Bachelor Self Study Project:

Similarity in Tree Data Structures

Overview

1 Introduction 1
2 Basic principles for XML Documents 2
3 Algorithm for change detection 3
3.1 X-Diff 3
3.2 X-Tree-Diff 4
3.3 X-Tree-Diff+ 6
4 Step-by-Step explanation of X-Diff 7
4.1 X-Diff 8
5 Comparison 11
6 Conclusion 11
7 Appendix 12
7.1 Source Code File 12
7.2 Literature 12
7.3 Grammar 13

1 Introduction

XML[1] is nowadays the standard format for publishing and transporting documents on the web. The topics of the actual research are to find an effective algorithm to quickly detect changes in XML documents [2]. The algorithm has to be fast, reliable and simple [3].

We are accustomed to have up-to-date information at the right moment, this causes a real-time detection of changes [3]. Computing changes between documents is important because the Internet is crowded with excessive information and we want to detect real-time hackers' defacement attacks [4].

The topic of my work was now to describe the most commonly used techniques in detecting changes, compare these different algorithms and explain it step-by-step.

This paper is structured as follows: first I introduce some basics and principles of XML documents. Then in chapter 3 I describe the different algorithms. Chapter 4 goes down in the functionality of the algorithms. After the description I specify the results of the comparison.

[1] XML is the abbreviation of eXtensible Markup Language and defined from the World Wide Web Consortium [6].

2 Basic principles for XML Documents

In this chapter I introduce the basics, definitions and used notations for the further parts of this paper. To understand the algorithms we need knowledge of the structure of XML-documents and their representation in a tree structure, based on the Document Object Model (DOM)[2].

XML is a hierarchically structured document and can be represented in a tree structure. The data are ordered in logical blocks. The order of the attributes is irrelevant. In such a tree exist different kinds of relations. Nodes can be a child, a parent or a sibling of another node. Every node in a tree is reachable via these relations, based on the root-node [2].

To understand the algorithms we need to discuss three kinds of nodes:

- Element nodes are non-leaf nodes and represent an element of the XML document.
- Text nodes are leaf nodes and show the content of an element or attribute.
- Attribute nodes are leaf nodes and represent an attribute of the XML document. They are not nodes in the tree structure, but attributes of elements nodes [2].

To change a tree structure we need some edit operations. These are for example deleting and inserting nodes. It is possible to represent such an operation as a pair of nodes. Each operation has a cost function, which defines the costs to execute it [1].

For the transformation of one tree into another we need an edit script, which is a sequence of basic edit operations. For two given trees, there are many possible edit scripts. Now it is important to find a minimum-cost edit script or optimal edit script [2].

The tree edit distance between two trees is defined as the minimum cost sequence of node edit operations, that transform one tree into the other [1].

In the Document Object Model, every node has a signature, the first criterion for matching two nodes. The signature is defined as the path from the root to the selected node [2].

A set of node pairs will be called a matching from one tree to another, if the signature is equal. And a minimum-cost matching is then a matching, which generates a minimum-cost edit script for two trees [2].

[2] Document Object Model is a convention for representing and interacting with objects in XML documents, defined by the World Wide Web Consortium [6].

3 Algorithm for change detection

For every algorithm we need to parse through the input files and generate a tree. For easier understanding, I call them T1 (X-Tree from the first document) and T2 (X-Tree from the second document).

3.1 X-Diff

X-Diff is an algorithm for unordered trees that integrates key XML structure characteristics. Given are two different XML documents (D1 and D2). The algorithm takes three steps to generate a minimum-cost edit script [2].

Step 1: Parsing and Hashing

In this step the algorithm parses the two input documents D1 and D2 into DOM-trees[3] (T1 and T2) according to the rules of the Document Object Model. During the parsing process, the algorithm computes a hash value for every node. The X-Hash[4] value of a node represents the entire subtree rooted at this node. Important is, that two identical trees should have the same X-Hash value [2].

Step 2: Matching

Substep 2.1: First the algorithm compares the XHash value of the two roots. If there are identical, D1 and D2 are equivalent. The procedure stops.

Substep 2.2: Now the algorithm filters out next-level subtrees that have equal X-Hash values for reducing matching space. Two subtrees are with extremely high probability identical, if the X-Hash values are equal [2].

Substep 2.3: In this part, the algorithm computes the edit distance for each of the remaining subtree pairs. It starts from leaf node pairs and move upward. For this we need two sets N1 containing all leafs of T1 and N2 with all leafs of T2. The algorithm compares the signature of two corresponding elements of N1 and N2, if they are equal, it writes it into the Distance Table (DT) and computes the edit distance. If the distance is “0”, no editing operation has to be done. A “1” relates to an update operation. After the matching, we update N1 and N2 by replacing the matched node by its parent. This process works as long as both sets (N1 or N2) are not empty.

Substep 2.4: Now the minimum-cost matching will be generated by adding node matchings into it.

Step 3: Generating a minimum-Cost Edit Script

In this step, the algorithm generates based on the minimum-cost matching a minimum-cost edit script. The algorithm writes every matching whose distance is not “0” into the script. It also deletes every node from T1 and inserts every node from T2, which is not in the minimum-cost matching.

---

[3] DOM-Tree is a tree structure defined in the context of the Document Object Model [3].
[4] X-Hash is a hash function used in the paper [2]

3.2 X-Tree-Diff

X-Tree-Diff is also a change detection algorithm for tree-structured data, using the structure of X-trees[5]. It allows exact matching at early stage to reduce wrong matchings [3].

A node of an X-tree needs several fields. Showed in figure 1.

Label	Type	Value	Index	nMD	tMD	nPtr	Op

figure 1: the fields used for the X-Tree-Diff algorithm

First there are *Label*, *Type* and *Value*, which represent the elements of the XML document. Then follows a field *Index*. It distinguishes sibling nodes with the same label. ILabel is a concatenation of the *Label*- and *Index*-field. At the end, there are four fields to detect changes in the XML document. The first one called *nMD* stores the hash value of the content of a node and the second one called *tMD* the structure and data of the subtree rooted at this node. These two variables are implemented using DOMHASH[6]. The field *nPtr* stores a pointer and the field *Op* an operation. In this model exist five edit operations, called *INS* (insert a node), *DEL* (delete a node), *UPD* (update a node), *MOV* (move a node) and *NOP* (a dummy operation) [3].

Step 0: Build up X-Trees

The first step is to convert the XML document into an X-Tree and generate two hash-tables containing tuples of *tMD* values and pointers of an X-tree node with *tMD* as the key. In the *O_Htable* are all entries from T1 saved, that *tMD* is non-unique while *N_Htable* stores all entries from T2, where *tMD* is unique During this process, all fields are initialized or computed [3].

Step 1: Match identical subtrees with 1-to-1 correspondence

Now, the algorithm finds pairs of identical subtrees and matches them using the operation *NOP* (set the *Op* field to *NOP*). For this the algorithm traverses T1 in breadth-first order and takes all nodes, which satisfy the condition, that there is no equal entry in the *O_Htable* and an equal entry in the *N_Htable*, compared by the *tMD* value. If there exists such a node, it will be a matching including its subtree. For finishing the matching, we set the *Op* field to *NOP* and add it to a List, called *M_List*. We do not need to visit the subtree and we can go directly to the next node. Nodes with equal *tMD* values are ignored in this step. They will be matched later.

Step 2: Propagate matchings upward

Then, X-Tree-Diff propagates the matchings found in Step 1 upward to the roots [3]. For each matching stored in the M_List, we need to decide whether the parent of one part of a matching pair can be matched with the parent of the other part based on their labels [4].

---

[5] X-Tree is a labeled ordered tree, designed to reflect the hierarchical structure of XML documents [3].

[6] DOMHASH is a clear and unambiguous definition of digest (hash) values of the XML objects regardless of the surface string variation of XML [5].

Step 3: Match remaining nodes downwards

Now the algorithm traverses T1 in depth-first order and tries to find matched nodes which have unmatched children. If the unmatched children have the same *tMD* value, the subtree rooted on this node will be matched. If they have the same *ILabel* and the same *Value*, they will be matched with the operation *NOP*. Otherwise, if they have only the same *ILabel*, they will be matched with the update operation [3].

Step 4: Determine nodes for addition and deletion

All nodes that can be matched have already been matched in step 1-3. Therefore unmatched nodes in T1 have to be deleted and all unmatched nodes in T2 have to be marked to insert. In this step, the algorithm traverses the two trees and sets the operation field to *DEL* respectively to *INS* [3].

3.3 X-Tree-Diff+

This algorithm is based on X-Tree-Diff. It increases the matching ratio and produces better edit scripts. X-Tree Diff+ uses the eight fields presented in chapter 3.2 and introduces a new one name *iMD* (see figure 2). This field represents an ID attribute value for a node and allows the uniquely identification. Additionally a new copy operation is also defined [4].

Label	Type	Value	Index	nMD	tMD	nPtr	Op	iMD

figure 2: the fields used for the X-Tree-Diff+ algorithm

Step 0: Build up X-Trees

The preprocessing step works equal to the X-Tree Diff algorithm. Additionally to *O_Htable* and *N_Htable* the algorithm computes a new hash-table called *N_IDHtable* for nodes with ID attributes in the T2. The entries consist of *iMD* (as a key) of a node and a pointer to the node [4].

Step 1: Match identical subtrees with 1-to-1 correspondence and match nodes with ID attributes

First we match identical subtrees with 1-to-1 correspondence analogous to Step 1 in the X-Tree-Diff algorithm. After this previous sub-step, we try to match nodes with a same *iMD* values. For this, we traverse T1 and look for every unmatched node with ID attribute if there exists an entry in the *N_IDHtable* with the same *iMD* value. Then we match them with setting the *Op* field to *NOP* [4].

Step 2 and 3: Propagate matching upward and matching remaining nodes downwards

These steps work exactly the same as in the X-Tree Diff-Algorithm [4].

Step 4: Tune existing matches

Now the algorithm analyzes the quality of matches and tunes some ineffective matches by looking for an alternative match for a node. It does it by computing a ratio called consistency of matching, defined as the number of positive children's matches divided by the total number of children matched. If this ratio is lower than a certain value, X-Tree Diff+ looks for alternative matches [4].

Step 5: Match remaining identical subtrees with move and copy operations

In this step we try to match identical subtrees, which have not been matched in step 1, with move and copy operations. For this procedure we need two additional hash-tables for all the unmatched nodes, called *S_Htable* for T1 and *T_Htable* for T2. The entries of these tables are tuples of a certain *tMD* value t and a list of nodes with the same *tMD* value t.

The algorithm now will look for each entry in *T_Htable* if there is an equal *tMD* value t in *S_Htable*. If this search is successful, we will match pairs of nodes with the same position in the lists of nodes [4].

4 Step-by-Step explanation of X-Diff

The input files are D1 (test.c) and D2 (test1.c). For every algorithm we need to parse through the input files and generate a tree with a given grammar (see the appendix). The result is displayed in figure (1 and 2). For easier understanding, I call them T1 (X-Tree from D1) and T2 (X-Tree from D2). Every node has a number that has no relevancy for the algorithm. It is only for a better understanding of my explanation. The yellow boxes in figure 4 are the nodes, which came new or are updated.

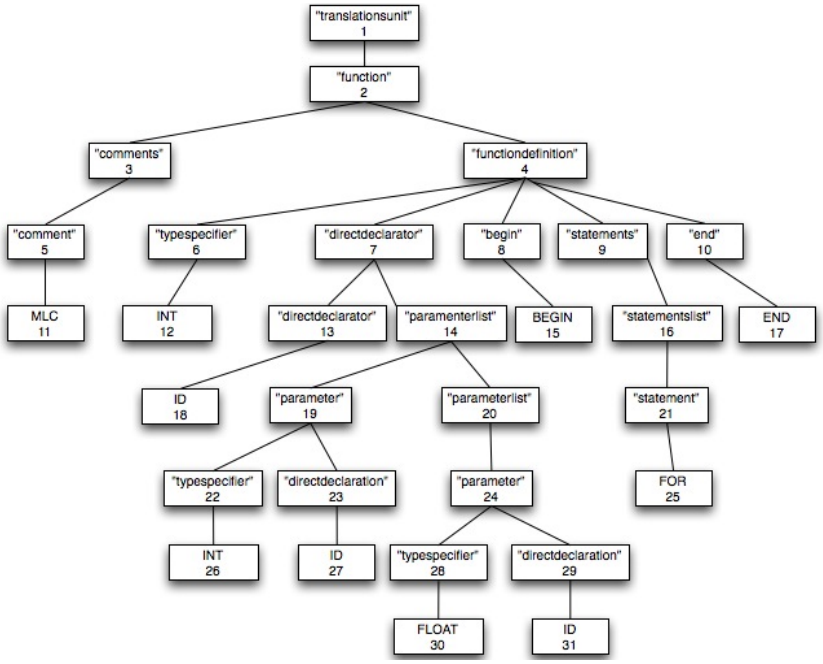

figure 3: X-Tree of test.c

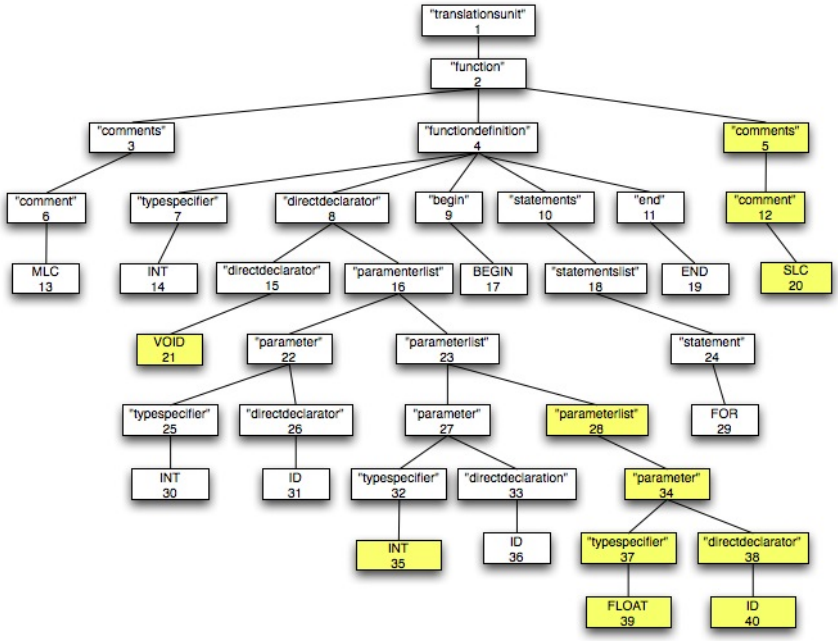

figure 4: X-Tree of test1.c

4.1 X-Diff

Step 1: Parsing and Hashing

In this step the algorithm parses the two input documents D1 and D2 into trees (T1 and T2). During the parsing process, the algorithm computes a hash value for every node. The X-Hash value of a node represents the entire subtree rooted at this node. Important is, that two identical trees should have the same X-Hash value [2]. For the step-by-step explanation I look only on a small subtree showed in figure 5 called ST1 and ST2. The hash values in this example are symbolic and not based on a hash function.

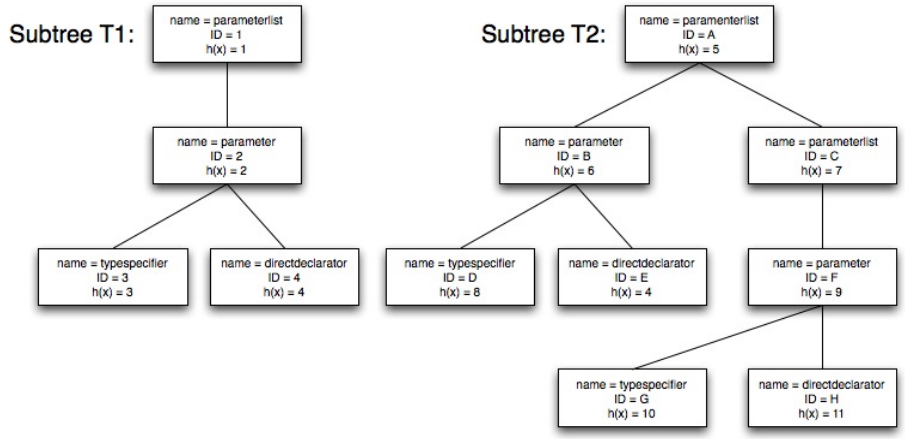

figure 5: subtrees of T1 and T2

List of the signature of all nodes:

ID	Signature
1	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ element node
2	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ parameter/element node
3	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ parameter/typespecifier/ element node
4	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ parameter/directdeclarator/ element node
A	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ element node
B	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ parameter/element node
C	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ parameterlist/element node
D	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ parameter/typespecifier/ element node
E	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ parameter/directdeclarator/ element node

F	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ parameterlist/parameter/element node
G	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ parameterlist/parameter/typespecifier/element node
H	Translationunit/function/functiondef/directdeclarator/parameterlist/parameterlist/ parameterlist/parameter/directdeclarator/element node

Step 2: Matching

Substep 2.1: First the algorithm compares the XHash value of the two roots.

h(root ST1) = 1

h(root ST2) = 5

They are not equal and the algorithm keeps moving on substep 2.2.

Substep 2.2: Now the algorithm filters out next-level subtrees that have equal X-Hash values for reducing matching space. We have to compare in ST1 the node with the ID 2 and in ST2 the nodes with ID B and C.

h(ID = 2) = 2

h(ID = B) = 6

h(ID = C) = 7

These values are all different; therefore we can not eliminate a subtree.

Substep 2.3: In this part, the algorithm computes the edit distance for each of the remaining subtree pairs. It starts from leaf node pairs and move upward. For this we need two sets N1 containing all leafs of ST1 and N2 with all leafs of ST2.

N1 = {3, 4} N2 = {D, E, G, H} DT = { }

The algorithm compares the signature of two corresponding elements of N1 and N2, if they are equal, it writes it into the Distance Table (DT) and computes the edit distance. If the distance is "0", no editing operation has to be done. A "1" relates to an update operation. After the matching, we update N1 and N2 by replacing the matched node by its parent.

First match is node 3 from ST1 with node D from ST2. The sets look as follows:

N1 = {2, 4} N2 = {B, E, G, H} DT = {(3,D) = 0}

The second match relates the nodes 4 and E:

N1 = {2} N2 = {B, G, H} DT = {(3,D) = 0; (4,E) = 0}

Than we can match the nodes 2 and B:

N1 = {1} N2 = {A, G, H} DT = {(3,D) = 0; (4,E) = 0; (2,B)=0}

Than we can match the root nodes 1 and A:

N1 = { } N2 = {G, H} DT = {(3,D) = 0; (4,E) = 0; (2,B)=0; (1,A) = 0}

Now there are no other matches possible because the set N1 is empty.

Substep 2.4: Now the minimum-cost matching (Mmin(ST1,ST2)) will be generated. First I have to insert the roots of the two subtrees.

Mmin (ST1, ST2) = {(1,A)}

Than for every non-leaf mapping in Mmin, I have to retrieve matchings between their child nodes that are stored in DT and add them to Mmin.

Mmin (ST1, ST2) = {(1,A) = 0, (3,D) = 0; (4,E) = 0; (2,B) = 0}

Step 3: Generating a minimum-Cost Edit Script

In this step, the algorithm generates based on the minimum-cost matching a minimum-cost edit script called E. The algorithm writes every matching whose distance is not "0" into the script. In this example E is still empty, because every matching from Mmin equals "0".

E = { }

The next step is to delete nodes from ST1, which could not be matched in step 2. In the example there are no nodes of this type. E is still empty.

E = { }

After the deleting we have to insert all nodes from ST2 that are not contained in the minimum-cost matching. The insert-function has two variables insert(X, Y). This means, that the node X has to be insert as a leaf child of node Y.

E = {insert(C,A), insert (F,C), insert(G,F), insert(H,F)}

5 Comparison

In this chapter I want to give an overview about the differences between the different algorithms.

	X-Diff [2]	*X-Tree Diff [3]*	*X-Tree Diff+ [4]*
Tree structure	Unordered	X-Tree (labeled ordered tree)	X-Tree (labeled ordered tree)
Procedure of algorithm	Computes edit distance and generate minimum-cost edit script	Works with the special data structure (X-Tree) and its fields.	Works with the special data structure (X-Tree) and its fields.
Hash function	XHash	Works with tMD and pointer of the node with tMD as the key	Works with tMD and pointer of the node with tMD as the key
Additional storage	One Distance table	Two hash-tables	Five hash-tables
Reliability	Near-optimal results	Less reliable than X-Tree Diff+	Much higher matching ratio than the other two algorithms.

6 Conclusion

In the introduction I have wrote, that the research is looking for an algorithm that is fast, reliable and simple. I think every of the three presented algorithms has its advantages and disadvantages. X-Diff is maybe simple and reliable, but not fast enough for real-time change detection. X-Tree-Diff runs in linear time, but the quality of the edit scripts is not as good as it should be (depends on the utilization). X-Tree-Diff+ is fast and reliable (has a much higher matching ratio than the other two algorithms [4]) but not simple to understand and use.

The requirements on real-time data will increase in the future also with a rising amount of data. With these three algorithms and two different approaches on the problem (X-Tree-Diff and X-Tree Diff+ are similar) we have solutions for different applications. It is not possible to say, one algorithm is better than the other, it depends on the context of use.

7 Appendix

7.1 Source Code File

Test.c

```
/* Hallo Welt */

int z(int i, float k){
    for
}
```

Test1.c

```
/* Hallo Welt */

void z(int i, int j, float k){
    for
}
//Hallo Welt 1
```

7.2 Literature

[1] Augsten, N.: **lecture slides.** http://www.inf.unibz.it/dis/teaching/ATA/. (retrieved on 14.01.2010)

[2] Cai, J., DeWitt, D.J. and Wang, Y.: **X-Diff: An Effective Change Detection Algorithm for XML Documents**. http://www.cs.wisc. edu/~yuanwang/xdiff.html. (retrieved on 14.01.2010)

[3] Kim, D.A., Lee, S.: **Efficient Change Detection in Tree-Structured Data.** http://www.springerlink.com/content/dk3b8m4r2phck282/. (retrieved on 14.01.2010)

[4] Kim, D.A., Lee, S.: **X-Tree Diff+: Efficient Change Detection Algorithm in XML Documents.** http://www.springerlink.com/content/r1t6h8631868k615/. (retrieved on 14.01.2010)

[5] Maruyama, H., Tamura, K., Uramoto, N.: **Digest Values for DOM.** https://www.research.ibm.com/trl/projects/xml/xss4j/docs/rfc2803.html (retrieved on 03.02.2010)

[6] W3C: **Homepage of the World Wide Web Consortium**. http://www.w3.org/DOM/ and http://www.w3.org/XML (retrieved on 03.02.2010)

7.3 Grammar

```
/*definitions*/
%token <Object> VOID
%token <Object> FLOAT
%token <Object> INT
%token <Object> FOR
%token <Object> WHILE
%token <Object> ID
%token <Object> DECINTLIT
%token <Object> EQ
%token <Object> EQEQ
%token <Object> PLUS
%token <Object> PLUSPLUS
%token <Object> MINUS
%token <Object> MINUSMINUS
%token <Object> LT
%token <Object> GT
%token <Object> LEQ
%token <Object> GEQ
%token <Object> MLC
%token <Object> SLC
%token <Object> STRING
%token <Object> BEGIN
%token <Object> END
%type <Object> translationunit
%type <Object> functions
%type <Object> functiondefinition
//%type <Object> functiondefinitionwcomment
%type <Object> comments
%type <Object> comment
%type <Object> typespecifier
%type <Object> parameter
%type <Object> parameterlist
%type <Object> statementlistwcomment
%type <Object> statements
%type <Object> statement
%type <Object> statementlist
%type <Object> statementwoptcomment
%type <Object> varDecl
%type <Object> directdeclarator
%type <Object> begin
%type <Object> end
%start translationunit
%%

/*rules*/
/*1*/
translationunit:
                                                    functions {
                                                    $$ = new Node("translationunit");
                                                    ((Node) $1.obj).setParent(((Node) $$.obj).getPos());
                                                    }
                                                    ;


/*2*/
functions:
                                                    functiondefinition {
                                                    $$.obj = new Node("functions");

                                                    ((Node) $1.obj).setParent(((Node) $$.obj));
                                                    }
                                                    | functiondefinition comments {
                                                    $$.obj = new Node("functions");

                                                    ((Node) $1.obj).setParent(((Node) $$.obj);
                                                    ((Node) $1.obj).setNextSibling(((Node) $2.obj);

                                                    ((Node) $2.obj).setParent(((Node) $$.obj));
                                                    }
                                                    | comments functiondefinition {
                                                    $$.obj = new Node("functions");

                                                    ((Node) $1.obj).setParent(((Node) $$.obj));
                                                    ((Node) $1.obj).setNextSibling(((Node) $2.obj));

                                                    ((Node) $2.obj).setParent(((Node) $$.obj));
                                                    }
                                                    | comments functiondefinition comments {
                                                    $$.obj = new Node("functions");

                                                    ((Node) $1.obj).setParent(((Node) $$.obj));
                                                    ((Node) $1.obj).setNextSibling(((Node) $2.obj));

                                                    ((Node) $2.obj).setParent(((Node) $$.obj));
                                                    ((Node) $2.obj).setNextSibling(((Node) $3.obj));

                                                    ((Node) $3.obj).setParent(((Node) $$.obj));
                                                    }
                                                    | functiondefinition functions {
                                                    $$.obj = new Node("functions");

                                                    ((Node) $1.obj).setParent(((Node) $$.obj));
                                                    ((Node) $1.obj).setNextSibling(((Node) $2.obj));

                                                    ((Node) $2.obj).setParent(((Node) $$.obj));
                                                    }
                                                    | comments functiondefinition functions {
                                                    $$.obj = new Node("functions");

                                                    ((Node) $1.obj).setParent(((Node) $$.obj));
                                                    ((Node) $1.obj).setNextSibling(((Node) $2.obj));
```

```
((Node) $2.obj).setParent(((Node) $$.obj));
((Node) $2.obj).setNextSibling(((Node) $3.obj));

((Node) $3.obj).setParent(((Node) $$.obj));
}
;

/*3*/
comments:
                                comment comments {
                                $$.obj = new Node("comments");

                                ((Node) $1.obj).setParent(((Node) $$.obj));
                                ((Node) $1.obj).setNextSibling(((Node) $2.obj));

                                ((Node) $2.obj).setParent(((Node) $$.obj));
                                }
                                | comment {
                                $$.obj = new Node("comments");

                                ((Node) $1.obj).setParent(((Node) $$.obj));
                                }
                                ;

/*4*/
comment:
                                MLC {
                                $$.obj = new Node("comment", MLC);
                                }
                                | SLC {
                                $$.obj = new Node("comment", SLC);
                                }
                                ;

/*5*/
functiondefinition:
                                typespecifier directdeclarator begin statements end {
                                $$.obj = new Node("functiondefinition");

                                ((Node) $1.obj).setParent(((Node) $$.obj));
                                ((Node) $1.obj).setNextSibling(((Node) $2.obj));

                                ((Node) $2.obj).setParent(((Node) $$.obj));
                                ((Node) $2.obj).setNextSibling(((Node) $3.obj));

                                ((Node) $3.obj).setParent(((Node) $$.obj));
                                ((Node) $3.obj).setNextSibling(((Node) $4.obj));


                                ((Node) $4.obj).setParent(((Node) $$.obj));
                                ((Node) $4.obj).setNextSibling(((Node) $5.obj));

                                ((Node) $5.obj).setParent(((Node) $$.obj));
                                }
                                ;

/*6*/
typespecifier:
                                VOID {
                                $$.obj = new Node("typespecifier", VOID);
                                }
                                | INT {
                                $$.obj = new Node("typespecifier", INT);
                                }
                                | FLOAT {
                                $$.obj = new Node("typespecifier", FLOAT);
                                }
                                ;

/*7*/
parameterlist:
                                parameter {
                                $$.obj = new Node("parameterlist");

                                ((Node) $1.obj).setParent(((Node) $$.obj));
                                }
                                | parameter ',' parameterlist {
                                $$.obj = new Node("parameterlist");

                                ((Node) $1.obj).setParent(((Node) $$.obj));
                                ((Node) $1.obj).setNextSibling(((Node) $3.obj));

                                ((Node) $3.obj).setParent(((Node) $$.obj));
                                }
                                ;

/*8*/
parameter:
                                typespecifier directdeclarator {
                                $$.obj = new Node("parameter");

                                ((Node) $1.obj).setParent(((Node) $$.obj));
                                ((Node) $2.obj).setNextSibling(((Node) $2.obj));

                                ((Node) $2.obj).setParent(((Node) $$.obj));
                                }
                                | typespecifier {
                                $$.obj = new Node("parameter");
```

```
                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        }
                                        ;

/*9*/
statements:
                                        statementlistwcomment {
                                        $$.obj = new Node("statements");

                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        }
                                        | statementlist {
                                        $$.obj = new Node("statements");

                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        }
                                        ;

/*10*/
statementlist:
                                        statement {
                                        $$.obj = new Node("statementlist");

                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        }
                                        | statement statementlist {
                                        $$.obj = new Node("statementlist");

                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        ((Node) $1.obj).setNextSibling(((Node) $2.obj));

                                        ((Node) $2.obj).setParent(((Node) $$.obj));
                                        }
                                        ;

/*11*/
statementlistwcomment:
                                        comment statementwoptcomment {
                                        $$.obj = new Node("statementlistwcomment");

                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        ((Node) $1.obj).setNextSibling(((Node) $2.obj));

                                        ((Node) $2.obj).setParent(((Node) $$.obj));
                                        }
                                        | statementwoptcomment comment {
                                        $$.obj = new Node("statementlistwcomment");

                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        ((Node) $1.obj).setNextSibling(((Node) $2.obj));

                                        ((Node) $2.obj).setParent(((Node) $$.obj));
                                        }
                                        ;

/*12*/
statement:
                                        FOR {
                                        $$.obj = new Node("statement", FOR);
                                        }
                                        | WHILE {
                                        $$.obj = new Node("statement", WHILE);
                                        }
                                        | varDecl {
                                        $$.obj = new Node("statement");

                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        }
                                        ;

/*13*/
statementwoptcomment:
                                        comment statementwoptcomment {
                                        $$.obj = new Node("statementlistwoptcomment");

                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        ((Node) $1.obj).setNextSibling(((Node) $2.obj));

                                        ((Node) $2.obj).setParent(((Node) $$.obj));
                                        }
                                        | statement {
                                        $$.obj = new Node("statementwoptcomment");

                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        }
                                        ;

/*14*/
varDecl:
                                        typespecifier directdeclarator ';' {
                                        $$.obj = new Node("varDecl");

                                        ((Node) $1.obj).setParent(((Node) $$.obj));
                                        ((Node) $1.obj).setNextSibling(((Node) $2.obj));

                                        ((Node) $2.obj).setParent(((Node) $$.obj));
                                        }
                                        ;

/*15*/
```

```
directdeclarator:
                                                    ID {
                                                    $$.obj = new Node("directdeclarator", -1, -1, 0, ((Tuple) $1.obj), ((Tuple) $1.obj));
                                                    }
                                                    | directdeclarator '(' parameterlist ')' {
                                                    $$.obj = new Node("directdeclarator");

                                                    ((Node) $1.obj).setParent(((Node) $$.obj));
                                                    ((Node) $1.obj).setNextSibling(((Node) $3.obj));

                                                    ((Node) $3.obj).setParent(((Node) $$.obj));
                                                    }
                                                    ;

/*16*/
begin:
                                                    BEGIN {
                                                    $$.obj = new Node("begin", BEGIN);
                                                    }
                                                    ;

/*17*/
end:
                                                    END {
                                                    $$.obj = new Node("end", END);
                                                    }
                                                    ;


%%
```