerHTML = null;
      alert("Nessuna voce trovata");
```

```
   } else {
      nodo.innerHTML = r.visualizza(l);
   }
}

function cercaPerNomeCognome() {
   var n = document.getElementById("nome").value;
   var c = document.getElementById("cognome").value;
   var l = r.cercaNomeCognome(n, c);
   var nodo = document.getElementById("elenco");
   if (l == "") {
      nodo.innerHTML = null;
      alert("Nessuna voce trovata");
   } else {
      nodo.innerHTML = r.visualizza(l);
   }
}
```

Per effettuare la ricerca le due funzioni invocano, rispettivamente, il metodo *cercaGruppo* e *cercaNomeCognome* dell'oggetto *Rubrica*.

```
this.cercaGruppo =
   function(g) {
      var l = [];
      for (i in this.voci) {
         if (this.voci[i].gruppo == g) {
            l.push(this.voci[i]);
         }
      }
      return l;
   }
this.cercaNomeCognome =
   function(n, c) {
      var l = [];
      for (i in this.voci) {
         if ((this.voci[i].nome == n) ||
             (this.voci[i].cognome == c)) {
            l.push(this.voci[i]);
         }
      }
```

```
      return l;
   }
```

17.7 Visualizzazione

La visualizzazione dei risultati utilizza una finestra di *alert*, in caso di esito negativo, il metodo *visualizza* dell'oggetto *Rubrica* altrimenti.

```
this.visualizza =
   function(l) {
      var s = "";
      for (i in l) {
         s += l[i].visualizza();
      }
      return s;
   }
```

La visualizzazione delle informazioni di una voce è a carico del metodo *visualizza* dell'oggetto *Voce*.

```
this.visualizza =
   function() {
      var n = this.nome;
      var c = this.cognome;
      var g = this.gruppo;
      var t = this.telefono;
      return '<li>' +
             n + ' ' + c + " (" + g + ") " + t +
             '</li>';
   }
```

Infine, il pulsante *Reimposta* è associato alla funzione *reimposta*.

```
function reimposta() {
   var nodo = document.getElementById("elenco");
   nodo.innerHTML = null;
}
```

18 Soluzione degli esercizi della seconda parte

18.1 Esercizi del capitolo 11

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.

```
function sottrazione(x, y) {
   if (y == 0) {
      return x;
   } else {
      return sottrazione(x - 1, y - 1);
   }
}
print (sottrazione(5, 0));
print (sottrazione(3, 3));
print (sottrazione(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.

```
function minoreDi(x, y) {
   if (x == 0 && y == 0) {
      return false;
   } else if (x != 0 && y == 0) {
      return false;
   } else if (x == 0 && y != 0) {
      return true;
   } else {
      return minoreDi(x - 1, y - 1);
   }
}
```

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.

```
function divisione(x, y) {
   if (minoreDi(x, y)) {
      return 0;
   } else {
      return divisione(sottrazione(x, y), y) + 1;
   }
}
```

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.

```
function resto(x, y) {
   if (minoreDi(x, y)) {
      return x;
   } else {
      return resto(sottrazione(x, y), y);
   }
}
```

5. Definire in JavaScript una funzione ricorsiva che calcola e restituisce l'esponenziazione 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.

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

18.2 Esercizi del capitolo 12

1. Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano *sveglie*.

   L'oggetto ha i seguenti metodi:

- *oraCorrente* (*h*, *m*): imposta l'ora corrente. Se *h* è maggiore di 23 o *m* è maggiore di 59, i rispettivi valori sono impostati a zero;
- *allarme* (*h*, *m*): imposta l'allarme. Se *h* è maggiore di 23 o *m* è maggiore di 59, i rispettivi valori sono impostati a zero;
- *tic* (): fa avanzare di uno i minuti dell'ora corrente. Se i minuti arrivano a 60, azzera i minuti e fa avanzare di uno le ore. Se le ore arrivano a 24, azzera le ore. Se l'ora corrente è uguale all'allarme impostato, restituisce il valore *true*, *false* altrimenti.

Definire una funzione che

- crea una sveglia,
- imposta l'ora corrente alle 13:00,
- imposta l'allarme alle 13:02,
- fa avanzare l'ora corrente di due minuti.

```
function Sveglia () {
   this.ore;
   this.minuti;
   this.oreAllarme;
   this.minutiAllarme;
   this.oraCorrente =
      function (h, m) {
         if (h > 23) {
            this.ore = 0;
         } else {
            this.ore = h;
         }
         if (m > 59) {
            this.minuti = 0;
         } else {
            this.minuti = m;
         }
      }
   this.allarme =
      function(h, m) {
         if (h > 23) {
```

```
            this.oreAllarme = 0;
         } else {
            this.oreAllarme = h;
         }
         if (m > 59) {
            this.minutiAllarme = 0;
         } else {
            this.minutiAllarme = m;
         }
      }
   this.tic =
      function () {
         this.minuti++;
         if(this.minuti == 60) {
            this.minuti = 0;
            this.ore++;
         }
         if (this.ore == 24) {
            this.ore = 0;
         }
         return ((this.ore == this.oreAllarme) &&
                 (this.minuti == this.minutiAllarme));
      }
}
function foo() {
   var s = new Sveglia();
   s.oraCorrente(13, 0);
   s.allarme(13, 2);
   print(s.tic());
   print(s.tic());
}
```

2. Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano *distributori automatici di caffè*.

   L'oggetto ha i seguenti metodi:

   - *carica* (*n*): aggiunge *n* capsule per erogare *n* caffè;
   - *eroga* (*n*, *c*): verifica se è possibile erogare *n* caffè e, in caso positivo, li eroga addebitandoli al codice *c*;

- *rapporto* (*c*): restituisce il numero di caffè addebitati al codice *c* e il numero di capsule disponibili.

Definire una funzione che:

- crea un distributore automatico di caffè,
- carica 20 capsule,
- eroga 12 caffè per il codice *Carlo*,
- genera il rapporto per il codice *Carlo*.

```
function Distributore() {
   this.capsule = 0;
   this.crediti = {};
   this.carica =
      function(n) {
         this.capsule += n;
      }
   this.eroga =
      function(n, c) {
         if(n < this.capsule){
            this.capsule -= n;
            if (c in this.crediti) {
               this.crediti[c] = +n;
            } else {
               this.crediti[c] = n;
            }
         }
      }
   this.rapporto =
      function (c) {
         return c + " " + this.crediti[c] +
                " Capsule " + this.capsule;
      }
}
function foo() {
   var d = new Distributore();
   d.carica(20);
   d.eroga(12, "Carlo");
   print(d.rapporto("Carlo"));
}
```

18.3 Esercizi del capitolo 13

1. Un'*espressione aritmetica* può essere rappresentata mediante un albero binario i cui nodi sono *operatori* (addizione, sottrazione, moltiplicazione, divisione) e le foglie sono valori numerici.

   Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano espressioni aritmetiche.

   L'oggetto ha i seguenti metodi:

   - *calcola* (): restituisce il valore dell'espressione,
   - *visualizza* (): restituisce una stringa che rappresenta l'espressione.

   Definire una funzione che

   - crea l'espressione *2 * 3 + 5 – 6 / 2 + 1*
   - stampa la rappresentazione dell'espressione
   - calcola e stampa il valore dell'espressione.

```
function Espressione(v, sx, dx) {
 this.valore   = v;
 this.sinistro = sx;
 this.destro   = dx;
 this.visualizza =
 function() {
 var vs;
 if (this.sinistro != null) {
 vs = this.sinistro.visualizza ();
 } else {
 var vs = "";
 }
 var vd;
 if (this.destro != null) {
 vd = this.destro.visualizza ();
 } else {
 vd = "";
 }
         return vs + " " + this.valore + vd;
      }
```

```
    this.calcola =
       function() {
          if (this.sinistro == null &&
              this.destro == null) {
             return this.valore;
          }
          switch (this.valore) {
             case "+": var vs =this.sinistro.calcola();
                       var vd = this.destro.calcola();
                       return vs + vd;
             case "-": var vs =this.sinistro.calcola();
                       var vd = this.destro.calcola();
                       return vs - vd;
             case "*": var vs =this.sinistro.calcola();
                       var vd = this.destro.calcola();
                       return vs * vd;
             case "/": var vs =this.sinistro.calcola();
                       var vd = this.destro.calcola();
                       return vs / vd;
           }
       }
}
function foo() {
   var e = new Espressione("+",
             new Espressione("-",
              new Espressione("+",
               new Espressione("*",
                new Espressione(2, null, null),
                new Espressione(3, null, null)),
               new Espressione(5, null, null)),
              new Espressione("/",
               new Espressione(6, null, null),
               new Espressione(2, null, null))),
             new Espressione(1, null, null));
   print(e.visualizza());
   print(e.calcola());
}
```

2. L'*albero genealogico* è una rappresentazione (parziale) che mostra i rapporti familiari tra gli antenati di un individuo. Abitualmente un albero genealogico è realizzato utilizzando delle

caselle, quadrate per i maschi e circolari per le femmine, contenenti i nomi di ciascuna persona, spesso corredate di informazioni aggiuntive, quali luogo e data di nascita e morte, occupazione o professione. Questi simboli, disposti dall'alto verso il basso in ordine cronologico, sono connessi da vari tipi di linee che rappresentano matrimoni, unioni extra coniugali, discendenza [Wikipedia, alla voce *Albero genealogic*o].

Un *albero genealogico patrilineare* è un particolare tipo di albero genealogico in cui sono rappresentati tutti e soli i discendenti per via paterna.

Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano alberi genealogici patrilineari.

Il costruttore ha un parametro: *persona*.

L'oggetto ha i seguenti metodi:

- *aggiungiConiuge* (*persona*, *coniuge*): aggiunge, nell'albero genealogico, il *coniuge* a una *persona*;
- *aggiungiDiscendente* (*persona*, *discendente*): aggiunge, nell'albero genealogico, un *discendente* diretto a una *persona*;
- *visualizzaPersona* (*persona*): restituisce una stringa contenente le informazioni relative a una *persona*;
- *grado*(*persona*): calcola il grado di parentela tra una *persona* e il suo antenato più anziano nell'albero genealogico;
- *linea* (*persona*): restituisce una stringa contenente la linea di discendenza tra una *persona* e il suo antenato più anziano nell'albero genealogico.

Definire una funzione che

- crea un albero genealogico con queste caratteristiche: *Bruno*, *Carlo* e *Daniela* sono figli di *Aldo*, *Enzo* e *Fabio* sono figli di *Bruno*, *Giacomo* è figlio di *Carlo*, *Alessia* è la moglie di *Aldo*, *Beatrice* è la moglie di *Bruno*, *Cecilia* è la moglie di *Carlo*;
- visualizza le informazioni di *Aldo* ed *Enzo*;

- calcola il grado di parentela di *Bruno*, *Giacomo* e *Fabio*;
- visualizza la linea di discendenza di *Enzo* e *Giacomo*.

```
function AlberoGenealogico(persona) {
   this.nome = persona;
   this.coniuge = "";
   this.figli = []
   this.aggiungiConiuge =
      function (p, c) {
         if (p == this.nome) {
            this.coniuge = c;
         } else {
            for (var i in this.figli) {
               this.figli[i].aggiungiConiuge(p, c);
            }
         }
      }
   this.aggiungiDiscendente =
      function (p, d) {
         if (p == this.nome) {
            var n = new AlberoGenealogico(d);
            this.figli.push(n);
         } else {
            for (var i in this.figli) {
              this.figli[i].aggiungiDiscendente(p, d);
            }
         }
      }
   this.visualizzaPersona =
      function (p) {
         if (p == this.nome) {
            return(this.nome +
                   " (" + this.coniuge + ")");
         } else {
            var s = ""
            for (var i in this.figli) {
              s += this.figli[i].visualizzaPersona(p);
            }
         return s;
         }
      }
```

```
    this.grado =
       function (p) {
          if (p == this.nome) {
             return 1;
          } else {
             var d = 0;
             for (var i in this.figli) {
                var df = this.figli[i].grado(p);
                if (df > d) {
                   d = df;
                }
             }
             if (d > 0) {
                return d + 1;
             } else {
                return 0;
             }
           }
         }
    this.linea =
       function (p) {
          if (p == this.nome) {
             return this.nome;
          }
          var s = "";
          for (var i in this.figli) {
             s += this.figli[i].linea(p)
          }
          if (s != "") {
             return this.nome + " - " + s;
          } else {
             return s;
          }
       }
}

function foo() {
   var ag = new AlberoGenealogico("Aldo");
   ag.aggiungiConiuge("Aldo", "Alessia");
   ag.aggiungiDiscendente("Aldo", "Bruno");
   ag.aggiungiConiuge("Bruno", "Beatrice");
```

```
    ag.aggiungiDiscendente("Aldo", "Carlo");
    ag.aggiungiConiuge("Carlo", "Cecilia");
    ag.aggiungiDiscendente("Aldo", "Daniela");
    ag.aggiungiDiscendente("Bruno", "Enzo");
    ag.aggiungiDiscendente("Bruno", "Fabio");
    ag.aggiungiDiscendente("Carlo", "Giacomo");
    print(ag.visualizzaPersona("Aldo"));
    print(ag.visualizzaPersona("Enzo"));
    print("Bruno : grado " + ag.grado("Bruno"));
    print("Giacomo : grado " + ag.grado("Giacomo"));
    print("Fabio : grado " + ag.grado("Fabio"));
    print(ag.linea("Enzo"));
    print(ag.linea("Giacomo"));
}
```

18.4 Esercizi del capitolo 14

1. Definire in HTML una pagina relativa a un ristorante. La pagina deve contenere una descrizione e un menu. Il menu deve essere diviso in più parti: antipasti, primi, secondi, contorni, dolci. Cliccando sul nome di un piatto deve comparire il suo prezzo. Passando il mouse sul nome del proprietario, indicato nella descrizione, deve comparire un breve messaggio. Un doppio clic sul nome del ristorante, indicato nella descrizione, deve far comparire un numero di telefono.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-type"
      content="charset=windows-1252"/>
<title>Taverna Rossa</title>
</head>
<body>
<h1 ondblclick="alert('050 221133');">Taverna Rossa</h1>
Ristorante gestito da <span onmouseover="alert('Nato a Pisa');">Mar-
co</span>
<br/>
<h3>Antipasti</h3>
<ul>
  <li onclick="alert('5 Euro');">Prosciutto e melone</li>
  <li onclick="alert('6 Euro');">Crostini di mare</li>
```

```
</ul>
<h3>Primi</h3>
<ul>
  <li onclick="alert('7 Euro');">Pasta alla carbonara</li>
  <li onclick="alert('8 Euro');">Gnocchi</li>
</ul>
<h3>Secondi</h3>
<ul>
  <li onclick="alert('9 Euro');">Cotoletta</li>
  <li onclick="alert('8 Euro');">Frittura di mare</li>
</ul>
<h3>Contorni</h3>
<ul>
  <li onclick="alert('3 Euro');">Patate</li>
  <li onclick="alert('2 Euro');">Carote</li>
</ul>
<h3>Dolci</h3>
<ul>
  <li onclick="alert('4 Euro');">Torta al limone</li>
  <li onclick="alert('5 Euro');">Budino al cioccolato</li>
</ul>
</body>
</html>
```

18.5 Esercizi del capitolo 15

1. Definire in HTML una pagina che contiene una lista numerata di nomi di animali. Cliccando sul nome di un animale, il nome viene spostato in fondo alla lista.

```
<html>
<head>
<script type="text/javascript" src="Dinamica.js">
</script>
</head>
<body>
  <ol>
    <li>Cane</li>
    <li>Gatto</li>
    <li>Topo</li>
  </ol>
</body>
</html>
```

```
function sposta() {
   var nodo = this.parentNode;
   nodo.removeChild(this);
   nodo.appendChild(this);
}
function inizializza() {
   var l = document.getElementsByTagName("li");
   for (var i in l) {
      l[i].onclick = sposta;
   }
}
window.onload = inizializza;
```

2. Definire in HTML una pagina che contiene un'area di testo e un pulsante. Cliccando sul pulsante la pagina visualizza la lista delle parole (senza ripetizioni) contenute nell'area di testo.

```
<html>
<head>
<script type="text/javascript" src="Frase.js">
</script>
</head>
<body>
<form>
  <textarea cols="60" rows="10" id="area"></textarea>
  <br/>
  <input type="button" value="Estrai" id="estrai"/>
  <br/>
  <ol id="lista"></ol>
</form>
</body>
</html>
```

```
function estrai() {
   var testo = document.getElementById("area").value;
   var a = [];
   var i = 0;
   while (i < testo.length) {
      var p = "";
      while ((i < testo.length) &&
```

```
                (testo[i]!= " ")) {
            p += testo[i];
            i++;
         }
         var j = 0;
         while ((j < a.length) &&
                (a[j] != p)) {
            j++;
         }
         if (j == a.length) {
            a.push(p);
         }
         while ((i < testo.length) &&
                (testo[i]== " ")) {
            i++;
         }
      }
      var s = "";
      for (var k in a) {
         s += "<li>" + a[k] + "</li>";
      }
      var nodo = document.getElementById("lista");
      nodo.innerHTML = s;
   }
   function inizializza() {
      var pulsante = document.getElementById("estrai");
      pulsante.onclick = estrai;
   }
   window.onload = inizializza;
```

18.6 Esercizi del capitolo 16

1. Definire in JavaScript una funzione che calcola e restituisce una stringa che rappresenta un documento XML. La funzione ha come argomento l'albero DOM creato dal parser XML a partire dal documento XML stesso. La funzione assume che nel documento XML le parti testuali siano solo all'interno di coppie di marche terminali (che non hanno figli, cioè). Invocare la funzione utilizzando il file *Ricettario.xml* dopo averlo letto mediante la funzione *caricaXML*.

```
function visitaDOM (nodo, r) {
   var s = "";
   s += rientro + nodo.nodeName + "\n";
   for (var i in nodo.childNodes) {
      s += visitaDOM(nodo.childNodes[i], r + ' ');
   }
   return s;
}

print(visitaDOM(caricaXML("Ricettario.xml"), " "));
```

2. Definire in JavaScript una funzione che calcola e restituisce una stringa che rappresenta un documento XML. La funzione ha come argomento l'albero DOM creato dal parser XML a partire dal documento XML stesso. La funzione assume che nel documento XML le parti testuali siano solo all'interno di coppie di marche terminali (che non hanno figli, cioè). Infine, la funzione visualizza anche il valore degli attributi XML, rispettandone la sintassi.

   Invocare la funzione utilizzando il file *Ricettario.xml* dopo averlo letto mediante la funzione *caricaXML*.

```
function generaXML (nodo, r) {
   if (nodo.nodeType == 1) {
      var s = "";
      if ((nodo.childNodes.length == 1) &&
          (nodo.childNodes[0].nodeType == 3)) {
         s += r + "<" + nodo.nodeName;
         for (var i in nodo.attributes) {
            s += nodo.attributes[i].nodeName + '="' +
                 nodo.attributes[i].nodeValue + '"';
         }
         s += ">"
         s += nodo.childNodes[0].nodeValue;
         s += "</" + nodo.nodeName + ">\n";
         return s;
      } else {
         s += r + "<" + nodo.nodeName;
         for (var i in nodo.attributes) {
            s += " " + nodo.attributes[i].nodeName +
```

```
                    '="' +
                    nodo.attributes[i].nodeValue + '"';
            }
            s += ">\n";
            for (var i in nodo.childNodes) {
               if (nodo.childNodes[i].nodeType == 1) {
                  s += generaXML(nodo.childNodes[i],
                                 r + ' ');
               }
            }
            s += r + "</" + nodo.nodeName + ">\n";
            return s;
         }
      } else if (nodo.nodeType == 3) {
         return r + nodo.nodeName + " " +
                nodo.nodeType + "\n";
      } else {
         return generaXML(nodo.childNodes[0], r);
      }
   }

   print(generaXML(caricaXML("Ricettario.xml"), " "));
```

3. Definire in HTML una pagina che visualizza un menu a tendina contenente i nomi degli autori (senza ripetizioni) dei libri contenuti in un documento XML.

```
<?xml version="1.0"?>
<biblioteca>
 <libro genere="Narrativa">
  <autore>Dan Brown</autore>
  <titolo>Il codice da Vinci</titolo>
 </libro>
 <libro genere="Narrativa">
  <autore>Dan Brown</autore>
  <titolo>Angeli e demoni</titolo>
 </libro>
 <libro genere="Poesia">
  <autore>Giacomo Leopardi</autore>
  <titolo>Nelle nozze della sorella Paolina</titolo>
 </libro>
```

```
 <libro genere="Narrativa">
  <autore>Umberto Eco</autore>
  <titolo>Il nome della rosa</titolo>
 </libro>
</biblioteca>
```

```
<html>
<head>
<script type="text/javascript"
        src="Biblioteca.js">
</script>
</head>
<body>
<b>La mia biblioteca</b>
<br/>
<br/>
<select id="selectAutori"></select>
</body>
</html>
```

```
function caricaXML(nomeFile) { … }
function Biblioteca () {
 this.lista = [];
 this.autori = {};
 this.inizializza =
 function (r) {
         var nodo =
             r.getElementsByTagName("biblioteca")[0];
         var l = nodo.getElementsByTagName("libro");
         for (var i = 0; i < l.length; i++) {
            var li = new Libro();
            li.inizializza(l[i]);
            this.lista.push(li);
            this.autori[li.autore] = true;
         }
      }
   this.listaAutori =
      function() {
```

```
            var s = "";
            for (var i in this.autori) {
               s += '<option value="' + i + '">' + i +
                    '</option>';
            }
            return s;
         }
   }
   function Libro () {
      this.autore;
      this.genere;
      this.titolo;
      this.inizializza =
         function(nodo) {
            var g = nodo.getAttribute("genere");
            this.genere = g;
            var la = nodo.getElementsByTagName("autore");
            var a = la[0].firstChild.nodeValue;
            this.autore = a;
            var lt = nodo.getElementsByTagName("titolo");
            var t = lt[0].firstChild.nodeValue;
            this.titolo = t;
         }
   }
   var laBiblioteca;
   function inizializza() {
      var nodo = caricaXML("Biblioteca.xml");
      laBiblioteca = new Biblioteca();
      laBiblioteca.inizializza(nodo);
      var selectAutori =
          document.getElementById("selectAutori");
      selectAutori.innerHTML =
          laBiblioteca.listaAutori();
   }
   window.onload = inizializza;
```

19 Codice degli oggetti

In questo capitolo è riportato il codice completo degli oggetti presentati nella seconda parte.

19.1 Insieme

```
function Insieme(){
   this.elementi = {};
   this.cardinalità =
      function() {
         var c = 0;
         for (var i in this.elementi) {
            c++;
         }
         return c;
      }
   this.appartiene =
      function(x) {
         return (x in this.elementi);
      }
   this.aggiungi =
      function(k) {
         this.elementi[k] = k;
      }
   this.intersezione =
      function(x) {
         var n = new Insieme();
         for (var i in this.elementi) {
            if (x.appartiene(i)) {
                n.aggiungi(i);
            }
         }
         return n;
```

```
      }
   this.unione =
      function(x) {
         var n = new Insieme();
         for (var i in this.elementi){
            n.aggiungi(i);
         }
         for (var j in x.elementi){
            n.aggiungi(j);
         }
         return n;
      }
   this.visualizza =
      function() {
         var s = "{";
         var b = true;
         for (var i in this.elementi) {
            if (b) {
               s += i;
               b = false;
            } else {
               s += ", " + i;
            }
         }
         s += "}";
         return s;
      }
}

function foo() {
   var alfa = new Insieme();
   alfa.aggiungi(10);
   alfa.aggiungi(20);
   print(alfa.cardinalità());
   print(alfa.visualizza());
   var x = 10;
   if (alfa.appartiene(x)) {
      print (x + " appartiene ad alfa");
   } else {
      print (x + " non appartiene ad alfa");
   }
```

```
   var beta = new Insieme();
   beta.aggiungi(10);
   beta.aggiungi(30);
   print(beta.visualizza());
   print(alfa.intersezione(beta).visualizza());
   print(alfa.unione(beta).visualizza());
}
```

19.2 Tabella

```
function Tabella() {
   this.mappa = {};
   this.appartiene =
      function (c) {
         return (c in this.mappa);
      }
   this.aggiungi =
      function (c, v) {
         this.mappa[c] = v;
      }
   this.valore =
      function (c) {
         return this.mappa[c];
      }
   this.visualizza =
      function () {
         var s = '';
         for (var i in this.mappa) {
            s += ' ' + i + '="' + this.mappa[i] + '"';
         }
      return s;
      }
}

function foo() {
   var t = new Tabella();
   var c1 = "pippo"
   t.aggiungi(c1, 12);
   if (t.appartiene(c1)) {
      print (c1 + '="' + t.valore(c1) + '"');
   } else {
```

```
      print (c1 + ' non è in tabella');
   }
   var c2 = "pluto";
   t.aggiungi(c2, 13);
   if (t.appartiene(c2)) {
      print (c2 + '="' + t.valore(c2) + '"');
   } else {
      print (c2 + ' non è in tabella');
   }
   var c3 = "paperino";
   if (t.appartiene(c3)) {
      print (c3 + '="' + t.valore(c3) + '"');
   } else {
      print (c3 + ' non è in tabella');
   }
   print(t.visualizza());
   t.aggiungi(c2, 3);
   if (t.appartiene(c2, 14)) {
      print (c2 + '="' + t.valore(c2) + '"');
   } else {
      print (c2 + ' non è in tabella');
   }
   print(t.visualizza());
}
```

19.3 Albero binario

```
function AlberoBinario(v, sx, dx) {
   this.valore   = v;
   this.sinistro = sx;
   this.destro   = dx;
   this.visitaAnticipata =
      function() {
         var vs;
         if (this.sinistro != null) {
            vs = this.sinistro.visitaAnticipata();
         } else {
            vs = "";
         }
         var vd;
         if (this.destro != null) {
```

```
            vd = this.destro.visitaAnticipata();
         } else {
            vd = "";
         }
         return this.valore + " " + vs + vd;
   }
   this.visitaDifferita =
      function() {
         var vs;
         if (this.sinistro != null) {
            vs = this.sinistro.visitaDifferita();
         } else {
            var vs = "";
         }
         var vd;
         if (this.destro != null) {
            vd = this.destro.visitaDifferita();
         } else {
            vd = "";
         }
         return vs + vd + " " + this.valore;
      }
   this.visitaSimmetrica =
      function() {
         var vs;
         if (this.sinistro != null) {
            vs = this.sinistro.visitaSimmetrica();
         } else {
            var vs = "";
         }
         var vd;
         if (this.destro != null) {
            vd = this.destro.visitaSimmetrica();
         } else {
            vd = "";
         }
         return vs + " " + this.valore + vd;
      }
   this.altezza =
      function() {
         if (this.sinistro == null &&
```

```
            this.destro == null) {
            return 0;
        }
        var as;
        if (this.sinistro != null) {
            as = this.sinistro.altezza();
        } else {
            as = 0;
        }
        var ad;
        if (this.destro != null) {
            ad = this.destro.altezza();
        } else {
            ad = 0;
        }
        if (as > ad) {
            return as + 1;
        } else {
            return ad + 1;
        }
    }
this.frontiera =
    function() {
        if (this.sinistro == null &&
            this.destro == null) {
            return this.valore + " ";
        }
        var fs;
        if (this.sinistro != null) {
            fs = this.sinistro.frontiera();
        } else {
            fs = "";
        }
        var fd;
        if (this.destro != null) {
            fd = this.destro.frontiera();
        } else {
            fd = "";
        }
        return fs + fd;
```

```
      }
}

function foo() {
   var alfa = new AlberoBinario(1,
                  new AlberoBinario(2,
                     new AlberoBinario(4, null, null),
                     new AlberoBinario(5, null, null)),
                  new AlberoBinario(3, null, null));
   print(alfa.visitaAnticipata());
   print(alfa.visitaDifferita());
   print(alfa.visitaSimmetrica());
   print(alfa.altezza());
   print(alfa.frontiera());
}
```

19.4 Albero di ricerca

Il codice completo del costruttore *AlberoDiRicerca* è il seguente.

```
function AlberoDiRicerca (n) {
   this.valore = n;
   this.sinistro = null;
   this.destro = null;
   this.aggiungi =
      function (k) {
         if (k < this.valore) {
            if (this.sinistro == null) {
               this.sinistro = new AlberoDiRicerca(k);
            } else {
               this.sinistro.aggiungi(k);
            }
         }
         if (k > this.valore) {
            if (this.destro == null) {
               this.destro = new AlberoDiRicerca (k);
            } else {
               this.destro.aggiungi(k);
            }
         }
      }
   this.cerca =
```

```
      function (k) {
         if (k == this.valore) {
            return true;
         }
         if (k < this.valore) {
            if (this.sinistro == null) {
               return false;
            } else {
               return this.sinistro.cerca(k);
            }
         }
         if (k > this.valore) {
            if (this.destro == null) {
               return false;
            } else {
               return this.destro.cerca(k);
            }
         }
      }
   this.visita =
      function () {
         var vs;
         if (this.sinistro != null) {
            vs = this.sinistro.visita();
         } else {
            vs = "";
         }
         var vd;
         if (this.destro != null) {
            vd = this.destro.visita();
         } else {
            vd = "";
         }
         return vs + " " + this.valore + " " + vd;
      }
}

function foo() {
   var a = new AlberoDiRicerca(1);
   a.aggiungi(5);
   a.aggiungi(3);
```

```
   a.aggiungi(8);
   print(a.visita());
   print(a.cerca(1));
   print(a.cerca(4));
}
```

19.5 Albero n-ario

```
function AlberoNArio(v) {
   this.valore = v;
   this.figli = new Array();
   this.aggiungiFiglio =
      function (n) {
         this.figli[this.figli.length] = n;
      }
   this.visitaAnticipata =
      function() {
         var visita = "";
         for (var i = 0; i < this.figli.length; i++) {
           visita += this.figli[i].visitaAnticipata() + " ";
         }
         return this.valore + " " + visita;
      }
   this.visitaDifferita =
      function() {
         var visita = "";
         for (var i = 0; i < this.figli.length; i++) {
            visita += this.figli[i].visitaDifferita() + " ";
         }
         return visita + this.valore + " ";
      }
   this.altezza =
      function() {
         if (this.figli.length == 0) {
            return 0;
         }
         var maxAltezza = this.figli[0].altezza();
         for (var i = 1; i < this.figli.length; i++) {
            var altezzaFiglio = this.figli[i].altezza();
            if (altezzaFiglio > maxAltezza) {
               maxAltezza = altezzaFiglio;
```

```
            }
         }
         return maxAltezza + 1;
      }
   this.frontiera =
      function() {
         if (this.figli.length == 0) {
            return this.valore + " ";
         }
         var f = "";
         for (var i = 0; i < this.figli.length; i++) {
            f += this.figli[i].frontiera();
         }
         return f;
      }
}

function foo() {
   var a1 = new AlberoNArio(1);
   var a2 = new AlberoNArio(2);
   var a3 = new AlberoNArio(3);
   var a4 = new AlberoNArio(4);
   var a5 = new AlberoNArio(5);
   var a6 = new AlberoNArio(6);
   var a7 = new AlberoNArio(7);
   a1.aggiungiFiglio(a2);
   a1.aggiungiFiglio(a3);
   a1.aggiungiFiglio(a4);
   a2.aggiungiFiglio(a5);
   a2.aggiungiFiglio(a6);
   a3.aggiungiFiglio(a7);
   print(a1.visitaAnticipata());
   print(a1.visitaDifferita());
   print(a1.frontiera());
   print(a1.altezza());
}
```

19.6 Albero n-ario con attributi

```
function AlberoNArioAttr(v) {
   this.valore = v;
```

```
   this.figli = new Array();
   this.attributi = new Tabella();
   this.aggiungiFiglio =
      function (n) {
         this.figli[this.figli.length] = n;
      }
   this.aggiungiAttributo =
      function (n, v) {
         this.attributi.aggiungi(n, v);
      }
   this.visitaAnticipata =
      function() {
         var visita = "";
         for (var i = 0; i < this.figli.length; i++) {
           visita += this.figli[i].visitaAnticipata() + " ";
         }
         return this.valore + " " + visita;
      }
   this.visitaDifferita =
      function() {
         var visita = "";
         for (var i = 0; i < this.figli.length; i++) {
           visita += this.figli[i].visitaDifferita() + " ";
         }
         return visita + this.valore + " ";
      }
   this.altezza =
      function() {
         if (this.figli.length == 0) {
            return 0;
         }
         var maxAltezza = this.figli[0].altezza();
         for (var i = 1; i < this.figli.length; i++) {
            var altezzaFiglio = this.figli[i].altezza();
            if (altezzaFiglio > maxAltezza) {
               maxAltezza = altezzaFiglio;
            }
         }
         return maxAltezza + 1;
      }
   this.frontiera =
```

```
      function() {
         if (this.figli.length == 0) {
            return this.valore + " ";
         }
         var f = "";
         for (var i = 0; i < this.figli.length; i++) {
            f += this.figli[i].frontiera();
         }
         return f;
      }
}

function foo() {
   var a1 = new AlberoNArioAttr(1);
   var a2 = new AlberoNArioAttr(2);
   var a3 = new AlberoNArioAttr(3);
   a1.aggiungiFiglio(a2);
   a1.aggiungiFiglio(a3);
   a1.aggiungiAttributo("x", 100);
}
```

19.7 Ricettario

```
function Ricettario (a) {
   this.anno = a;
   this.lista = [];
   this.visualizza =
      function() {
         var s = "";
         for (var i in this.lista) {
            s += this.lista[i].visualizza();
         }
         return s;
      }
}

function Ricetta (c, n, d, p) {
   this.categoria = c;
   this.nome = n;
   this.difficoltà = d;
   this.preparazione = p;
```

```
   this.visualizza =
      function() {
         return "<li>" +
                this.nome + ", difficoltà " +
                this.difficoltà + ", " +
                this.preparazione +
                            " minuti di preparazione" +
                "</li>";
      }
}

function creaRicettario(radice) {
   var nodo = radice.getElementsByTagName("ricettario")[0];
   var anno = nodo.getAttribute("anno");
   var l = nodo.getElementsByTagName("ricetta");
   var a = [];
   for (var i = 0; i < l.length; i++) {
      a.push(creaRicetta(l[i]));
   }
   var r = new Ricettario(anno);
   r.lista = a;
   return r;
}

function creaRicetta(nodo) {
   var c = nodo.getAttribute("categoria");
   var ln = nodo.getElementsByTagName("nome");
   var n = ln[0].firstChild.nodeValue;
   var ld = nodo.getElementsByTagName("difficolta");
   var d = ld[0].firstChild.nodeValue;
   var lp = nodo.getElementsByTagName("preparazione");
   var p = lp[0].firstChild.nodeValue;
   return new Ricetta(c, n, d, p);
}

function caricaXML(nomeFile) {
   var xmlhttp;
   if (window.XMLHttpRequest) {
      // IE7+, Firefox, Chrome, Opera, Safari
      xmlhttp = new XMLHttpRequest();
   } else {
      // IE6, IE5
```

```
      xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
   }
   xmlhttp.open("GET", nomeFile, false);
   xmlhttp.send();
   return xmlhttp.responseXML;
}

var ilRicettario;

function inizializza() {
   var nodo = caricaXML("Ricettario.xml");
   ilRicettario = creaRicettario(nodo);
   var lista = document.getElementById("lista");
   lista.innerHTML = ilRicettario.visualizza();
}

window.onload = inizializza;
```

19.8 Rubrica

```
function Voce(nodo) {
   this.nome;
   this.cognome;
   this.telefono;
   this.gruppo;
   this.inizializza =
      function(nodo) {
         var l;
         l = nodo.getElementsByTagName("nome");
         this.nome = l[0].firstChild.nodeValue;
         l = nodo.getElementsByTagName("cognome");
         this.cognome = l[0].firstChild.nodeValue;
         l = nodo.getElementsByTagName("telefono");
         this.telefono = l[0].firstChild.nodeValue;
         this.gruppo = nodo.getAttribute("gruppo");
      }
   this.visualizza =
      function() {
         var n = this.nome;
         var c = this.cognome;
         var g = this.gruppo;
```

```
         var t = this.telefono;
         return '<li>' +
                n + ' ' + c + " (" + g + ") " + t +
                '</li>';
      }
}

function Rubrica() {
   this.voci = [];
   this.gruppi = [];
   this.inizializza =
      function(nomeFile) {
         var doc = caricaXML(nomeFile);
         var l = doc.getElementsByTagName("voce");
         for (i = 0; i < l.length; i++) {
            var v = new Voce();
            v.inizializza(l[i]);
            this.voci.push(v);
            var j = 0;
            while ((j < this.gruppi.length) &&
                   (this.gruppi[j].gruppo != v.gruppo)) {
               j++;
            }
            if (j == this.gruppi.length) {
               this.gruppi.push(v.gruppo);
            }
         }
      }
   this.creaSelect =
      function() {
         var s = "";
         s += '<option value="" selected="selected">' +
              'Seleziona un gruppo</option>'
         var l = r.gruppi;
         for (i in l) {
            s += '<option value="' + l[i] + '">' +
                 l[i] + '</option>';
         }
         return s;
   }
   this.cercaGruppo =
```

```
      function(g) {
         var l = [];
         for (i in this.voci) {
            if (this.voci[i].gruppo == g) {
               l.push(this.voci[i]);
            }
         }
         return l;
      }
   this.cercaNomeCognome =
      function(n, c) {
         var l = [];
         for (i in this.voci) {
            if ((this.voci[i].nome == n) ||
                (this.voci[i].cognome == c)) {
               l.push(this.voci[i]);
            }
         }
         return l;
      }
   this.visualizza =
      function(l) {
         var s = "";
         for (i in l) {
            s += l[i].visualizza();
         }
         return s;
      }
}

function caricaXML(nomeFile) {
   var xmlhttp;
   if (window.XMLHttpRequest) {
      // IE7+, Firefox, Chrome, Opera, Safari
      xmlhttp = new XMLHttpRequest();
   } else {
      // IE6, IE5
      xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
   }
   xmlhttp.open("GET", nomeFile, false);
   xmlhttp.send();
```

```
   return xmlhttp.responseXML;
}

function cercaPerGruppo() {
   var s = document.getElementById("selectGruppo");
   var i = 0;
   while ((i < s.options.length) &&
           !s.options[i].selected) {
      i++;
   }
   var l = r.cercaGruppo(s.options[i].value);
   var nodo = document.getElementById("elenco");
   if (l == "") {
      nodo.innerHTML = null;
      alert("Nessuna voce trovata");
   } else {
      nodo.innerHTML = r.visualizza(l);
   }
}

function cercaPerNomeCognome() {
   var n = document.getElementById("nome").value;
   var c = document.getElementById("cognome").value;
   var l = r.cercaNomeCognome(n, c);
   var nodo = document.getElementById("elenco");
   if (l == "") {
      nodo.innerHTML = null;
      alert("Nessuna voce trovata");
   } else {
      nodo.innerHTML = r.visualizza(l);
   }
}

function reimposta() {
   var nodo = document.getElementById("elenco");
   nodo.innerHTML = null;
}

var r;

function inizializza() {
   var p1 = document.getElementById("cercaPerGruppo");
```

```
   p1.onclick = cercaPerGruppo;
   var p2 = document.getElementById("cercaPerNomeCognome");
   p2.onclick = cercaPerNomeCognome;
   var p3 = document.getElementById("reimposta");
   p3.onclick = reimposta;
   r = new Rubrica();
   r.inizializza("Rubrica.xml");
   var nodo = document.getElementById("selectGruppo");
   nodo.innerHTML = r.creaSelect();
}

window.onload = inizializza;
```

20 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.

20.1 Parole riservate

```
false, true, null

this, new

var, function

return, break

if, else

switch, case, default

while, for, in
```

20.2 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[12]
             | ! | " | # | $ | % | & | ' | ( | )
             | * | + | , | - | . | / | : | ; | <
             | = | > | ? | @ | [ | \ | ] | ^ | _
             | ` | { | | | } | ~
```

12 Carattere di spaziatura.

20.3 Identificatore

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

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

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

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

20.4 Costante

```
<Costante>       ::= <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>
```

20.5 Espressione

```
<Espressione> ::= <Costante>
                |  <Identificatore>
                |  (<Espressione>)
                |  <UnOp> <Espressione>
                |  <Espressione> <BinOp> <Espressione>
                |  this
                |  <Espressione>.<Identificatore>
                |  <Espressione>.<Chiamata>
                |  <Espressione>[<Espressione>]
                |  <Chiamata>
                |  new <Identificatore>()
                |  new <Identificatore>(<Espressioni>)
                |  []
                |  [<Espressioni>]
                |  {}
                |  {<Coppie>}

<UnOp>        ::= -  | +  | !
<BinOP>       ::= -  | +  | *  | /  | %
                |  && | || |
                |  <  | <= | >  | >= | == | !=

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

<Chiamata>    ::= <Identificatore>()
                |  <Identificatore>(<Espressioni>)

<Coppie>      ::= <Coppia>
                |  <Coppia>, <Coppie>
<Coppia>      ::= <Espressione>: <Espressione>
```

20.6 Programma, dichiarazione, comando, blocco

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

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

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

<Dichiarazione>    ::= <DicVar>
                     |  <DicFun>
                     |  <DicCostr>

<ComandoSemplice>  ::= <Assegnamento>
                     |  <Invocazione>
                     |  <Return>
                     |  <Break>

<ComandoComposto>  ::= <Blocco>
                     |  <If>
                     |  <Switch>
                     |  <For>
                     |  <While>

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

20.7 Dichiarazione

```
<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>
```

20.8 Comando semplice

```
<Assegnabile>  ::= <Identificatore>
                 |  <Assegnabile>[<Espressione>]
                 |  this.<Assegnabile>
                 |  <Identificatore>.<Assegnabile>

<Assegnamento> ::= <Assegnabile> = <Aspressione>;
                 |  <Assegnabile> += <Aspressione>;
                 |  <Assegnabile> -= <Espressione>;
                 |  <Assegnabile> *= <Espressione>;
                 |  <Assegnabile> /= <Espressione>;
                 |  <Assegnabile> %= <Espressione>;
                 |  <Assegnabile>++;
                 |  <Assegnabile>--;

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

<Return>       ::= return <Espressione>;

<Break>        ::= break;
```

20.9 Comando composto

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

<Alternativa> ::= case <Costante>: <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>
```

Indice analitico