Esame di stato per l'abilitazione alla professione di Ingegnere
24 Giugno 2003
Tema di Informatica

In un ambiente di rete eterogeneo, si è deciso di utilizzare come formato di scambio delle informazioni un metalinguaggio di *markup* basato sul testo. Tale metalinguaggio è descritto da una grammatica G = {S, VN, VT, P }, dove S è lo scopo, VN l'insieme dei simboli non terminali, VT l'alfabeto terminale e P l'insieme delle regole di produzione. Più precisamente, l'insieme VN comprende i meta-simboli:

VN = { *Document*, *Element*, *Name*, *AttributeList*, *EValue*, *ElementList*, *Text*, *Attribute*}

di cui il primo, *Document*, è lo scopo, VT = { A...Z, a...z, 0...9, <, >, /, =, " } e le regole di produzione sono quelle di seguito specificate in formato BNF:

```
Document      ::=  Element
Element       ::=  < Name AttributeList > EValue </ Name >
EValue        ::=  ElementList | Text
ElementList   ::=  Element ElementList | Element
AttributeList ::=  Attribute AttributeList | Attribute | ε
Attribute     ::=  Name = " Text "
Name          ::=  identifier
Text          ::=  string
```

dove *identifier* rappresenta un identificatore secondo l'usuale sintassi di linguaggi come C o Java, *string* è una sequenza di caratteri dell'alfabeto VT diversi da <, >, " ed infine ε denota la stringa vuota.

Vi è, inoltre, un vincolo semantico: in ogni frase lecita del linguaggio, i due *Name* che compaiono nella definizione di un certo *Element* devono essere uguali fra loro, ossia il *Name* che compare nell'apertura di un dato elemento deve essere uguale a quello che compare nella chiusura di tale elemento. Come esempio di documento valido, si consideri il seguente frammento di codice:

```
<Radice attributo="valore">
   <Elemento1>
        <Sottoelemento>V1</Sottoelemento>
   </Elemento1>
   <Elemento2>V2</Elemento2>
</Radice>
```

L'obiettivo del progetto è di realizzare un riconoscitore / valutatore per il linguaggio denotato da questa grammatica. Tale riconoscitore / valutatore deve essere composto da tre moduli:

1. un analizzatore lessicale (Lexer)
2. un analizzatore sintattico-semantico (Parser)
3. un valutatore che costruisca un modello di documento (DOM – Document Object Model)

1. Progettazione del Lexer

Si supponga che esista un modulo Tokenizer che consenta di suddividere il documento sorgente in parole (dette "token") significativi. Tale componente dispone per ipotesi di due funzioni:

`boolean readToken()` – legge il prossimo token dal dispositivo d'ingresso e lo memorizza in una variabile di stato interna del Tokenizer stesso; i caratteri corrispondenti sul dispositivo d'ingresso sono consumati in modo irrevocabile. Tale lettura ignora i caratteri bianchi e quelli di fine linea, e restituisce `false` nel caso in cui si raggiunga la fine del file o `true` in caso di successo.

`string getToken()` – restituisce l'ultimo token letto da `readToken()`

La funzione `readToken` riconosce (e la funzione `getToken` restituisce) come token sia le entità *string* definite come sopra, sia i singoli caratteri dell'alfabeto terminale <, >, /, =, ".

È lecito supporre che il componente in questione possa essere costruito, *a scelta del candidato,* a partire o da uno stream, o da un file, o da una qualunque altra sorgente di dati (da specificare chiaramente insieme alle modalità di utilizzo ipotizzate).

Si progetti il componente Lexer in modo che esponga le tre funzioni:

`boolean readNextToken()` – fa avanzare il Lexer leggendo il prossimo token

`string getNextToken()` – restituisce l'ultimo token letto

`string getNextTokenType()` – fornisce il tipo (cioè la categoria lessicale) dell'ultimo di token letto, secondo le seguente tabella:

token letto	TokenType
<	`OpenStartTag`
</	`OpenEndTag`
>	`CloseTag`
"	`DoubleQuote`
=	`Equal`
identifier	`Identifier`
string	`GenericString`

A questo proposito, si noti che il Tokenizer riconosce come *string* le sequenze di caratteri racchiuse fra un `CloseTag` e un `OpenStartTag` o `OpenEndTag`.

2. Progettazione del Parser

Partendo dal componente Lexer di cui al punto 1, si progetti un Parser che verifichi la correttezza sintattica e semantica del documento fornito in input.

Quando il Parser riconosce l'inizio o la fine di un elemento, o la presenza di un attributo, o una stringa contenuta dentro un elemento, esso deve invocare una delle seguenti funzioni che incapsulano le relative *azioni semantiche* – ossia, svolgono l'azione desiderata per tale situazione.
(NB: nel caso si adotti per l'implementazione un linguaggio a oggetti, si suppongono definite in una interfaccia, mentre nel caso si adotti un linguaggio imperativo tradizionale si suppongono definite in un modulo o file di header o altra struttura idonea disponibile)

```
void StartElement(string name) – Apertura di un elemento
void Attribute(string name, string value) – Attributo
void EndElement(string name) – Chiusura di un elemento
void TextString(string value) – Testo contenuto in un elemento
```

Ad esempio, nel caso in cui venga analizzato il documento riportato precedentemente, il Parser effettuerà nell'ordine le seguenti chiamate:

```
StartElement("Radice")
Attribute("attributo", "valore")
StartElement("Elemento1")
StartElement("SottoElemento")
TextString("V1")
EndElement("SottoElemento")
EndElement("Elemento1")
StartElement("Elemento2")
TextString("V2")
EndElement("Elemento2")
EndElement("Radice")
```

Durante lo svolgimento del progetto del Parser, si discuta brevemente un possibile modo per la gestione dei possibili errori di analisi.

3. Progettazione del costruttore di DOM

L'implementazione delle funzioni descritte al punto 2 deve consentire di creare in memoria un modello ad albero del documento analizzato: a questo fine si deve innanzitutto progettare una tassonomia di classi o strutture adatte a costruire l'albero in memoria e, basandosi su questa tassonomia, implementare i metodi di cui sopra.
Per la progettazione dei componenti di cui ai punti 1,2 e 3 si segua la seguente traccia:

a. Definire i requisiti e l'architettura logico-funzionale del modulo
b. Descrivere i componenti necessari alla costruzione modulo in questione evidenziando le funzionalità offerte agli altri componenti del sistema e le possibili interazioni
c. Codificare il modulo descritto tramite linguaggi di alto livello come C, C++, Java, C# possibilmente utilizzando un approccio Object-Oriented o comunque tale da assicurare alla soluzione le necessarie modularità, manutenibilità e estendibilità incrementale