Laboratorio di Programmazione 2 - Progetto 2006/2007 – Modulo 3

Il terzo modulo consiste nell'analizzatore sintattico per un documento xsML. L'analizzatore sintattico verifica se il documento è ben formato, ovvero, basandosi sull'analizzatore lessicale (implementato nel modulo 1) che suddivide il documento in token, verifica se la sequenza di token che compongono il documento corrisponde a un documento xsML corretto.

Le regole della sintassi di xsML sono riportate nella seguente grammatica (astratta).

tag ::= *start_tag blocco end_tag*
start_tag ::= <*nome attr_pair*$_1$ *attr_pair*$_2$ … *attr_pair*$_n$>
end_tag ::= </*nome*>
attr_pair ::= *nome* = *stringa*
blocco ::= *testo*$_0$ *tag*$_1$ *testo*$_1$ *tag*$_2$ … *testo*$_{n-1}$ *tag*$_n$ *testo*$_n$

Senza essere troppo formali nella definizione di grammatica (argomento che verrà affrontato in altri corsi), ogni nome rappresenta un pezzo di documento la cui struttura è definite per mezzo di una regola. Le regole sono indicate da coppie separate dal simbolo ::=. Sul lato sinistro della regola compare il nome della parte di documento che si sta definendo; sul lato destro viene data la struttura della parte che si sta definendo per composizione (concatenazione) di altre parti di documento (definite dalla grammatica) o di simboli che denotano i corrispondenti caratteri (in dettaglio, >, <, / e = indicano occorrenze dei caratteri corrispondenti).

Vediamo più in dettaglio le regole.

- Col nome *tag* indichiamo tutto ciò che è compreso tra una coppia start/end tag compresi. La regola corrispondente dice che è formato da uno *start_tag* seguito da un *blocco* e da un *end_tag*.
- Uno *start_tag* comincia con il carattere < seguito da un *nome* (il nome del tag), da una sequenza di n *attr_pair* (in particolare, se n=0 la sequenza è vuota) e dal carattere > che chiude il tag.
- Un *end_tag* è fomato dalla coppia di caratteri </ seguiti dal nome del tag e dal carattere > che chiude il tag.
- Le coppie di attributi sono formate da un *nome* seguito dal carattere = e da una *stringa.*
- Un *blocco* è formato da una sequenza di n *tag* preceduti e seguiti da un *testo*. In particolare, si osservi che un blocco comincia e termina sempre con un *testo* e che se n=0 il blocco si riduce solo a un testo.
- Nella grammatica sono omesse la definizione di nome, testo e stringa. Le definizioni di questi elementi sono già state date nelle specifiche dell'analizzatore lessicale.

Si osservi che la definizione delle parti che compongono un documento è ricorsiva. Infatti, dato che un *blocco* può contenere un *tag*, un tag può contenere altri *tag*. Questa ricorsione termina sicuramente perché prima o poi si arriva a un blocco che contiene solo testo. Questa struttura ricorsiva corrisponde a quella di un albero (già descritto in dettaglio nelle specifiche dei precedenti moduli). L'analizzatore sintattico, oltre a verificare la correttezza del documento ricevuto in ingresso, che essendo un albero sarà formato da un unico tag (detto appunto il tag radice), dovrà costruire il corrispondente albero.

La grammatica sopra descritta in realtà non fornisce tutte le regole che un documento xsML ben formato

deve rispettare. In particolare, oltre alle regole della grammatica occorre che lo *start_tag* e l'*end_tag* di un *tag* contengano lo stesso *nome*.

Per questo modulo del progetto, si deve implementare la seguente funzione:

```
int get_xsml_doc(char *, struct node**);
```

che riceve come input il nome del file contenente il documento e che deve creare l'albero corrispondente al documento letto restituendo il puntatore alla radice nel secondo argomento (si noti infatti che si tratta di un puntatore a un puntatore a nodo). La funzione restituisce il valore 0 se l'operazione termina con successo (documento corretto e nessun problema nella creazione dell'albero) o un codice di errore compreso tra uno dei seguenti:

`SAERR_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).
`SAERR_INPUT`	Nel caso si verifichi un qualsiasi errore nella lettura dallo stream dell'analizzatore lessicale.
`SAERR_MEM`	Nel caso si verifichi un problema di allocazione di memoria.
`SAERR_UNEXPECTED_EOF`	Nel caso si incontri l'EOF prima di aver completato la lettura del documento.
`SAERR_INVALID_TOKEN`	Nel caso si individui un token non valido.
`SAERR_UNEXPECTED_TOKEN`	Nel caso l'analizzatore sintattico riceva dall'analizzatore lessicale un token non valido nella posizione corrente.
`SAERR_GENERIC`	Nel caso si individui una situazione di errore non classificabile tra le precedenti.

In caso di errore, deve essere possibile recuperare il valore dell'ultimo token letto per mezzo di una chiamata alla funzione (anche questa da implementare):

```
token last_token(void);
```

Si osservi che `SAERR_UNEXPECTED_EOF` e `SAERR_INVALID_TOKEN` corrispondono a situazioni di errore individuate dall'analizzatore lessicale, e in questo caso è molto probabile che il token restituito dalla `last_token` non sia un token valido, ma la parte di stringa letta dall'analizzatore lessicale che ha portato all'errore. Nel caso di errore di tipo `SAERR_UNEXPECTED_TOKEN`, l'errore è sintattico e l'ultimo token letto, pur essendo uno dei token possibili, in base alle regole della grammatica non può apparire nella posizione in cui è stato trovato.

Funzioni per la scansione di attributi e nodi

Oltre all'implementazione dell'analizzatore sintattico, questo modulo prevede l'implementazione di alcune funzioni ausiliarie per la scansione dell'albero prodotto dall'analizzatore lessicale. In pratica, si tratta di istanziare al caso della lista degli attributi e al caso della lista dei figli di un tag l'iteratore

definito nel secondo modulo.

Per quanto riguarda la lista degli attributi di un nodo, si devono definire le funzioni:

1. `attr_it` **`attrlist_it`**`(struct node *);`
   che crea un iteratore sulla lista attributi del nodo ricevuto in ingresso;

2. `char *`**`next_attr`**`(attr_it);`
   che restituisce la stringa del nome del successivo attributo nella lista scandita dall'iteratore ricevuto in ingresso, se ci sono ancora attributi da visitare, o `NULL` se la lista rimasta da visitare è vuota.

Il tipo `attr_it` utilizzato per gli iteratori sulle liste di attributi, che in base a quanto specificato nel secondo modulo sono di tipo `list`, è un sinonimo del tipo `iterator` definito nel secondo modulo per implementare un iteratore su `list`.

```
typedef iterator attr_it;
```

che quanto riguarda la lista degli sttributi di un nodo, si devono definire le funzion

3. `node_it` **`nodelist_it`**`(struct node *);`
   che crea un iteratore sulla lista dei figli del nodo ricevuto in ingresso;

4. `struct node *`**`next_node`**`(node_it);`
   che restituisce il puntatore al successivo figlio nella lista scandita dall'iteratore ricevuto in ingresso, se ci sono ancora filgi da visitare, o `NULL` se la lista rimasta da visitare è vuota.

Il tipo `node_it` utilizzato per gli iteratori sulle liste di filgi, che in base a quanto specificato nel secondo modulo sono di tipo `list`, è un sinonimo del tipo `iterator` definito nel secondo modulo per implementare un iteratore su `list`.

```
typedef iterator node_it;
```