Introduzione: XML e xsML

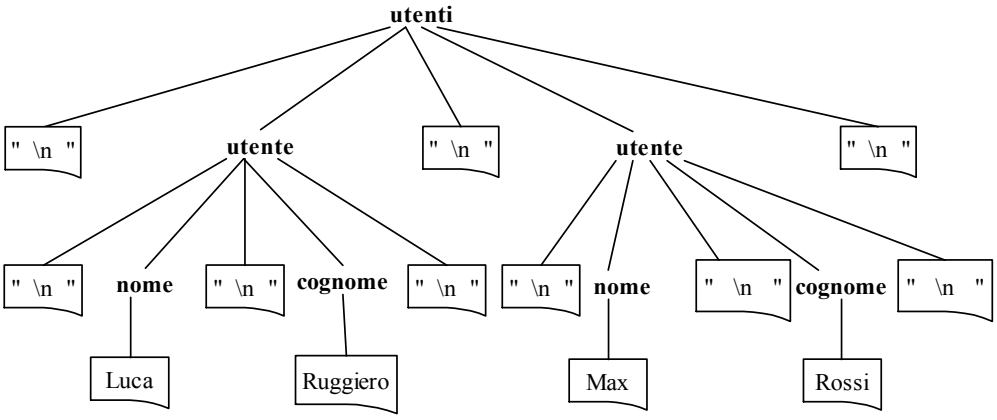
Il linguaggio *XML*, acronimo di *eXtensible Markup Language*, ovvero *linguaggio di marcatura estensibile*, è un metalinguaggio creato e gestito dal World Wide Web Consortium (W3C). È una semplificazione e adattamento dell'SGML, da cui è nato nel 1998. Rispetto al più ben noto HTML, l'XML ha uno scopo ben diverso: mentre il primo è un linguaggio per la specifica della struttura e della visualizzazione di pagine Web, il secondo è un linguaggio utile alla strutturazione e allo scambio dei dati.

Un esempio di file XML è il seguente:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<utenti>
    <utente>
        <nome>Luca</nome>
        <cognome>Ruggiero</cognome>
    </utente>
    <utente>
        <nome>Max</nome>
        <cognome>Rossi</cognome>
    </utente>
</utenti>
```

La prima riga indica la versione di XML in uso e specifica la codifica ISO per la corretta interpretazione dei dati. La struttura vera e propria dell'XML è composta dai *tag* (o *elementi*) e dal loro contenuto testuale. Un tag viene aperto (*start-tag*) mediante una coppia di parentesi angolari, (ad esempio `<utenti>`). La stringa che segue il carattere < costituisce il nome del tag (nell'esempio, `utenti`). Ogni tag deve essere chiuso (*end-tag*) mediante un'analoga coppia di parentesi angolari in cui il carattere slash (/) segue la parentesi angolare < (nell'esempio `</utenti>`). Il nome del tag chiuso deve corrispondere a quello del tag aperto.

Un documento XML può essere rappresentato come un albero, ovvero una struttura gerarchica in cui ogni elemento (detto nodo) dell'albero è un tag e un arco collega due nodi se il tag figlio è annidato all'interno del tag padre. L'esempio precedente può essere rappresentato mediante l'albero seguente:



dove in grassetto sono specificati i nomi dei tag e all'interno dei rettangolo sono specificate le porzioni di testo contenute all'interno del tag che le racchiude. Ad esempio, per il primo tag `nome` è specificato come contenuto la stringa "Luca"; ancora, il primo tag `utente` ha tre stringhe di contenuto: il testo compreso tra lo start-tag `<utente>` e lo start-tag `<nome>`, il testo compreso tra l'end-tag `</nome>` e lo start-tag `<cognome>` e il testo compreso tra l'end-tag `<cognome>` e l'end-tag `</utente>`. Notiamo che nell'esempio queste tre stringhe di testo contengono in realtà solo spazi e

caratteri di a capo (‘\n’), sebbene possano contenere qualsiasi tipo di carattere che non sia interpretato in modo speciale (ad esempio, un < che indica l’apertura di un nuovo tag).

Per ogni tag, è possibile specificare nello start-tag uno o più attributi, seguendo il formato nome=”valore”. Ad esempio:

```
<utente id=”mr10” eta=”20”>
    <nome>Mario</nome>
    <cognome>Rossi</cognome>
</utente>
```

dove specifichiamo che l’utente in questione ha un id uguale a `mr10` e un’età pari a 20. Infine notiamo che possiamo specificare commenti secondo il seguente formato:

```
<!-- questo e’ un commento -->
```

Un documento XML viene detto ben formato se, tra le altre cose, contiene un solo tag radice (nell’esempio, `<utenti>`) e se i tag sono sempre aperti e chiusi e correttamente annidati. Non entreremo qui in ulteriori dettagli delle specifiche del linguaggio XML. Per approfondimenti sull’XML, visitate le pagine: http://it.wikipedia.org/wiki/Xml e http://en.wikipedia.org/wiki/XML.

Il linguaggio *xsML* (*extra small Markup Language*), oggetto del progetto del laboratorio di programmazione, è un sottoinsieme del linguaggio XML che permette di specificare:

- start-tag e end-tag (es. rispettivamente **`<utente>`** e **`</utente>`**);
- attributi dei tag (es. `<utente` **`attributo=”valore”`**`>`);
- contenuto di un tag (tutto il testo racchiuso tra uno start-tag e/o un end-tag (es. il testo in grassetto: `<utente>`**`testo`**`<nome>`**`Mario Rossi`**`</nome>`**`bla bla`**`</utente>`).

Scopo del progetto è quello di realizzare un analizzatore lessicale e sintattico (*parser*) di documenti xsML e di implementare funzioni per la gestione e il trattamento di tali documenti.

Primo Modulo: Analizzatore lessicale

Al livello più basso, un documento è una sequenza di caratteri. Quello dei caratteri non è però il livello migliore da cui partire nell’analisi della struttura di un documento. Infatti, è più comodo e facile partire da unità atomiche elementari formate da blocchi di caratteri, i cosiddetti *token*. La forma e il tipi dei token in cui può essere decomposto un documento dipendono dal suo tipo; ad esempio, se il documento contiene un programma, i token dipenderà dal linguaggio in cui è scritto il programma. La fase di analisi che suddivide un documento nei token che lo compongono è detta *analisi lessicale*.

Il primo modulo del progetto implementa l’analizzatore lessicale dei documenti xsML.

Per comprendere meglio cos’è un token, prendiamo il caso di un testo scritto in linguaggio naturale. Se vogliamo analizzarne la struttura, raramente il livello più basso da cui cominciamo è quello dei singoli caratteri. Più comunemente, il livello più basso che individuiamo è quello della parole. Parole che vengono poi messe insieme a formare frasi, frasi che a loro volta sono parti di paragrafi, che a loro volta possono formare capitoli, e così via, a seconda della complessità del documento che si sta analizzando. Anche senza entrare nel merito della correttezza sintattica/semantica delle sequenze di parole che compongono il testo, già a livello delle parole possiamo cominciare ad analizzare la correttezza del documento verificando l’appartenenza delle parole al lessico della lingua in cui è scritto il documento. Si osservi però che, per comprendere l’organizzazione delle parole in frasi, non può essere trascurata la punteggiatura. Per questo motivo, le strutture atomiche da cui partiamo nell’analisi del testo, i cosiddetti *token*, saranno sia le parole che i simboli di punteggiatura. In realtà, nel caso del linguaggio naturale le cose sono anche più complicate a causa dalle espressioni atomiche composte da più parole, le cosiddette polirematiche, come ad esempio

"capro espiatorio"; espressioni che devono essere viste come un unico token e non come la sequenza dei token delle singole parole. Di conseguenza, nel caso del linguaggio naturale la decomposizione in token dell'input, o *tokenizzazione*, non è immediata, visto che non può basarsi solo sulla decomposizione della sequenza dei caratteri del testo fornita dagli spazi tra le parole e dalla punteggiatura. Cosa che invece possiamo fare nel caso dei linguaggi di programmazione, e in particolare di xsML.

Le regole della suddivisione in token

Per capire quali sono i token da individuare, osserviamo che, per quanto riguarda i tag:

- in uno start-tag, il nome dell'elemento aperto dal tag deve seguire immediatamente il carattere <. Analogamente, in un end-tag, il nome dell'elemento chiuso dal tag deve seguire immediatamente la coppia di caratteri </. Quindi, in pratica,
  - il blocco composto da <*nome-elemento* è un unico token, che chiameremo `START_TAG`, al quale è associata la stringa corrispondente a *nome-elemento*;
  - il blocco composto da </*nome-elemento* è un unico token, che chiameremo `END_TAG`, al quale è associata la stringa corrispondente a *nome-elemento*.
- In xsML, un nome può contenere esclusivamente
  - caratteri alfanumerici
  - il carattere _ (underscore)
  - il carattere - (trattino)
  - il carattere . (punto)

  inoltre, il nome deve necessariamente cominciare con un carattere alfabetico o con _.
- All'interno dei tag aperti, gli spazi non sono significativi (a parte quelli contenuti all'interno delle stringhe dei valori degli attributi). Più precisamente, tra il nome dell'elemento e quello del primo attributo (se presente) ci deve essere almeno uno spazio, ma ce ne possono essere un numero arbitrario. Analogamente, ci possono essere un numero arbitari di spazi tra le coppie nome-valore degli attributi associati all'elemento, o prima del carattere > che chiude il tag. Infine, prima e dopo il simbolo = che separa il nome dal valore di un attributo, possono esserci un numero arbitrario di spazi, ma anche non esserci alcuno spazio (se il simbolo = è attaccato al nome o al valore in ogni caso i due token possono essere separati senza ambiguità). Gli spazi bianchi non formano o entrano a far parte dei token, ma servono solo a separare i token in cui si decompone un tag. Inoltre, con spazio bianco si intende sia il semplice carattere spazio che i caratteri di tabulazione o di fine linea.

In base alla precedenti considerazioni, il tag

```
<documento id = ”1” autore=”Mario Rossi” >
```

va separato nei seguenti token

```
<documento
id
=
”1”
autore
=
”Mario Rossi”
>
```

anche se vedremo che l'analizzatore lessicale, oltre a dividere l'input in token dovrà restituire il tipo del token individuato e l'informazione rilevante ad esso associata. Ad esempio, in `”Mario Rossi”`

le virgolette che delimitano la stringa del valore dovranno essere eliminate, oppure in `<documento`, anziché restituire l'intera stringa contenente anche il carattere < si dovrà dire che il token è uno start-tag dell'elemento di nome `documento`. Maggiori dettagli li vedremo quando specificheremo più esattamente la rappresentazione dei token e le informazioni ad essi associate.

Al contrario di quanto accade all'interno dei tag, nel contenuto degli elementi, gli spazi bianchi sono significativi e non possono essere ignorati o cancellati. In particolare, si osservi che:

- Se l'elemento non contiene altri elementi, il contenuto dell'elemento è la sequenza di caratteri che va dal > che chiude lo start-tag dell'elemento, al < che apre l'end-tag dell'elemento. In questo caso, l'intero contenuto dell'elemento forma un unico blocco, ovvero, un unico token di tipo stringa. Ad esempio, nell'elemento
```
<indirizzo>
    Via Salaria, 113 - 00198 Roma
  </indirizzo>
```
tutta la seguente stringa C
```
"\n    Via Salaria, 113 - 00198 Roma\n  "
```
è un unico. Si noti che anche gli spazi iniziali e finali (inclusi i new-line) fanno parte del token.
- Se l'elemento contiene altri elementi, in generale, il contenuto si compone di una sequenza alternata di stringhe ed elementi, elementi che a loro volta saranno decomposti in token. In pratica, tutto il testo compreso tra il > che chiude un tag e il < che apre il tag successivo forma un unico token. Ad esempio, in
```
<p>all <b>work</b> and <i>no play</i> make Jack a dull boy</p>
```
il primo token del contenuto dell'elemento `p` è la stringa `"all "`. Quindi, dopo la decomposizione in token dell'elemento `b`, si ha la stringa `" and "` e, dopo la decomposizione in token dell'elemento `i`, la stringa `"make Jack a dull boy"`. Si osservi comunque che se due elementi sono attaccati, tra di essi non c'è nessun token di tipo stringa. Ad esempio, in
```
<indirizzo><via>Via Salaria</via><civico>113</civico></indirizzo>
```
il token successivo al > che chiude lo start-tag di `indirizzo` è il token `<via` che apre lo start-tag di `via`. Gli unici token stringa nell'esempio sono i contenuti degli elementi `via` e `civico`, ovvero le stringhe `"Via Salaria"` e `"113"`.

In XML, e quindi anche in xsML, è esplicitamente specificato che il simbolo < può apparire solo come apertura di uno start-tag. In particolare, < non può occorrere direttamente in nessuna stringa, né nel valore di un attributo, né nel contenuto di un elemento (del resto in questo caso verrebbe interpretato immediatamente come inizio di un tag). Per poter inserire < all'interno di una tag occorre utilizzare una cosiddetta entità, ovvero, un nome preceduto dal carattere `&` e seguito da `;`. In particolare, per < occorre scrivere `<`. Analogo discorso vale per >, per il quale occorre usare l'entità `>`. Nella nostra analisi noi non riserveremo alcun trattamento particolare alle entità, ma le tratteremo come tutte le altre stringhe. Rimane comunque il vincolo sull'uso di < e >.

Per quanto riguarda i caratteri virgolette doppie " e semplici ', si osservi che all'interno di un tag, queste hanno un significato particolare e servono a delimitare i valori degli attributi e (si noti bene) non sono parte della stringa del valore. Però, è possibile inserire singoli apici, quando il valore è delimitato da doppi apici (esempio, `"l'albero dalle 'foglie' rosse"`) o inserire i doppi apici quando il valore è delimitato da singoli apici (esempio, `'i "token" sono'`). Anche per gli apici esistono delle entità che si possono usare nei casi più complicati (`&quot;` per i doppi apici e `&apos;` per quelli singoli), ma il loro uso non ha rilievo per l'analisi lessicale. Invece, tutti gli apici possono essere usati senza problemi nel contenuto degli elementi, visto che queste sono delimitate da tag.

Tipi di token

Riassumendo quanto visto precedentemente, per l'analisi lessicale dei documenti xsML, si possono individuare i seguenti tipi di token:

Tipo Token	Forma del Token
`OPEN_START_TAG`	<*nome-elemento* dove *nome-elemento* è un nome valido (di elemento)
`OPEN_END_TAG`	</*nome-elemento* dove *nome-elemento* è un nome valido (di elemento)
`CLOSE_TAG`	>
`EQUAL`	=
`NAME`	nome valido (di attributo)
`STRING`	Stringa di caratteri ▪ valore di un attributo ▪ frammento di testo contenuto in un elemento

dove un nome valido è una sequenza di caratteri che può contenere esclusivamente caratteri alfanumerici, o il carattere _ (underscore), o il carattere - (trattino), o il carattere . (punto), e che deve necessariamente cominciare con un carattere alfabetico o con _.

Tutti i precedenti tipi corrispondono a costanti positive definite all'interno del file *xsml_lex.h* fornito insieme alle specifiche del modulo.

Partendo dalla posizione corrente, l'analizzatore lessicale deve leggere la parte di documento corrispondente al successivo token in input (ignorando, nel caso ci si trovi all'interno di un tag, eventuali spazi bianchi che lo precedono) e restituire il tipo del token riconosciuto, o errore (vedi sotto) nel caso di token non valido. In alcuni casi, oltre al tipo del token, dovrà anche restituire:

- la stringa del nome dell'elemento aperto o chiuso dal tag, nel caso di `OPEN_START_TAG` o `OPEN_END_TAG` token, oppure la stringa del nome letto, nel caso di token di tipo `NAME`;
- la stringa letta (priva degli apici, nel caso di valore di un attributo), nel caso di token di tipo `STRING`.

Inizio e fine di un token

Quando l'analizzatore lessicale scandisce l'input, il primo carattere che trova (diverso da spazio se all'interno di un tag) gli permette di individuare il tipo del successivo token presente in input. Ad esempio, un carattere alfabetico indica che il successivo token sarà un `NAME`, gli apici che sarà una stringa, e così via.

Per quanto riguarda la fine del token, in alcuni casi, la si può individuare riconoscendo il suo ultimo carattere. Ad esempio, questa considerazione si applica banalmente ai token formati da un solo carattere, ma vale anche per le stringhe racchiuse da apici, dove il termine del token è segnalato dal raggiungimento dell'apice finale. In altri casi, invece, la fine del token può essere individuata solo dopo aver letto il primo carattere del token successivo. Ad esempio, nel caso della scansione della stringa contenuta nell'elemento `<elemento>contenuto</elemento>`, per riconoscere la fine del token `contenuto` occorre leggere il carattere `<` che appartiene al token `</elemento`. Quindi, se i caratteri vengono letti direttamente dal file del documento senza alcuna forma di bufferizzazione

(gestita esplicitamente dal modulo), alla fine della scansione del token `contenuto`, dopo la lettura di `<`, sullo stream di input si ha `/elemento`. Ora, se la successiva scansione di un token cominciasse da questo stato, si andrebbe a leggere `/` come primo carattere, individuando un errore in una sequenza in realtà corretta. Il punto è che il carattere `<` deve essere letto per due volte consecutive; in pratica, dopo la prima lettura occorre accorgersi che si è letto un carattere di troppo e cercare di ripristinare lo stato come se il carattere non fosse stato letto, garantendo così che la successiva acquisizione di un carattere dall'input legga nuovamente `<`.

La soluzione a questo problema non è unica. Una possibilità è che l'analizzatore lessicale gestisca esplicitamente una forma di *bufferizzazione*: il file viene letto a blocchi e ogni blocco sotto analisi viene caricato in un vettore, il buffer; la lettura all'interno di un blocco corrisponde pertanto ad avanzare l'indice della posizione corrente nel buffer e, quando ci si accorge di essere andati troppo avanti, ripristinare l'input corrisponde a far tornare indietro l'indice (ovviamente bisogna fare attenzione ai casi in cui ci si trovi ai limiti, inizio e fine, del blocco nel buffer). Se invece i caratteri vengono letti direttamente dal file del documento senza alcuna forma di bufferizzazione esplicita si può utilizzare la funzione `ungetc` della libreria standard del C che ha proprio il compito di riposizionare in testa allo stream di input il carattere che si vuole leggere per primo nella successiva lettura.

Posizione del token

Per poter eventualmente indicare la posizione in cui si verifica un errore nella struttura del documento (ad esempio, quando viene letto un token corretto, ma in base alla eregole della sintassi ci si accorge che il token non può occorrere in quella posizione del documento), insieme al tipo e all'eventuale stringa del nome o del testo associato al token, ad ogni token dell'input va anche associata la sua posizione nel documento, ovvero, il numero di riga e il numero della colonna del suo primo carattere.

Un esempio

Si consideri un documento xsML che contiene il seguente testo:

```
<radice>
    <documento id=”1”>
        <titolo>Titolo 1</titolo>
    </documento>
    <documento id=”2”>
        <titolo>Titolo 2</titolo>
    </documento>
</radice>
```

L'analizzatore lessicale deve fornire, nell'ordine, i token riportati nella tabella qui di seguito, tenendo conto che nella prima colonna riportiamo la posizione del token nel documento, nella seconda riportiamo il tipo del token e nella terza il nome o il testo associato al token, se presente (le virgolette nella stringa associata al token non fanno parte della stringa, servono solo per delimitarla, in accordo con la sintassi del C).

```
1,1     OPEN_START_TAG     “radice”
1,8     CLOSE_TAG
1,9     STRING             “\n    ”
2,5     OPEN_START_TAG     “documento”
2,16    NAME               “id”
2,18    EQUAL
2,19    STRING             “1”
2,22    CLOSE_TAG
2,23    STRING             “\n        ”
3,9     OPEN_START_TAG     “titolo”
3,16    CLOSE_TAG
3,17    STRING             “Titolo 1”
```

```
3,25     OPEN_CLOSE_TAG     "titolo"
3,33     CLOSE_TAG
3,34     STRING             "\n    "
4,5      OPEN_CLOSE_TAG     "documento"
4,16     CLOSE_TAG
4,17     STRING             "\n    "
5,5      OPEN_START_TAG     "documento"
5,16     NAME               "id"
5,18     EQUAL
5,19     STRING             "2"
5,22     CLOSE_TAG
5,23     STRING             "\n        "
6,9      OPEN_START_TAG     "titolo"
6,16     CLOSE_TAG
6,17     STRING             "Titolo 2"
6,25     OPEN_CLOSE_TAG     "titolo"
6,33     CLOSE_TAG
6,34     STRING             "\n    "
7,5      OPEN_CLOSE_TAG     "documento"
7,16     CLOSE_TAG
7,17     STRING             "\n"
8,1      OPEN_CLOSE_TAG     "radice"
8,9      CLOSE_TAG
```

Errori

L'analizzatore lessicale decompone l'input in token ma non controlla se questi si susseguono secondo le regole stabilite dalla sintassi del linguaggio. Ad esempio, uno start-tag potrebbe non essere correttamente chiuso da un end-tag dello stesso elemento. Situazioni di errore legate alla violazione di regole sintattiche di xsML saranno riconosciute e segnalate dall'analizzatore sintattico. Esistono però errori che corrispondono alla violazione delle regole di formazione dei token e quindi sono riconosciuti a livello lessicale. Ad esempio, nessun token può cominciare con una cifra decimale o con i caratteri `.` o `-`. Quindi, se l'analizzatore lessicale si trova davanti ad una situazione del genere, deve restituire un errore.

Nei casi in cui riconosca un errore lessicale, l'analizzatore dovrà restituire il codice di errore `ERR_INVALID_TOKEN`, e un token di tipo `ERROR` contenente la stringa con l'errore sino al carattere che causa l'errore (incluso) e la posizione di tale stringa all'interno del documento.

Un particolare caso di errore è quello in cui non si riesce a leggere nulla perché la prima cosa che si trova nello stream di input è l'EOF (end-of-file), oppure si arriva ad un EOF prima di riuscire a completare correttamente il token che si sta analizzando (ad esempio, dopo aver letto i doppi apici si comincia a leggere una stringa ma si arriva all'EOF prima di trovare i doppi apici di chiusura). In questi casi, oltre al token di tipo `ERROR` con le informazioni sul pezzo di token letto prima dell'EOF, l'analizzatore dovrà restituire il codice di errore `ERR_EOF`.

Funzioni da implementare

Si richiede l'implementazione nel file *xsml_lex.c* delle 3 funzioni seguenti, i cui prototipi sono specificati nel file *xsml_lex.h* assegnato insieme alle specifiche:

- `int get_next_token(lex_an *pla, token *ptk);`
  Legge il successivo token (nello stream dell'analizzatore lessicale `*pla`). Restituisce `0` se legge correttamente un token o un codice di errore nel caso in cui la funzione individui un errore lessicale, oppure nel caso in cui si verifichi un errore di lettura dallo stream di input o un qualsiasi altro errore.
  Nel caso di lettura di un token la funzione dovrà memorizzare in `*ptk` tutte le informazioni relative al token letto (se necessario, dovrà anche allocare la memoria per la stringa contenente il nome o il testo associati al token).

In caso di errore, la funzione dovrà restituire uno dei seguenti codici di errore (i codici di errore sono costanti negative):

`ERR_BAD_ARG`	Se i valori ricevuti negli argomenti non sono validi (ad esempio, hanno valore `NULL`, oppure i valori nei campi di `*pla` non sono corretti).
`ERR_INPUT`	Nel caso si verifichi un qualsiasi errore nella lettura dallo stream dell'analizzatore lessicale.
`ERR_MEM`	Nel caso si verifichi un problema di allocazione di memoria.
`ERR_EOF`	Nel caso si incontri l'EOF prima di aver completato la lettura del token.
`ERR_INVALID_TOKEN`	Nel caso si individui un token non valido.
`ERR_GENERIC`	Nel caso si individui una situazione di errore non classificabile tra le precedenti.

Inoltre, nel caso l'errore sia `ERR_EOF` o `ERR_INVALID_TOKEN` in `*ptk` dovranno essere memorizzate tutte le informazioni sulla stringa di input letta prima di incontrare EOF o sulla stringa di input riconosciuta come token non valido.

- `int new_lexical_analyzer(char *fname, lex_an *pla);`
  Crea un nuovo analizzatore lessicale, associandolo al file `fname`. La funzione restituisce `0` se l'analizzatore può essere aperto o un codice di errore (una costante negativa) altrimenti. In pratica, la funzione verifica se `fname` esiste e può essere aperto in lettura e, dopo l'apertura dello stream di input, inizializza lo stato dell'analizzatore lessicale e la posizione del cursore (riga e colonna).
  I codici di errore restituiti dalla funzione sono:

`ERR_BAD_ARG`	Se i valori ricevuti negli argomenti non sono validi (ad esempio, hanno valore `NULL`, o `fname` è la stringa vuota).
`ERR_FILE`	Nel caso lo stream di input non possa essere aperto in lettura (perché non esiste o non si hanno diritti sufficienti).
`ERR_GENERIC`	Nel caso si individui una situazione di errore non classificabile tra le precedenti.

- `int close_lexical_analyzer(lex_an *pla);`
  Chiude lo stream associato all'analizzatore lessicale e restituisce `0` o uno dei seguenti codici in caso di errore:

`ERR_BAD_ARG`	Se il valori ricevuto nell'argomento non è valido (ad esempio è `NULL`, oppure i valori nei campi di `*pla` non sono corretti).
`ERR_GENERIC`	Nel caso si individui una situazione di errore non classificabile tra le precedenti.

Strutture dati da utilizzare

Le strutture utilizzate dalle funzioni dell'analizzatore lessicale e visibili all'esterno del modulo sono definite nel file *xsml_lex.h*.

La struttura per la rappresentazione del token

```
/* rappresenta il token */
typedef struct
{
    /* tipo del token o codice di errore */
    token_type type;
```

```
    /* testo o nome associato al token */
    char *text;
    /* posizione del token nel documento */
    int riga, colonna;
} token;
```

contiene la stringa col nome o il testo associato al token, la riga e la colonna del token e il tipo del token.

Il tipo di un token è uno dei valori della enumerazione

```
/* enumerazione dei tipi di token dell'analizzatore lessicale /
typedef enum
{
    ERROR,           /* token non valido /
    OPEN_START_TAG,
    OPEN_END_TAG,
    CLOSE_TAG,
    EQUAL,
    NAME,
    STRING
} token_type;
```

In particolare, si noti il tipo `ERROR`, da utilizzare per restituire la parte di token letta prima del verificarsi di un errore.

L'analizzatore lessicale è definito mediante la struttura

```
/* struttura che rappresenta un file da analizzare /
typedef struct
{
    /* stream di input dell'analizzatore */
    FILE *file;
    /* stato dell'analizzatore */
    lex_state state;
    /* posizione del prossimo carattere dello stream */
    int riga, colonna;
} lex_an;
```

in cui il campo `file` contiene lo stream di input associato all'analizzatore, `riga` e `colonna` indicano la posizione del prossimo carattere dello stream di input, mentre `state` è uno dei due valori della enumerazione:

```
/* stati dell'analizzatore lessicale */
typedef enum
{
    LEX_STATE_CONTENT,
    LEX_STATE_TAG
} lex_state;
```

dove

`LEX_STATE_CONTENT`	indica che l'analizzatore lessicale sta esaminando il contenuto di un elemento
`LEX_STATE_TAG`	indica che l'analizzatore lessicale si trova all'interno di un tag

Il file *xsml_lex.h* contiene inoltre la definizione di tutte le costanti dei codici di errore. Si ricorda ancora una volta che tali costanti sono tutte negative.

Considerazioni conclusive

Le funzioni da implementare non devono ricevere altri input o produrre alcun output oltre quelli indicati nelle specifiche. In particolare, tali funzioni non dovranno leggere nulla dallo stream di

input, né stampare messaggi sullo stream di output. Le funzioni dovranno comunicare solo attraverso i parametri ricevuti al momento della chiamata e i valori di ritorno delle funzioni, in accordo con quanto indicato nelle specifiche.

Particolare cura dovrà essere messa nella individuazione degli errori. In caso di errore, la funzione non dovrà chiudere la computazione o stampare messaggi, ma solo restituire il codice di errore corrispondente all'errore individuato, avendo cura di liberare tutta l'eventuale memoria allocata e non più utilizzata.