

A Simple XML Compiler to Check An XML File Against An XML DTD

To finish the implementation of a simple XML compiler, that checks an XML file against an XML DTD, we need to put it all together, what have been implemented while doing the previous tasks.

The front-end of a compiler performs at least lexical analysis, parsing, type-checking and translation into intermediate code for a given source code such that the corresponding target code is produced. The symbol table for handling the corresponding environments implemented by a hash table must be involved in every phase of the compiling mechanisms (even if it has been discussed within the semantic analysis phase). Therefore, we consider the program code for putting it all together in a slightly different order as it is processed by the compiler itself.

Guideline The reader must be aware of the following:

1. The guideline for putting it all together starts with the `Main` class.
2. Then the lexical analyzer is revisited.
3. We study how the symbol table and types are related to the lexer. We study the implementation of symbols stored within a hash table to perform semantic analysis.
4. The symbol table plays a certain role generating intermediate code. Therefore, we continue with the code example related to intermediate code generation.
5. The parser is the “heart“ of every compiling mechanism so that it finally binds all other parts together.
6. The guideline closes with tasks related to the determined sections so that you should be able to combine your program modules, which have been implemented by doing the previous exercise sheets.

Listings The following program listings are incomplete in the sense that they don't give the complete program code that is automatically generated by the *JFlex* tool or the *BYacc/J* tool. The listings only include parts that you must be aware of. This means these lines of

code have a certain functionality or some lines must be added to become your compiler running. Therefore, in the following description the running example defining a front-page of an article is considered again, see also in the former exercises.

Main

All so far implemented classes must be ordered starting in class `Main`. The execution begins in method `main` in class `Main`. Method `main` creates a lexical analyzer and a parser and then it calls method `xmlFile` in the parser. The parser has a parameter that refers to the class of the lexical analyzer.

```
package main;
import java.io.*; import lexer.*; import parser.*;

public class Main {
    public static void main(String[] args) throws IOException {
        Lexer lex = new Lexer();
        Parser parse = new Parser(lex);
        parse.xmlFile();
        System.out.write('\n');
    }
}
```

Lexer

The *JFlex* tool has generated a lexer for us, but we have also to check that some lines of code are included in our *Java* classes.

To check an XML-file against a given XML DTD, the mark-up's can be understood as "reserved words". These reserved words must be handled by the *automatical* generated code for the lexer using *Jflex*. Take care that your lexer includes program code as follows for handling the symbol table using the hash table later on.

```
package lexer;
```

```

import java.io.*; import java.util.*; import symbols.*;
public class Lexer {
    public static int line = 1;
    char peek = ' ';
    Hashable words = new Hastable();
    void void reserve (Word w) { words.put(w.lexeme, w); }
    public Lexer () {
        reserve( new Word("<frontpage>", Tag.LFRONT) );
        reserve( new Word("</frontpage>", Tag.RFRONT) );
        reserve( new Word("<title>", Tag.LTITLE) );
        reserve( new Word("</title>", Tag.RTITLE) );
        reserve( new Word("<name>", Tag.LNAME) );
        reserve( new Word("</name>", Tag.RNAME) );
        reserve( new Word("<address>", Tag.LADDR) );
        reserve( new Word("</address>", Tag.RADDR) );
        reserve( new Word("<content>", Tag.LCONT) );
        reverse( new Word("</content>", Tag.RCONT) );
    }
}

```

Function `readch()` is used to read the next input character into variable `peek`. The name `readch` is reused or overloaded to help recognize composite tokens; e.g. the markup's for a name are split or composed by *first-name* and *last-name*.

```

void readch() throws IOException { peek= (char)System.in.read(); {
boolean readch(char c) throws IOException {
    readch();
    if ( peek != c ) return false;
    peek = ' ';
    return true;
}
}

```

The most important function of the lexer is `scan`. It recognizes the token based on the concept of the deterministic automaton and implemented by the lookup table (see in the lecture notes). This lookup table is the main part of the lexer and is always produced in the same manner so that the *JFlex* tool as done it for us. After checking all possibilities of allowed token, the token must be returned; at the end of the `scan` functionality must be program code, like

```

        Token tok = new Token(peek); peek = ' ';
        return tok;
    }
} // end of lexer

```

Symbol tables and types

Package `symbols` implements symbol tables and types. We need a class that handles our environments. This class is called `Env`. Whereas class `Lexer` maps strings to words, class `Env` maps word token to objects of class `Id`, which is defined in package `inter` for obtaining the intermediate code. The package `inter` will be discussed later on. Thus, `inter` stands for “intermediate code”.

```

package symbols;                // File Env.Java
import java.util.*; import lexer.*; import inter.*;
public class Env Env {
    private Hashable table;
    protected Env prev;
    public Env(Env n) { table = new Hashtable(); prev = n; }
    public void put(Token w, Id i) { table.put(w, i); }
    public Id get(Token w) {
        for ( Env e = this; e != null; e = e.prev ) {
            Id found = (Id)(e.table.get(w));
            if ( found != null ) return found;
        }
        return null;
    }
}

```

We define a class `Type` (see exercise on semantic analysis) to be a subclass of `Word` since basic type names, like `#PCDATA` or `IMAGE` are simply reserved words, to be mapped from the lexemes to appropriate objects by the lexical analyzer. The objects for the basic types are `Type.#PCDATA` and `Type.IMAGE`. All of them have inherited field `tag` set to `Tag.BASIC`, so that the parser treats them all alike.

```

package symbols;

```

```

import lexer.*;
public class Type extends Word {
    public int width = 0;
    public Type(String s; int tag, int w) { super(s, tag); width = w; }
    public static final type Type
    \#PCDATA = new Type( "\#PCDATA", Tag.BASIC, 28),
    IMAGE = new Type ("IMAGE", Tag.BASIC, 35);
}

```

The numbers of token must match with your numbers of token in your specific type definition.

Intermediate code

Examples of classes for handling intermediate code was given in the lecture notes. In any case the class Node has to be extended for representing the markup's and the content in between the markup's.

The markup's and its contents between the parenthesis is implemented by a subclass of Node. The nodes Node correspond to the nodes of the parse tree (or derivation tree).

```

package inter;
import lexer.*;
public class Node {
    int lexline = 0;
    Node() { lexline = Lexer.line; }
    void error(String s) { throw new Error("near line "+lexline+" : "+s); }
    static int labels = 0;
    public int newlabel() { return ++labels; }
    // Left markup stored in Id, see above
    public void emitlabelLeft(int i) { System.out.print((String) Id); }
    // Right markup stored in Id, see above
    public void emitlabelRight(int i) { System.out.print((String) Id); }
    // Content between the markup's
    public void emit(String s) { System.out.println {"\t" + (String) s); }
}

```

The content CONTENT is finally constructed as follows

```
package inter; // File Content.java
import lexer.*; import symbols.*;
public class Content extends Node {
    public Token lMarkup;
    public Token rMarkup;
    public Type type;
    Content(Token ltok, Token rtok, Type p) {
        lMarkup = ltok; rMarkup = rtok; type = p;
    }
}
```

The content must be re-implemented to take that the string in between markup's is valid.

This is only a part of all nodes that must be extended; see previous exercise sheet.

Parser

The parser reads the stream of tokens and builds a parse tree by calling the appropriate constructor functions from the functionality given above. The current symbol table is maintained by the translation scheme implemented by the intermediate code generation phase extending each possible Node.

The package `parser` contains one class `Parser`

```
package parser;
import java.io; import lexer.*; import symbols.*; import inter.*;
public class Parser {
    private Lexer lex; // lexical analysis is bind to the parser
    private Token look // implements the lookahead string
    Env top = null; // current or top symbol table
    int used = 0; // storage used for (composed) markup's
    public Parser(Lexer l) throws IOException { lex = l; move(); }
    void move() throws IOException { look.scan(); }
    void error(String s) { throw new Error("near line"+lex.line+" : "+s); }
```

```

void match(int t) throws IOException {
    if ( look.tag == t ) move ();
    else error("syntax error");
}

```

Parsing begins with a call of the procedure `xmlFile`, which calls `xmlStatement` to parse the input stream and build the parse tree.

```

public void xmlFile() throws IOException { // xmlFile -> xmlStatement
    Stmt s = xmlStatement();
    int begin = s.newlabel(); int after = s.newlabel();
    if ( top == null ) then {
        s.emitlabel(root);
        s.gen("<?xml version='1.0' encoding='UTF-8'?">", after);
        s.emitlabel(after);
    } // create a unit root for every XML document
    else { s.emitlabel(markup); s.gen(markup, after); s.emit(after); }
}

```

The symbol-table that matches with the current parse (sub-)tree is handled by procedure `xmlStatement`.

```

Stmt xmlStatement throws IOException { \ \ xmlStatement -> terms with markups
    match('<'); Env savedEnv = top; top = new Env(top);
    Stmt s = stmts;
    match('</'); top = savedEnv;
    return s;
}

```

The (pure) parsing process is controlled within a `while`-loop, like

```

while ( look.tag == Tag.BASIC ) { // predefined XML markup
    // take care that every markup needs its own definition
    // this is only an example
    Type p = type();
}

```

```
Token tok = look;
match(Tag.markup); match(Tag.inBetween); match(Tag.markup);
Markup markup = new((Word)tok, p, used);
top.put( tok, markup );
used = used + p.width;
}
```

To implement your XML checker

1. Implement *your* Main class.
2. Check your lexer and add the commands for handling the hash table.
3. If it is necessary, reprocess your lexer specification so that the *BYacc/J* tool can be applied as well. Therefore, see the former information about *JFlex*.
4. Extend your symbol table for handling the environments, see also above.
5. Take care that your type checking mechanism is complete. The numbers of token must match with your numbers of token in your specific type definition. The given example cannot be copied! Don't forget to implement the type LINK.
6. Extend the your intermediate code generation, if it is not done before, so that every node is extended regarding the elements of XML DTD. That is the redefinition of all elements so that you can be sure that your XML document is finally valid.
7. Check that your parser is complete (normally this code is generated by *BYacc/J*).
8. Run your compiler and test a valid and an invalid XML document in respect of your XML DTD.