Realizzazione di un parser per il linguaggio XML

Vincenzo Scotti M63/693

22 febbraio 2017

Sommario

In questo documento viene realizzata una libreria per il parsing del linguaggio XML, in particolare adottando un subset della specifica ufficiale. Il processo di progettazione e' documentato nella sua integrita', dalla fase iniziale di specifica della grammatica fino al codice risultante, scritto in C++11. Nel paragrafo finale e' riportato un programma che fa uso della suddetta libreria, e vari casi di test.

Parte I
Introduzione

Si e' deciso di implementare un parser per il linguaggio XML. L'XML e' un metalinguaggio di markup, ovvero utilizzato per definire dei linguaggi di markup. Lo standard XML prevede quindi una descrizione sintattica della struttura del documento, tramite l'utilizzo di *tag* disposti in maniera gerarchica, con eventualmente attributi a corredo degli stessi. A questo segue una serie di costrutti per la specifica formale della semantica del documento, definendo quindi i *tag* ammissibili e come possono essere innestati tra loro.

Un documento XML e' detto:

- *well-formed* se il documento e' sintatticamente corretto;
- *valido* se il documento e' semanticamente corretto.

Il parser che viene qui realizzato si preoccupa solamente dell'analisi sintattica di un documento XML. Per questo motivo, per la realizzazione e' stato tenuto in considerazione solo un subset della specifica completa del linguaggio.

Le soluzioni al parsing di un documento XML possono essere distinte in due categorie:

- i parser *SAX* (Simple API for XML), che prevedono l'analisi del documento riga per riga. Questi tipi di parser offrono una API *event-driven*, consentendo all'utente di inserire una serie di *callback*, che verranno invocate al presentarsi di un evento di natura sintattica (ad esempio apertura di un tag, chiusura di un tag, etc...);
- i parser *DOM* (Document Object Model), che prevedono il parsing del documento nella sua integrita'. Per questo motivo il documento viene

prima caricato in memoria e successivamente viene analizzato. Il risultato di questa operazione e' una struttura dati gerarchica che rispetta le proprieta' strutturali del documento.

Si procede quindi all'implementazione di un parser DOM. Il parser effettuera' l'analisi sintattica del documento e restituira' un errore in caso di documento non *well-formed*, oppure una struttura dati in caso affermativo, col quale e' possibile esplorare la struttura del documento.

Parte II
Grammatica del linguaggio

Per una specifica completa della grammatica dell'XML si faccia riferimento a www.w3.org/TR/xml . Di seguito e' riportato il processo di specifica della grammatica per il solo subset del linguaggio inerente al lavoro svolto.

La descrizione e' riportata prima in termini informali, utilizzando il linguaggio naturale, per poi essere formalizzata utilizzando l'eBNF. A parte i casi descritti di seguito, gli identificatori sono da intendersi come simboli non terminali.

I simboli terminali sono espressi come:

- *#xNN* dove NN e' un numero, per indicare il valore di un byte in esadecimale;
- *[#xNN to #xNN]* per indicare un range di valori esadecimali.

Nel caso di simboli letterali di tipo stringa, essa sara' riportata interamente nella regola grammaticale.

1 Definizione della grammatica

Un documento XML e' *well-formed* se e' presente un prologo, opzionale, seguito dalla definizione dell'elemento radice del documento.

⟨*document*⟩ ::= ⟨*prologue*⟩? ⟨*element*⟩

Il prologo e' un tag speciale nel quale viene indicata la versione dello standard utilizzata nel documento. Allo stato attuale, l'unica versione consentita e' la *1.0*.

⟨*prologue*⟩ ::= ⟨*xml_decl*⟩ ⟨*maybe_space*⟩

⟨*xml_decl*⟩ ::= `<?xml` ⟨*version_info*⟩ ⟨*maybe_space*⟩ `?>`

⟨*version_info*⟩ ::= ⟨*space*⟩ `version` ⟨*equals*⟩ (`'1.0'` | `"1.0"`)

⟨*equals*⟩ ::= ⟨*maybe_space*⟩ `=` ⟨*maybe_space*⟩

Successivamente segue la definizione dell'elemento radice, che segue le regole sintattiche di tutti gli altri elementi nel documento. Per questo si introduce una regola generale che determina la struttura generica di un qualsiasi elemento.

Un elemento e' composto da un tag di apertura, che ne determina il nome, piu' una lista opzionale di attributi. L'elemento puo' essere immediatamente chiuso o puo' possedere un contenuto. Il contenuto dell'elemento puo' essere vuoto, composto da altri elementi o da una sequenza di caratteri. Ad ogni elemento non vuoto deve essere associato il rispettivo tag di chiusura.

⟨*element*⟩ ::= < ⟨*name*⟩ (⟨*space*⟩ ⟨*attribute*⟩)* ⟨*maybe_space*⟩
(/> | (> ⟨*content*⟩ </ ⟨*name*⟩ ⟨*maybe_space*⟩ >))

⟨*name*⟩ ::= (⟨*letter*⟩ | _ | :) (⟨*letter*⟩ | ⟨*digit*⟩ | . | : | - | _)*

⟨*attribute*⟩ ::= ⟨*name*⟩ ⟨*equals*⟩ ⟨*attribute_value*⟩

⟨*attribute_value*⟩ ::= (" (⟨*char*⟩ - ")* ") | (' (⟨*char*⟩ - ')* ')

⟨*content*⟩ ::= (⟨*maybe_space*⟩ ⟨*element*⟩)* | ⟨*chardata*⟩

Infine sono riportati i caratteri terminali, secondo la notazione sopra riportata.

⟨*char*⟩ ::= [#x00 to #x7E]

⟨*chardata*⟩ ::= (⟨*char*⟩ - <)*

⟨*letter*⟩ ::= [#x41 to #x7A]

⟨*digit*⟩ ::= [#x30 to #x39]

⟨*space*⟩ ::= (#x20 | #x9 | #xD | #xA)+

⟨*maybe_space*⟩ ::= ⟨*space*⟩?

2 Analisi della grammatica

Si noti come la regola ⟨*element*⟩ e' ricorsiva in quanto, durante la sua espansione, essa puo' essere nuovamente invocata tramite la regola ⟨*content*⟩.

Va inoltre aggiunto che, affinche' un elemento sia valido, il tag di chiusura deve presentare lo stesso nome del tag di apertura. Questa condizione non e' espressa nella grammatica eBNF riportata sopra. Infatti un elemento definito come

```
<a> ... contenuto ... </b>
```

risulta corretto secondo la grammatica specificata, ma non secondo le regole sintattiche dell'XML.

Questo vincolo sintattico introduce una regola che dipende dal *contesto*, trasformando la sintassi da *context-free* a *context-sensitive*, e non esprimibile in eBNF. Tuttavia, in fase implementativa, sara' possibile adottare le soluzioni

progettuali per grammatiche *context-free*, con delle lievi modifiche per tenere conto di questo vincolo aggiuntivo.

Un'altra importante caratteristica da verificare e' se il linguaggio $\in LL(1)$, ovvero se da qualsiasi regola di produzione e' possibile sapere con precisione quale altra regola espandere esaminando solo il token successivo.

Indichiamo formalmente con G una generica grammatica, e con N l'insieme dei simboli non terminali della grammatica.

$$G \in LL(1) \iff FIRST(A) \cap FIRST(B) = \emptyset, \forall A, B \in N$$

dove con $FIRST(A)$ si intende l'insieme dei primi token ottenuti per ogni espansione di A.

Considerando la grammatica sopra descritta, esaminiamo i simboli $\langle prologue \rangle$ e $\langle element \rangle$.

$$FIRST(prologue) = FIRST(xml\_decl) = \text{'<'}$$

$$FIRST(element) = \text{'<'}$$

$$FIRST(prologue) \cap FIRST(element) \neq \emptyset$$

Da questo si puo' concludere che la grammatica $\notin LL(1)$.

Una spiegazione intuitiva puo' essere data in questo modo: all'inizio dell'analisi del documento, quando il parser applichera' la regola $\langle document \rangle$, pur supponendo che il prossimo token sia "<", non e' possibile sapere se esso appartiene alla regola $\langle prologue \rangle$ o $\langle element \rangle$.

Parte III
Progettazione del parser

Un approccio comune per la progettazione di parser per grammatiche *context-free* e' quello dei parser *ricorsivi discendenti*.

Le regole della grammatica vengono tradotte come funzioni che si richiamano a vicenda, in maniera ricorsiva. Di fronte a piu' espansioni possibili per una determinata regola, si puo' procedere in due modi:

- con *lookahead*, se la grammatica $\in LL(k)$. Esso consiste nell'analizzare i prossimi k token per determinare univocamente quale espansione applicare;
- con *backtracking*, se le varie espansioni sono testate in sequenza, selezionando la prima che ha successo.

Nel nostro caso e' stato adottato un parser *ricorsivo discendente* con *backtracking*, con un'opportuna modifica per gestire correttamente la regola $\langle element \rangle$, che rende la grammatica non strettamente *context-free*.

3 Implementazione

La libreria e' stata implementata utilizzando il linguaggio C++, facendo riferimento allo standard versione 11.

Per una maggiore modularita' si e' fatto uso di un namespace, chiamato *xml*, per raggruppare le strutture dati e le API.

L'unica funzionalita' offerta dalla libreria, allo stato attuale, e' appunto quella di effettuare il parsing di un documento, a partire da un iteratore al carattere iniziale e finale della stringa.

```
template <typename It>
xml_doc parse(It &s, const It &e);
```

La sua implementazione e' discussa in seguito.

3.1 Strutture dati

Il documento finale sara' rappresentato nella seguente struttura dati:

```
class xml_doc
{
public:
        std::string version;
        xml_node *root;

        xml_doc() :
                root{nullptr}
        {

        }
};
```

La struttura memorizza la versione dell'xml utilizzata nel documento e un puntatore al nodo radice. Il nodo e' rappresentato utilizzando la seguente struttura dati:

```
class xml_node
{
public:
        std::string name;
        std::map<std::string, std::string> attributes;
        std::vector<xml_node *> children;
        std::string value;
};
```

Si noti come per ogni nodo viene memorizzato il suo nome, gli eventuali attributi, gli eventuali nodi figli oppure il valore del suo contenuto.

3.2 Parsers

Ogni regola grammaticale viene implementata utilizzando una funzione col seguente prototipo:

```
template <typename It>
TipoDiRitorno NomeRegola(It &start, const It &end, ...);
```

La funzione prende in ingresso due iteratori, uno che punto al carattere corrente e l'altro che punta alla fine della sequenza d'ingresso. Il tipo di ritorno e i parametri aggiuntivi dipenderanno dalla specifica regola che si sta implementando. Il primo iteratore, sia in caso di successo che di fallimento della regola, verra' fatto avanzare fino all'ultimo carattere riconosciuto valido.

In caso di fallimento la funzione lancera' un eccezione di tipo *parser_ error*; in questo modo il *backtracking* viene realizzato sfruttando il meccanismo di eccezioni del linguaggio. La funzione chiamante quindi, si preoccupera' di catturare l'eccezione e di provare la regola successiva, oppure lascera' propagare l'eccezione nel caso non ci siano altre regole da testare.

```
class parser_error : public std::exception
{
public:
        parser_error(const std::string &msg);

        virtual const char *what() const noexcept override;

        virtual const std::string error_string() const;

private:
        std::string err_string;
};
```

Per una maggiore chiarezza, i parser sono tutti racchiusi nel namespace *xml::parser*.

Si procede dunque con l'implementazione delle regole della grammatica.

3.2.1 char

⟨*char*⟩ ::= [#x00 to #x7E]

Per realizzare la regola ⟨*char*⟩ e' stato utilizzato direttamente il tipo nativo *char* del C++.

3.2.2 chardata

⟨*chardata*⟩ ::= (⟨*char*⟩ - <)*

La regola chardata si occupa di consumare il buffer in ingresso fino al carattere "<".

```
template <typename It>
std::string chardata(It &s, const It &e)
{
        auto pos = std::find(s, e, '<');

        s = pos;

        return std::string{s, pos};
}
```

3.2.3 letter, digit

⟨*letter*⟩ ::= [#x41 to #x7A]

⟨*digit*⟩ ::= [#x30 to #x39]

Le regole sono state implementate utilizzando rispettivamente le funzioni *std::isalpha*, *std::isdigit* e *std::isalnum* fornite dalla libreria standard del C++.

3.2.4 space, maybe_space

⟨*space*⟩ ::= (#x20 | #x9 | #xD | #xA)+

⟨*maybe_space*⟩ ::= ⟨*space*⟩?

La regola ⟨*space*⟩ controllera' la presenza di almeno un carattere spaziatore, e consumera' il buffer fino al primo carattere non spaziatore. La regola ⟨*maybe_space*⟩ e' analoga alla precedente, ma non genera un fallimento in alcun caso.

```
template <typename It>
void space(It &s, const It &e)
{
        if (s == e || !std::isspace(*s)) {
                throw parser_error{"space"};
        }

        do {
                s++;
        } while (s != e && std::isspace(*s));
}

template <typename It>
void maybe_space(It &s, const It &e)
{
        try {
                space(s, e);
        } catch (const parser_error &) {

        }
}
```

3.2.5 document

⟨*document*⟩ ::= ⟨*prologue*⟩? ⟨*element*⟩

Per implementare questa regola e' necessario prima provare a procedere con la regola ⟨*prologue*⟩, e in ogni caso procedere con la regola ⟨*element*⟩.

Trattandosi della regola top-level, il suo valore di ritorno sara' un *xml_doc*. La struttura sara' riempita sfruttando i valori di ritorno delle due regole espanse.

Il meccanismo di gestione delle eccezioni consente di implementare agilmente il *backtracking*. Infatti, il fallimento della regola ⟨*prologue*⟩ e' prontamente rilevato dal blocco catch, che agisce di conseguenza, ovvero in questo caso ignorando l'errore,, essendo il prologo opzionale.

```
template <typename It>
xml_doc document(It &s, const It &e)
{
        xml_doc ret;

        try {
                auto curr = s;

                ret = prologue(curr, e);

                s = curr;
        } catch (const parser_error &ex) {
                ret.version = "1.0";
        }

        ret.root = element(s, e);

        return ret;
}
```

3.2.6 prologue

⟨*prologue*⟩ ::= ⟨*xml_decl*⟩ ⟨*maybe_space*⟩

Il prologo si occupa di leggere la versione dell'XML utilizzata nel documento, e di consumare eventuali spazi.

```
template <typename It>
xml_doc prologue(It &s, const It &e)
{
        xml_doc doc;

        doc.version = xml_decl(s, e);

        maybe_space(s, e);

        return doc;
}
```

3.2.7 xml_decl

⟨*xml_decl*⟩ ::= `<?xml` ⟨*version_info*⟩ ⟨*maybe_space*⟩ `?>`

La dichiarazione XML consiste nell'apertura di un tag, seguita dal numero di versione e da eventuali spazi prima del tag di chiusura.

```
template <typename It>
std::string xml_decl(It &s, const It &e)
{
        std::string version;
        auto first_match = "<?xml";
        auto last_match = "?>";

        match_string(s, e, first_match);
        version = version_info(s, e);
        maybe_space(s, e);
        match_string(s, e, last_match);

        return version;
}
```

Si noti come i simboli terminali sono consumati utilizzando le funzioni *match_string* e *match_char*, descritte in seguito.

3.2.8 version_info

⟨*version_info*⟩ ::= ⟨*space*⟩ `version` ⟨*equals*⟩ (`'1.0'` | `"1.0"`)

Questa funzione si occupa di consumare l'attributo *version* con l'annesso valore, che puo' essere pari solo ad *1.0*, come richiesto da specifica.

```
template <typename It>
std::string version_info(It &s, const It &e)
{
        space(s, e);
        match_string(s, e, "version");
        equals(s, e);

        auto curr = s;

        try {
                match_string(curr, e, "'1.0'");
        } catch (const parser_error &) {
                curr = s;
                match_string(curr, e, "\"1.0\"");
        }

        s = curr;

        return "1.0";
}
```

3.2.9 equals

⟨*equals*⟩ ::= ⟨*maybe_space*⟩ `=` ⟨*maybe_space*⟩

Questa regola consuma il simbolo terminale "=", circondato da eventuali spazi.

```
template <typename It>
void equals(It &s, const It &e)
{
        maybe_space(s, e);
        match_char(s, e, '=');
        maybe_space(s, e);
}
```

3.2.10 element, content

⟨*element*⟩ ::= < ⟨*name*⟩ (⟨*space*⟩ ⟨*attribute*⟩)* ⟨*maybe_space*⟩
(/> | (> ⟨*content*⟩ </ ⟨*name*⟩ ⟨*maybe_space*⟩ >))

⟨*content*⟩ ::= (⟨*maybe_space*⟩ ⟨*element*⟩)* | ⟨*chardata*⟩

Questa e' la funzione piu' articolata del parser. Le due regole, per facilitarne l'implementazione, sono realizzate entrambe in una unica funzione.

La logica di questa funzione si occupa inoltre di controllare la congruenza tra il nome del tag di apertura e quello di chiusura.

Il valore di ritorno e' di tipo *xml_node **, ovvero un puntatore ad un nodo, completo di eventuali figli ed attributi. I nodi figlio sono consumati richiamando ricorsivamente la funzione stessa.

```
template <typename It>
xml_node *element(It &s, const It &e)
{
        xml_node *ret = new xml_node;
        bool empty_node;

        try {
                match_char(s, e, '<');
                ret->name = name(s, e);

                while (true) {
                        try {
                                space(s, e);

                                ↪ ret->attributes.insert(attribute(s,
                                ↪ e));
                        } catch (const parser_error &) {
                                break;
                        }
                }

                try {
                        auto curr = s;

                        match_char(curr, e, '/');
                        empty_node = true;

                        s = curr;
```

```
                } catch (const parser_error &) {
                        empty_node = false;
                }

                match_char(s, e, '>');

                if (empty_node) {
                        return ret;
                }

                while (true) {
                        maybe_space(s, e);

                        try {
                                auto curr = s;


                                ↪  ret->children.push_back(element(curr,
                                ↪  e));

                                s = curr;
                        } catch (const parser_error &) {
                                break;
                        }
                }

                if (ret->children.empty()) {
                        ret->value = chardata(s, e);
                        rtrim(ret->value);
                }

                match_char(s, e, '<');
                match_char(s, e, '/');
                match_string(s, e, ret->name);
                maybe_space(s, e);
                match_char(s, e, '>');
        } catch (const parser_error &) {
                delete ret;
                throw;
        }

        return ret;
}
```

3.2.11 name

⟨*name*⟩ ::= (⟨*letter*⟩ | _ | :) (⟨*letter*⟩ | ⟨*digit*⟩ | . | : | - | _)*

Il nome di un tag puo' iniziare con una lettera o i due caratteri speciali riportati nella regola. Per i caratteri successivi il nome di un tag puo' iniziare

con una lettera o i due caratteri speciali riportati nella regola. Successivamente sono consentiti sono consentite anche cifre e ulteriori caratteri speciali.

```
template <typename It>
std::string name(It &s, const It &e)
{
        std::string ret;
        char c;

        if (s == e) {
                throw parser_error{"name"};
        }

        c = *s;
        if (!std::isalpha(c) && c != '_' && c != ':') {
                throw parser_error{"name"};
        }

        ret += c;
        s++;

        while (s != e) {
                c = *s;

                if (!std::isalnum(c) && c != '.' && c != ':' && c
                ↪  != '-' && c != '_') {
                        break;
                }

                ret += c;
                s++;
        }

        return ret;
}
```

3.2.12 attribute

⟨*attribute*⟩ ::= ⟨*name*⟩ ⟨*equals*⟩ ⟨*attribute_value*⟩

Un attributo e' costituito da una coppia (nome, valore). Per questo motivo il tipo di ritorno sara' un *std::pair<std::string, std::string>*.

```
template <typename It>
std::pair<std::string, std::string> attribute(It &s, const It &e)
{
        std::pair<std::string, std::string> ret;

        ret.first = name(s, e);
        equals(s, e);
        ret.second = attribute_value(s, e);
```

```
        return ret;
}
```

3.2.13 attribute_value

⟨*attribute_value*⟩ ::= (" (⟨*char*⟩ - ")* ") | (' (⟨*char*⟩ - ')* ')

Il valore di un attributo puo' essere racchiuso tra singoli o doppi apici. Ogni carattere e' consentito (tranne quello terminatore).

```
template <typename It>
std::string attribute_value(It &s, const It &e)
{
        std::string ret;
        char quote_char = '"';
        char c;

        try {
                match_char(s, e, quote_char);
        } catch (const parser_error &) {
                quote_char = '\'';
                match_char(s, e, quote_char);
        }

        while (s != e) {
                c = *s;

                if (c == quote_char) {
                        s++;
                        break;
                }

                ret += c;
                s++;
        }

        return ret;
}
```

3.3 match_char, match_string

Le funzioni *match_char* e *match_string* sono utilizzate per consumare rispettivamente un carattere o una sequenza di caratteri nella stringa d'ingresso.

```
template <typename It>
void match_char(It &s, const It &e, char c)
{
        if (s == e || *s != c) {
                throw parser_error{std::string{c}};
        }
```

```
        s++;
}

template <typename It>
void match_string(It &s, const It &e, const std::string &str)
{
        for (char c : str) {
                if (s == e || *s != c) {
                        throw parser_error{str};
                }

                s++;
        }
}
```

3.4 API

La libreria esporta all'utilizzatore una sola funzione chiamata *parse*. In maniera analoga alle funzioni di parsing, a partire da due iteratori ad inizio e fine stringa costruisce una struttura *xml_ doc* chiamando la regola grammaticale di piu' alto livello, ovvero $\langle document \rangle$.

```
template <typename It>
xml_doc parse(It &s, const It &e)
{
        return parser::document(s, e);
}
```

Parte IV
Utilizzo del parser

La libreria sopra progettata e' stata utilizzata per realizzare un semplice programma. Esso di occupa di leggere in memoria un documento XML da file, e (verificata la sua correttezza) di ristamparlo utilizzando un formato differente. Nel caso il documento non sia corretto, viene stampato a video un messaggio di errore, seguito dalla parte di documento per il quale il parsing e' fallito.

```
int main(int argc, char *argv[])
{
        if (argc != 2) {
                return 1;
        }

        std::ifstream infile(argv[1]);

        if (!infile) {
                std::cerr << "Can't open file " << argv[1] <<
                 ↪  std::endl;
```

```
                return 1;
        }

        infile >> std::noskipws;

        std::istream_iterator<char> file_it(infile), eof;

        std::string buffer(file_it, eof);
        auto start = buffer.begin();
        const auto end = buffer.end();

        xml::xml_doc doc;

        try {
                doc = xml::parse(start, end);
        } catch (const xml::parser_error &ex) {
                std::cerr << ex.what() << std::endl << std::endl;

                std::cerr << "Chars left: " <<
                ↪  std::distance(start, end) << std::endl <<
                ↪  std::endl;
                std::for_each(start, end, [] (char c) {
                        std::cerr << c;
                });
                std::cerr << std::endl;

                return 1;
        }

        std::cout << "XML version: " << doc.version << std::endl
        ↪  << std::endl;
        print_node(std::cout, doc.root);
        std::cout << std::endl;

        return 0;
}
```

La funzione dedicata alla ristampa del documento e' chiamata *print_ node*. Essa richiede in ingresso un oggetto di tipo *std::ostream* sul quale effettuare la stampa, un puntatore ad un nodo e la profondita' del nodo (che di default vale 0), utilizzata per indentare correttamente la struttura a video.

La funzione e' di tipo ricorsivo. Il caso base si verifica quando il nodo in ingresso e' nullo. Negli altri casi la funzione procede alla stampa del nome e degli attributi del nodo. Successivamente richiama se stessa su ognuno dei figli del nodo corrente.

```
void print_node(std::ostream &os, const xml::xml_node *node, int
↪  depth = 0)
{
```

```
        if (node == nullptr)
                return;

        for (int i = 0; i < depth; i++) {
                os << " ";
        }

        os << node->name << " [";

        for (auto &p : node->attributes) {
                os << p.first << "=" << p.second << ", ";
        }

        os << "]";

        if (!node->value.empty()) {
                os << " = " << node->value;
        }

        os << std::endl;

        for (auto &c : node->children) {
                print_node(os, c, depth + 1);
        }
}
```

La complessita' di questo algoritmo e' uguale a quella di una *Depth-first search* su un albero con N nodi, ovvero pari a $\Theta(N)$.

3.5 Esempi di parsing completo

3.5.1 Esempio 1

Input

```
<breakfast_menu>
        <food>
                <name>Belgian Waffles</name>
                <price>$5.95</price>
                <description>
                Two of our famous Belgian Waffles
                </description>
                <calories>650</calories>
        </food>
        <food>
                <name>Strawberry Belgian Waffles</name>
                <price>$7.95</price>
                <description>
                Light Belgian waffles covered with milk
                </description>
```

```
                <calories>900</calories>
        </food>
        <food>
                <name>French Toast</name>
                <price>$4.50</price>
                <description>
                Thick slices made from our homemade bread
                </description>
                <calories>600</calories>
        </food>
        <food>
                <name>Homestyle Breakfast</name>
                <price>$6.95</price>
                <description>
                Two eggs, bacon or sausage, toast
                </description>
                <calories>950</calories>
        </food>
</breakfast_menu>
```

Output

```
XML version: 1.0

breakfast_menu []
 food []
  name [] = Belgian Waffles
  price [] = $5.95
  description [] = Two of our famous Belgian Waffles
  calories [] = 650
 food []
  name [] = Strawberry Belgian Waffles
  price [] = $7.95
  description [] = Light Belgian waffles covered with milk
  calories [] = 900
 food []
  name [] = French Toast
  price [] = $4.50
  description [] = Thick slices made from our homemade bread
  calories [] = 600
 food []
  name [] = Homestyle Breakfast
  price [] = $6.95
  description [] = Two eggs, bacon or sausage, toast
  calories [] = 950
```

3.5.2 Esempio 2

Input

```
<note priority="high">
        <to>Tove</to>
        <from>Jani</from>
        <heading>Reminder</heading>
        <body>Don't forget me this weekend!</body>
</note>
```

Output

```
XML version: 1.0

note [priority=high, ]
 to [] = Tove
 from [] = Jani
 heading [] = Reminder
 body [] = Don't forget me this weekend!
```

3.5.3 Esempio 3

Input

```
<CATALOG>
        <PLANT>
                <COMMON>Bloodroot</COMMON>
                <BOTANICAL>Sanguinaria canadensis</BOTANICAL>
                <ZONE>4</ZONE>
                <LIGHT>Mostly Shady</LIGHT>
                <PRICE>$2.44</PRICE>
                <AVAILABILITY>031599</AVAILABILITY>
        </PLANT>
        <PLANT>
                <COMMON>Columbine</COMMON>
                <BOTANICAL>Aquilegia canadensis</BOTANICAL>
                <ZONE>3</ZONE>
                <LIGHT>Mostly Shady</LIGHT>
                <PRICE>$9.37</PRICE>
                <AVAILABILITY>030699</AVAILABILITY>
        </PLANT>
        <PLANT>
                <COMMON>Marsh Marigold</COMMON>
                <BOTANICAL>Caltha palustris</BOTANICAL>
                <ZONE>4</ZONE>
                <LIGHT>Mostly Sunny</LIGHT>
                <PRICE>$6.81</PRICE>
                <AVAILABILITY>051799</AVAILABILITY>
```

```
        </PLANT>
</CATALOG>
```

Output

```
XML version: 1.0

CATALOG []
 PLANT []
  COMMON [] = Bloodroot
  BOTANICAL [] = Sanguinaria canadensis
  ZONE [] = 4
  LIGHT [] = Mostly Shady
  PRICE [] = $2.44
  AVAILABILITY [] = 031599
 PLANT []
  COMMON [] = Columbine
  BOTANICAL [] = Aquilegia canadensis
  ZONE [] = 3
  LIGHT [] = Mostly Shady
  PRICE [] = $9.37
  AVAILABILITY [] = 030699
 PLANT []
  COMMON [] = Marsh Marigold
  BOTANICAL [] = Caltha palustris
  ZONE [] = 4
  LIGHT [] = Mostly Sunny
  PRICE [] = $6.81
  AVAILABILITY [] = 051799
```

3.6 Esempi di parsing incompleto

3.6.1 Esempio 1

Al nome del tag *note* viene aggiunto il carattere *!*.

Input

```
<!note>
  <to>Tove</to>
  <from>Jani</Ffrom>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Output

```
token not found: name

Chars left: 123
```

```
<!note>
        <to>Tove</to>
        <from>Jani</Ffrom>
        <heading>Reminder</heading>
        <body>Don't forget me this weekend!</body>
</note>
```

3.6.2 Esempio 2

La versione XML indicata e' diversa dalla *1.0*.

Input

```
<?xml version="2.0">
<!note>
        <to>Tove</to>
        <from>Jani</Ffrom>
        <heading>Reminder</heading>
        <body>Don't forget me this weekend!</body>
</note>
```

Output

```
token not found: name

Chars left: 295

?xml version="2.0">
<!note>
        <to>Tove</to>
        <from>Jani</Ffrom>
        <heading>Reminder</heading>
        <body>Don't forget me this weekend!</body>
</note>
```

Si noti come l'errore segnalato sembrerebbe riguardare la regola $\langle name \rangle$, anche se la regola violata e' in realta' $\langle version\_info \rangle$. Questo e' dovuto al fatto che, una volta fallita la regola $\langle version\_info \rangle$, il fallimento si propaga fino alla regola $\langle prologue \rangle$. A questo punto il parser prova ad invocare la regola $\langle element \rangle$, che anch'essa fallisce in quanto un punto esclamativo non e' valido come primo carattere del nome di un elemento.

3.6.3 Esempio 3

Il documento e' terminato prematuramente.

Input

```
<?xml version="1.0" ?>
<note>
```

```
        <to>Tove</to>
```

Output

```
token not found: <

Chars left: 0
```