```
////////////////////////////////////////////////////////////////////////////
//Class:        CS 445
//Semester:     Fall 2011
//Assignment:   Homework 3
//Author:       Colby Blair (modifier, see below)
//File name:    AS3.y
////////////////////////////////////////////////////////////////////////////

/*
 * AS3.y, from AS3.g3
 *
 * Copyright (c) 2005 Martin Schnabel
 * Copyright (c) 2006-2008 David Holroyd
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.

// Originally derived from the ANTLRv2 ActionScript grammar by
// Martin Schnabel, included in the ASDT project,
//   http://www.asdt.org
//   http://sourceforge.net/projects/aseclipseplugin/
 */

%{
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>

#include "tree.h"
#include "main.h"

extern int yylex();
extern int lineno;
extern int colno;
extern char yytext[];
extern struct tree *YY_TREE;
char *YY_PRODRULE;

//debugging
#ifdef YYDEBUG
int yydebug = 1;
#endif

// better error reporting
#define YYERROR_VERBOSE

//tree_create_node macro
#define _TCN tree_create_node

// bison requires that you supply this function
int yyerror(const char *msg)
{
        printf("ERROR(PARSER): %s at line %d:%d. token: %s\n",
                                                        msg, lineno,
                                                        colno, yytext);

        //return(1);
        exit(ERROR_SYNTAX);
}


%}

%expect 28
```

```
%union {
        struct tree *t;
}

%token <t> PACKAGE /*           :       'package';*/
%token <t> PUBLIC /*            :       'public';*/
%token <t> PRIVATE /*           :       'private';*/
%token <t> PROTECTED /* :       'protected';*/
%token <t> INTERNAL /*  :       'internal';*/
%token <t> OVERRIDE /*  :       'override';*/
%token <t> FUNCTION /*  :       'function';*/
%token <t> EXTENDS /*           :       'extends';*/
%token <t> IMPLEMENTS /*        :       'implements';*/
%token <t> VAR /*               :       'var';*/
%token <t> STATIC /*            :       'static';*/
%token <t> IF /*                :       'if';*/
%token <t> IMPORT /*            :       'import';*/
%token <t> FOR /*               :       'for';*/
%token <t> EACH /*              :       'each';*/
%token <t> IN /*                :       'in';*/
%token <t> WHILE /*             :       'while';*/
%token <t> DO /*                :       'do';*/
%token <t> SWITCH /*            :       'switch';*/
%token <t> CASE /*              :       'case';*/
%token <t> DEFAULT /*           :       'default';*/
%token <t> ELSE /*              :       'else';*/
%token <t> CONST /*             :       'const';*/
%token <t> CLASS /*             :       'class';*/
%token <t> INTERFACE /* :       'interface';*/
%token <t> TRUE /*              :       'true';*/
%token <t> FALSE /*             :       'false';*/
%token <t> DYNAMIC /*           :       'dynamic';*/
%token <t> USE /*               :       'use';*/
%token <t> XML /*               :       'xml';*/
%token <t> NAMESPACE /* :       'namespace';*/
%token <t> IS /*                :       'is';*/
%token <t> AS /*                :       'as';*/
%token <t> GET /*               :       'get';*/
%token <t> SET /*               :       'set';*/
%token <t> WITH /*              :       'with';*/
%token <t> RETURN /*            :       'return';*/
%token <t> CONTINUE /*  :       'continue';*/
%token <t> BREAK /*             :       'break';*/
%token <t> NULL_VAL /*          :       'null';*/
%token <t> NEW /*               :       'new';*/
%token <t> SUPER /*             :       'super';*/
%token <t> INSTANCEOF /*        :       'instanceof';*/
%token <t> DELETE /*            :       'delete';*/
%token <t> VOID /*              :       'void';*/
%token <t> TYPEOF /*            :       'typeof';*/
%token <t> TRY /*               :       'try';*/
%token <t> CATCH /*             :       'catch';*/
%token <t> FINALLY /*           :       'finally';*/
%token <t> UNDEFINED /* :       'undefined';*/
%token <t> THROW /*             :       'throw';*/
%token <t> FINAL /*             :       'final';*/

%token <t> QUESTION /*          :       '?'     ;*/
%token <t> LPAREN /*                    :       '('     ;*/
%token <t> RPAREN /*                    :       ')'     ;*/
%token <t> LBRACK /*                    :       '['     ;*/
%token <t> RBRACK /*                    :       ']'     ;*/
%token <t> LCURLY /*                    :       '{'     ;*/
%token <t> RCURLY /*                    :       '}'     ;*/
%token <t> COLON /*                     :       ':'     ;*/
%token <t> DBL_COLON /*         :       '::'    ;*/
%token <t> COMMA /*                     :       ','     ;*/
%token <t> ASSIGN /*                    :       '='     ;*/
%token <t> EQUAL /*                     :       '=='    ;*/
%token <t> STRICT_EQUAL /*              :       '==='   ;*/
%token <t> LNOT /*                      :       '!'     ;*/
%token <t> BNOT /*                      :       '~'     ;*/
%token <t> NOT_EQUAL /*         :       '!='    ;*/
```

```
%token <t> STRICT_NOT_EQUAL /*  :       '!=='   ;*/
%token <t> DIV /*                       :       '/'     ;*/
%token <t> DIV_ASSIGN /*                :       '/='    ;*/
%token <t> PLUS /*                      :       '+'     ;*/
%token <t> PLUS_ASSIGN /*               :       '+='    ;*/
%token <t> INC /*                       :       '++'    ;*/
%token <t> MINUS /*                     :       '-'     ;*/
%token <t> MINUS_ASSIGN /*              :       '-='    ;*/
%token <t> DEC /*                       :       '--'    ;*/
%token <t> STAR /*                      :       '*'     ;*/
%token <t> STAR_ASSIGN /*               :       '*='    ;*/
%token <t> MOD /*                       :       '%'     ;*/
%token <t> MOD_ASSIGN /*                :       '%='    ;*/
%token <t> SR /*                        :       '>>'    ;*/
%token <t> SR_ASSIGN /*         :       '>>='   ;*/
%token <t> BSR /*                       :       '>>>'   ;*/
%token <t> BSR_ASSIGN /*                :       '>>>='  ;*/
%token <t> GE /*                        :       '>='    ;*/
%token <t> GT /*                        :       '>'     ;*/
%token <t> SL /*                        :       '<<'    ;*/
%token <t> SL_ASSIGN /*         :       '<<='   ;*/
%token <t> LE /*                        :       '<='    ;*/
%token <t> LT /*                        :       '<'     ;*/
%token <t> BXOR /*                      :       '^'     ;*/
%token <t> BXOR_ASSIGN /*               :       '^='    ;*/
%token <t> BOR /*                       :       '|'     ;*/
%token <t> BOR_ASSIGN /*                :       '|='    ;*/
%token <t> LOR /*                       :       '||'    ;*/
%token <t> BAND /*                      :       '&'     ;*/
%token <t> BAND_ASSIGN /*               :       '&='    ;*/
%token <t> LAND /*                      :       '&&'    ;*/
%token <t> LAND_ASSIGN /*               :       '&&='   ;*/
%token <t> LOR_ASSIGN /*                :       '||='   ;*/
%token <t> E4X_ATTRI /*         :       '@'     ; */
%token <t> SEMI /*                      :       ';'     ;*/
%token <t> DOT /*                       :       '.'     ;*/
%token <t> E4X_DESC /*  :       '..'    ;*/
%token <t> REST /*                      :       '...'   ;*/
%token <t> AND INTRINSIC OR IDENT EOFX ENUMERABLE EXPLICIT FLOAT_LITERAL
%token <t> DECIMAL_LITERAL OCTAL_LITERAL STRING_LITERAL HEX_LITERAL
%token <t> INCLUDE INCLUDE_DIRECTIVE

%type <t> compilationUnit
%type <t> as2CompilationUnit
%type <t> as3CompilationUnit
%type <t> importDefinitions
%type <t> as2Type
%type <t> packageDecl
%type <t> packageBlock
%type <t> packageBlockEntry
%type <t> packageBlockEntries
%type <t> importDefinition
%type <t> semi
%type <t> classDefinition
%type <t> as2ClassDefinition
%type <t> interfaceDefinition
%type <t> as2InterfaceDefinition
%type <t> classExtendsClause
%type <t> interfaceExtendsClause
%type <t> commaIdentifiers
%type <t> implementsClause
%type <t> typeBlock
%type <t> typeBlockEntry
%type <t> typeBlockEntries
%type <t> as2IncludeDirective
%type <t> includeDirective
%type <t> blockOrSemi
%type <t> optionalTypeExpression
%type <t> methodDefinition
%type <t> optionalAccessorRole
%type <t> accessorRole
%type <t> namespaceDefinition
%type <t> useNamespaceDirective
```

```
%type <t> commaVariableDeclarators
%type <t> variableDefinition
%type <t> varOrConst
%type <t> optionalVariableInitializer
%type <t> variableDeclarator
%type <t> declaration
%type <t> declarationTail
%type <t> variableInitializer
%type <t> commaParameterDeclaration
%type <t> parameterDeclarationList
%type <t> parameterDeclaration
%type <t> basicParameterDeclaration
%type <t> parameterDefault
%type <t> parameterRestDeclaration
%type <t> blockEntries
%type <t> block
%type <t> blockEntry
%type <t> condition
%type <t> statement
%type <t> statements
%type <t> superStatement
%type <t> declarationStatement
%type <t> expressionStatement
%type <t> ifStatement
%type <t> elseClause
%type <t> throwStatement
%type <t> defaultStatement
%type <t> tryStatement
%type <t> catchBlocks
%type <t> catchBlock
%type <t> finallyBlock
%type <t> returnStatement
%type <t> continueStatement
%type <t> breakStatement
%type <t> switchStatement
%type <t> switchBlock
%type <t> caseStatements
%type <t> caseStatement
%type <t> optionalDefaultStatement
%type <t> switchStatementList
%type <t> forEachStatement
%type <t> forStatement
%type <t> traditionalForClause
%type <t> forInClause
%type <t> forInClauseDecl
%type <t> forInClauseTail
%type <t> forInit
%type <t> forCond
%type <t> forIter
%type <t> whileStatement
%type <t> doWhileStatement
%type <t> withStatement
%type <t> defaultXMLNamespaceStatement
%type <t> typeExpression
%type <t> identifier
%type <t> propertyIdentifier
%type <t> qualifier
%type <t> simpleQualifiedIdentifier
%type <t> expressionQualifiedIdentifier
%type <t> nonAttributeQualifiedIdentifier
%type <t> qualifiedIdentifier
%type <t> qualifiedIdent
%type <t> namespaceName
%type <t> reservedNamespace
%type <t> dotIdents
%type <t> identifierStar
%type <t> annotations
%type <t> annotation
%type <t> moreAnnotationParams
%type <t> annotationParamList
%type <t> annotationParam
%type <t> modifiers
%type <t> modifier
```

```
%type <t> arguments
%type <t> arrayLiteral
%type <t> elementList
%type <t> moreAssignmentExpressions
%type <t> nonemptyElementList
%type <t> element
%type <t> objectLiteral
%type <t> moreLiteralFields
%type <t> fieldList
%type <t> literalField
%type <t> fieldName
%type <t> expression
%type <t> expressionList
%type <t> assignmentExpression
%type <t> assignmentOperator
%type <t> conditionalExpression
%type <t> conditionalSubExpression
%type <t> logicalOrExpression
%type <t> logicalOrOperator
%type <t> logicalAndExpression
%type <t> logicalAndOperator
%type <t> bitwiseOrExpression
%type <t> bitwiseXorExpression
%type <t> bitwiseAndExpression
%type <t> equalityExpression
%type <t> equalityOperator
%type <t> relationalExpression
%type <t> relationalOperator
%type <t> shiftExpression
%type <t> shiftOperator
%type <t> additiveExpression
%type <t> additiveOperator
%type <t> multiplicativeExpression
%type <t> multiplicativeOperator
%type <t> unaryExpression
%type <t> unaryExpressionNotPlusMinus
%type <t> postfixExpression
%type <t> postfixExpression2
%type <t> e4xAttributeIdentifier
%type <t> primaryExpression
%type <t> constant
%type <t> number
%type <t> newExpression
%type <t> fullNewSubexpression
%type <t> brackets
%type <t> encapsulatedExpression
%type <t> functionSignature
%type <t> functionCommon
%type <t> functionExpression
%type <t> ident


%%


/**
 * this is the start rule for this parser
 */
compilationUnit:
        as2CompilationUnit              {YY_TREE = $1; return(0);}
        | as3CompilationUnit            {YY_TREE = $1; return(0);}
        ;
as2CompilationUnit:
        importDefinitions as2Type
        ;
importDefinitions: importDefinitions importDefinition
        | { $$ = NULL; }
        ;
as2Type: annotations modifiers as2ClassDefinition
        | annotations modifiers as2InterfaceDefinition
        ;
as3CompilationUnit:
```

```
        packageDecl packageBlockEntries EOFX    { $$ = _TCN ("as3CompilationUnit",2,
$1,$2); }
        ;
packageBlockEntries:
        packageBlockEntries packageBlockEntry   { $$ = _TCN ("packageBlockEntries",2
,$1,$2); }
        |                                       { $$ = NULL; }
        ;
packageDecl:
        PACKAGE identifier packageBlock         { $$ = _TCN ("packageDecl",3, $1,$2,
$3); }
        | PACKAGE packageBlock                  { $$ = _TCN ("packageDecl",2,$1,$2);
 }
        ;
packageBlock
        :       LCURLY packageBlockEntries RCURLY       { $$ = _TCN ("packageBlock",
3, $1,$2,$3); }
        ;
packageBlockEntry:
        importDefinition                                { $$ = _TCN ("packageBlockEn
try",1, $1); }
        |   annotations modifiers classDefinition       { $$ = _TCN ("packageBlockEn
try",3, $1,$2,$3); }
        |   annotations modifiers interfaceDefinition   { $$ = _TCN ("packageBlockEn
try",3, $1,$2,$3); }
        |   annotations modifiers variableDefinition    { $$ = _TCN ("packageBlockEn
try",3, $1,$2,$3); }
        |   annotations modifiers methodDefinition      { $$ = _TCN ("packageBlockEn
try",3, $1,$2,$3); }
        |   annotations modifiers namespaceDefinition   { $$ = _TCN ("packageBlockEn
try",3, $1,$2,$3); }
        |   annotations modifiers useNamespaceDirective { $$ = _TCN ("packageBlockEn
try",3, $1,$2,$3); }
        |   SEMI                                        { $$ = _TCN ("packageBlockEn
try",1, $1); }
        ;

importDefinition
        :       IMPORT identifierStar semi      { $$ = _TCN ("importDefinition",3,$1
,$2,$3); }
        ;

semi    :       SEMI                            { $$ = _TCN ("semi",1,$1); }
        ;

classDefinition:
        /*
        CLASS ident classExtendsClause implementsClause typeBlock
        | CLASS ident classExtendsClause implementsClause typeBlock
        */
        CLASS ident classExtendsClause typeBlock        { $$ = _TCN ("classDefinitio
n",4, $1,$2,$3,$4); }
        ;

as2ClassDefinition:
        CLASS identifier classExtendsClause implementsClause typeBlock  { $$ = _TCN
("as2ClassDefinition",5, $1,$2,$3,$4,$5); }
        ;

interfaceDefinition:
        INTERFACE ident interfaceExtendsClause typeBlock        { $$ = _TCN ("interf
aceDefinition",4, $1,$2,$3,$4); }
        ;

as2InterfaceDefinition:
        INTERFACE identifier interfaceExtendsClause typeBlock   { $$ = _TCN ("as2Int
erfaceDefinition",4, $1,$2,$3,$4); }
        ;

classExtendsClause:
        EXTENDS identifier      { $$ = _TCN ("classExtendsClause",2, $1,$2); }
        | /*empty*/             {$$ = NULL; }
        ;
```

```
interfaceExtendsClause:
        EXTENDS identifier commaIdentifiers     { $$ = _TCN ("interfaceExtendsClause
",3, $1,$2,$3); }
        | { $$ = NULL; }
        ;

commaIdentifiers:
        COMMA identifier commaIdentifiers       { $$ = _TCN ("commaIdentifiers",2, $
1,$2,$3); }
        | /*empty*/                             { $$ = NULL; }
        ;

implementsClause:
        IMPLEMENTS identifier commaIdentifiers  { $$ = _TCN ("implementsClause",2, $
1,$2,$3); }
        |                                       { $$ = NULL; }
        ;

typeBlock:
        LCURLY typeBlockEntries RCURLY          { $$ = _TCN ("typeBlock",3, $1,$2,$3
); }
        ;

typeBlockEntries:
        typeBlockEntry typeBlockEntries         { $$ = _TCN ("typeBlockEntries",2, $
1,$2); }
        |       /*empty*/                       { $$ = NULL; }
        ;

typeBlockEntry:
        annotations modifiers variableDefinition        { $$ = _TCN ("typeBlockEntry
",3, $1,$2,$3); }
        | annotations modifiers methodDefinition        { $$ = _TCN ("typeBlockEntry
",3, $1,$2,$3); }
        | importDefinition                              { $$ = _TCN ("typeBlockEntry
",1, $1); }
        | as2IncludeDirective                           { $$ = _TCN ("typeBlockEntry
",1, $1); }
        ;

as2IncludeDirective:
        INCLUDE_DIRECTIVE STRING_LITERAL        { $$ = _TCN ("as2IncludeDirective",2
, $1,$2); }
        ;

includeDirective:
        INCLUDE STRING_LITERAL semi             { $$ = _TCN ("as2IncludeDirective",3
, $1,$2,$3); }
        ;

blockOrSemi:
        block           { $$ = _TCN ("blockOrSemi",1, $1); }
        | semi          { $$ = _TCN ("blockOrSemi",1, $1); }
        ;

optionalTypeExpression:
        typeExpression  { $$ = _TCN ("optionalTypeExpression",1, $1); }
        | /*empty*/     { $$ = NULL; }
        ;

methodDefinition:
        FUNCTION optionalAccessorRole ident parameterDeclarationList optionalTypeExp
ression blockOrSemi
        { $$ = _TCN ("methodDefinition",6, $1,$2,$3,$4,$5,$6); }
        ;

optionalAccessorRole:
        accessorRole    { $$ = _TCN ("optionalAcessorRole",1, $1); }
        | /*empty*/     { $$ = NULL; }
        ;

accessorRole:
```

```
        GET             { $$ = _TCN ("accessorRole",1, $1); }
        | SET           { $$ = _TCN ("accessorRole",1, $1); }
        ;

namespaceDefinition:
        NAMESPACE ident { $$ = _TCN ("namespaceDefinition",2, $1,$2); }
        ;

useNamespaceDirective:
        USE NAMESPACE ident semi        { $$ = _TCN ("useNamespaceDirective",4, $1,$
2,$3,$4); }
        ;

commaVariableDeclarators:
        COMMA variableDeclarator commaVariableDeclarators       { $$ = _TCN ("commaV
ariableDeclaratios",3, $1,$2,$3); }
        | /*empty*/                                             { $$ = NULL; }
        ;

variableDefinition:
        varOrConst variableDeclarator commaVariableDeclarators semi { $$ = _TCN ("va
riableDefinition",4, $1,$2,$3,$4); }
        ;

varOrConst:
        VAR             { $$ = _TCN ("varOrConst",1, $1); }
        | CONST         { $$ = _TCN ("varOrConst",1, $1); }
        ;

optionalVariableInitializer:
        variableInitializer     { $$ = _TCN ("optionalVariableInitializer",1, $1); }
        | /*empty*/             { $$ = NULL; }
        ;

variableDeclarator:
        ident optionalTypeExpression optionalVariableInitializer        { $$ = _TCN
("variableDeclarator",3, $1,$2,$3); }
        | optionalTypeExpression optionalVariableInitializer            { $$ = _TCN
("variableDeclarator",2, $1,$2); }
        ;

declaration:
        varOrConst variableDeclarator declarationTail   { $$ = _TCN ("declaration",3
, $1,$2,$3); }
        ;

declarationTail:
        commaVariableDeclarators        { $$ = _TCN ("declarationTail",1, $1); }
        ;

variableInitializer:
        ASSIGN assignmentExpression { $$ = _TCN ("variableInitializer",1, $1,$2); }
        ;

commaParameterDeclaration:
        COMMA parameterDeclaration commaParameterDeclaration    { $$ = _TCN ("commaP
aramterDeclaration",3, $1,$2,$3); }
        | /*empty*/                                             { $$ = NULL; }
        ;

parameterDeclarationList:
        LPAREN RPAREN                                                   { $$ = _TCN
("parameterDeclarationList",2, $1,$2); }
        | LPAREN parameterDeclaration commaParameterDeclaration RPAREN  { $$ = _TCN
("parameterDeclarationList",4, $1,$2,$3,$4); }
        ;

parameterDeclaration:
        basicParameterDeclaration       { $$ = _TCN ("parameterDeclaration",1, $1);
}
        | parameterRestDeclaration      { $$ = _TCN ("parameterDeclaration",1, $1);
}
        ;
```

```
basicParameterDeclaration:
        ident optionalTypeExpression parameterDefault           { $$ = _TCN ("basicP
arameterDeclaration",3, $1,$2,$3); }
        | CONST ident optionalTypeExpression parameterDefault   { $$ = _TCN ("basicP
arameterDeclaration",4, $1,$2,$3,$4); }
        | ident optionalTypeExpression                          { $$ = _TCN ("basicP
arameterDeclaration",2, $1,$2); }
        | CONST ident optionalTypeExpression                    { $$ = _TCN ("basicP
arameterDeclaration",3, $1,$2,$3); }
        ;

parameterDefault:
        ASSIGN assignmentExpression     { $$ = _TCN ("parameterDefault",2, $1,$2); }
        ;

parameterRestDeclaration:
        REST ident      { $$ = _TCN ("parameterRestDeclaration",2, $1,$2); }
        | REST          { $$ = _TCN ("parameterRestDeclaration",1, $1); }
        ;

blockEntries:
        blockEntry blockEntries         { $$ = _TCN ("blockEntries",2, $1,$2); }
        | /*empty*/                     { $$ = NULL; }
        ;

block:
        LCURLY blockEntries RCURLY      { $$ = _TCN ("block",3, $1,$2,$3); }
        ;

blockEntry:
        statement       { $$ = _TCN ("blockEntry",1, $1); }
        ;

condition:
        LPAREN expression RPAREN        { $$ = _TCN ("condition",3, $1,$2,$3); }
        ;

statement
        :       superStatement          { $$ = _TCN ("statement",1, $1); }
        |       block                   { $$ = _TCN ("statement",1, $1); }
        |       declarationStatement    { $$ = _TCN ("statement",1, $1); }
        |       expressionStatement     { $$ = _TCN ("statement",1, $1); }
        |       ifStatement             { $$ = _TCN ("statement",1, $1); }
        |       forEachStatement        { $$ = _TCN ("statement",1, $1); }
        |       forStatement            { $$ = _TCN ("statement",1, $1); }
        |       whileStatement          { $$ = _TCN ("statement",1, $1); }
        |       doWhileStatement        { $$ = _TCN ("statement",1, $1); }
        |       withStatement           { $$ = _TCN ("statement",1, $1); }
        |       switchStatement         { $$ = _TCN ("statement",1, $1); }
        |       breakStatement          { $$ = _TCN ("statement",1, $1); }
        |       continueStatement       { $$ = _TCN ("statement",1, $1); }
        |       returnStatement         { $$ = _TCN ("statement",1, $1); }
        |       throwStatement          { $$ = _TCN ("statement",1, $1); }
        |       tryStatement            { $$ = _TCN ("statement",1, $1); }
        |       defaultXMLNamespaceStatement { $$ = _TCN ("statement",1, $1); }
        |       SEMI                    { $$ = _TCN ("statement",1, $1); }
        ;

superStatement:
        SUPER arguments semi            { $$ = _TCN ("superStatement",3, $1,$2,$3);
}
        ;

declarationStatement:
        declaration semi                { $$ = _TCN ("declarationStatement",2, $1,$2
); }
        ;

expressionStatement:
        expressionList semi             { $$ = _TCN ("expressionStatement",2, $1,$2)
; }
        ;
```

```
ifStatement:
        IF condition statement elseClause       { $$ = _TCN ("ifStatement",4, $1,$2,
$3,$4); }
        | IF condition statement                 { $$ = _TCN ("ifStatement",2, $1,$2,
$3); }
        ;

elseClause:
        ELSE statement          { $$ = _TCN ("elseClause",2, $1,$2); }
        ;

throwStatement:
        THROW expression semi   { $$ = _TCN ("throwStatement",3, $1,$2,$3); }
        ;

tryStatement:
        TRY block finallyBlock                  { $$ = _TCN ("tryStatement",3, $1,$2
,$3); }
        | TRY block catchBlocks finallyBlock    { $$ = _TCN ("tryStatement",4, $1,$2
,$3,$4); }
        | TRY block catchBlocks                 { $$ = _TCN ("tryStatement",3, $1,$2
,$3); }
        ;

catchBlocks:
        catchBlock catchBlocks  { $$ = _TCN ("catchBlocks",2, $1,$2); }
        | /*empty*/             { $$ = NULL; } ;

catchBlock:
        CATCH LPAREN ident optionalTypeExpression RPAREN block  { $$ = _TCN ("catchB
lock",6, $1,$2,$3,$4,$5,$6); }
        ;

finallyBlock:
        FINALLY block   { $$ = _TCN ("finallyBlock",2, $1,$2); }
        ;

returnStatement:
        RETURN expression semi  { $$ = _TCN ("returnStatement",3, $1,$2,$3); }
        | RETURN semi           { $$ = _TCN ("tryStatement",2, $1,$2); }
        ;

continueStatement:
        CONTINUE semi           { $$ = _TCN ("continueStatement",2, $1,$2); }
        ;

breakStatement:
        BREAK semi              { $$ = _TCN ("breakStatement",2, $1,$2); }
        ;

switchStatement:
        SWITCH condition switchBlock    { $$ = _TCN ("switchStatement",3, $1,$2,$3);
 }
        ;

switchBlock:
        LCURLY caseStatements RCURLY    { $$ = _TCN ("switchBlock",3, $1,$2,$3); }
        ;

caseStatements:
        caseStatements caseStatement    { $$ = _TCN ("caseStatements",2, $1,$2); }
        | /*empty*/                     { $$ = NULL; }
        ;

caseStatement:
        CASE expression COLON switchStatementList       { $$ = _TCN ("caseStatement"
,3, $1,$2,$3,$4); }
        ;

optionalDefaultStatement:
        defaultStatement        { $$ = _TCN ("optionalDefaultStatement",1, $1); }
        | /*empty*/             {$$ = NULL; } ;
```

```
defaultStatement
        :       DEFAULT COLON switchStatementList       { $$ = _TCN ("defaultStateme
nt",3, $1,$2,$3); }
        ;

switchStatementList:
        statements optionalDefaultStatement             { $$ = _TCN ("switchStatemen
tList",2, $1,$2); }
        ;

statements:
        statement statements    { $$ = _TCN ("statements",2, $1,$2); }
        | /*empty*/             { $$ = NULL }
        ;

forEachStatement:
        FOR EACH LPAREN forInClause RPAREN statement    { $$ = _TCN ("forEachStateme
nt",6, $1,$2,$3,$4,$5,$6); }
        ;

forStatement:
        FOR LPAREN forInClause RPAREN statement                 { $$ = _TCN ("forSta
tement",5, $1,$2,$3,$4,$5); }
        | FOR LPAREN traditionalForClause RPAREN statement      { $$ = _TCN ("forSta
tement",5, $1,$2,$3,$4,$5); }
        ;

traditionalForClause:
        forInit SEMI forCond SEMI forIter       { $$ = _TCN ("traditionalForClause",
5, $1,$2,$3,$4,$5); }
        ;

forInClause:
        forInClauseDecl IN forInClauseTail      { $$ = _TCN ("forInClause",3, $1,$2,
$3); }
        ;

forInClauseDecl:
        declaration             { $$ = _TCN ("forInClauseDecl",1, $1); }
        ;


forInClauseTail:
        expressionList          { $$ = _TCN ("expressionList",1, $1); }
        ;

forInit:
        declaration             { $$ = _TCN ("forInit",1, $1); }
        | expressionList        { $$ = _TCN ("forInit",1, $1); }
        | /*empty*/             { $$ = NULL; }
        ;

forCond :
        expressionList          { $$ = _TCN ("forInit",1, $1); }
        | /*empty*/             { $$ = NULL; }
        ;

forIter:
        expressionList          { $$ = _TCN ("forIter",1, $1); }
        | /*empty*/             { $$ = NULL; }
        ;

whileStatement:
        WHILE condition statement               { $$ = _TCN ("whileStatement",3, $1,
$2,$3); }
        ;

doWhileStatement:
        DO statement WHILE condition semi       { $$ = _TCN ("doWhileStatement",5, $
1,$2,$3,$4,$5); }
        ;
```

```
withStatement:
        WITH condition statement                { $$ = _TCN ("withStatement",3, $1,$
2,$3); }
        ;

defaultXMLNamespaceStatement:
        DEFAULT XML NAMESPACE ASSIGN expression semi            { $$ = _TCN ("defaul
tXMLNamespaceStatement",6, $1,$2,$3,$4,$5,$6); }
        ;

typeExpression
        :       COLON identifier                { $$ = _TCN ("typeExpression",2, $1,
$2); }
        |       COLON VOID                      { $$ = _TCN ("typeExpression",2, $1,
$2); }
        |       COLON STAR                      { $$ = _TCN ("typeExpression",2, $1,
$2); }
        ;

identifier:
        qualifiedIdent                  { $$ = _TCN ("identifier",1, $1); }
        ;

propertyIdentifier:
        STAR                    { $$ = _TCN ("propertyIdentifier",1, $1); }
        | ident                 { $$ = _TCN ("propertyIdentifier",1, $1); }
        ;

qualifier:
        propertyIdentifier      { $$ = _TCN ("qualifier",1, $1); }
        | reservedNamespace     { $$ = _TCN ("qualifier",1, $1); }
        ;

simpleQualifiedIdentifier
        :       propertyIdentifier                      { $$ = _TCN ("simpleQualifie
dIdentifier",1, $1); }
        |       qualifier DBL_COLON propertyIdentifier  { $$ = _TCN ("simpleQualifie
dIdentifier",3, $1,$2,$3); }
        |       qualifier DBL_COLON brackets            { $$ = _TCN ("simpleQualifie
dIdentifier",3, $1,$2,$3); }
        ;

expressionQualifiedIdentifier:
        encapsulatedExpression DBL_COLON propertyIdentifier     { $$ = _TCN ("expres
sionQualifiedIdentifier",3, $1,$2,$3); }
        | encapsulatedExpression DBL_COLON brackets             { $$ = _TCN ("expres
sionQualifiedIdentifier",3, $1,$2,$3); }
        ;

nonAttributeQualifiedIdentifier:
        simpleQualifiedIdentifier               { $$ = _TCN ("nonAttributeQualifiedI
dentifier",1, $1); }
        | expressionQualifiedIdentifier         { $$ = _TCN ("nonAttributeQualifiedI
dentifier",1, $1); }
        ;

qualifiedIdentifier
        :       e4xAttributeIdentifier          { $$ = _TCN ("qualifiedIdentifier",1
, $1); }
        |       nonAttributeQualifiedIdentifier { $$ = _TCN ("qualifiedIdentifier",1
, $1); }
        ;

qualifiedIdent
        :       namespaceName DBL_COLON ident   { $$ = _TCN ("qualifiedIdent",3, $1,
$2,$3); }
        |       ident                           { $$ = _TCN ("qualifiedIdent",1, $1)
; }
        ;

namespaceName:
        IDENT                   { $$ = _TCN ("namespaceName",1, $1); }
        | reservedNamespace     { $$ = _TCN ("namespaceName",1, $1); }
```

```
        ;

reservedNamespace
        :       PUBLIC          { $$ = _TCN ("reservedNamespace",1, $1); }
        |       PRIVATE         { $$ = _TCN ("reservedNamespace",1, $1); }
        |       PROTECTED       { $$ = _TCN ("reservedNamespace",1, $1); }
        |       INTERNAL        { $$ = _TCN ("reservedNamespace",1, $1); }
        ;

dotIdents:
        DOT ident dotIdents     { $$ = _TCN ("dotIdents",3, $1,$2,$3); }
        | /*empty*/             { $$ = NULL; }
        ;

identifierStar:
        ident dotIdents DOT STAR        { $$ = _TCN ("identifierStar",1, $1); }
        | ident dotIdents               { $$ = _TCN ("identifierStar",1, $1); }
        ;

annotations:
        annotation annotations          { $$ = _TCN ("annotations",2, $1,$2); }
        | includeDirective annotations  { $$ = _TCN ("annotations",2, $1,$2); }
        | /*empty*/                     { $$ = NULL; }
        ;

annotation:
        LBRACK ident annotationParamList RBRACK         { $$ = _TCN ("annotation",4,
 $1,$2,$3,$4); }
        | LBRACK ident RBRACK                           { $$ = _TCN ("annotation",3,
 $1,$2,$3); }
        ;

moreAnnotationParams:
        COMMA annotationParam moreAnnotationParams      { $$ = _TCN ("moreAnnoationP
arams",4, $1,$2,$3); }
        | /*empty*/                                     { $$ = NULL; }
        ;

annotationParamList:
        LPAREN annotationParam moreAnnotationParams RPAREN      { $$ = _TCN ("annota
tionParamList",4, $1,$2,$3,$4); }
        | LPAREN RPAREN                                         { $$ = _TCN ("annota
tionParamList",4, $1,$2); }
        ;

annotationParam:
        ident ASSIGN constant           { $$ = _TCN ("annotationParam",3, $1,$2,$3);
 }
        | constant                      { $$ = _TCN ("annotationParam",1, $1); }
        | ident                         { $$ = _TCN ("annotationParam",1, $1); }
        ;

modifiers:
        modifier modifiers      { $$ = _TCN ("modifiers",2, $1,$2); }
        | /*empty*/             { $$ = NULL; }
        ;

modifier
        :       namespaceName   { $$ = _TCN ("modifier",1, $1); }
        |       STATIC          { $$ = _TCN ("modifier",1, $1); }
        |       FINAL           { $$ = _TCN ("modifier",1, $1); }
        |       ENUMERABLE      { $$ = _TCN ("modifier",1, $1); }
        |       EXPLICIT        { $$ = _TCN ("modifier",1, $1); }
        |       OVERRIDE        { $$ = _TCN ("modifier",1, $1); }
        |       DYNAMIC         { $$ = _TCN ("modifier",1, $1); }
        |       INTRINSIC       { $$ = _TCN ("modifier",1, $1); }
        ;

arguments:
        LPAREN expressionList RPAREN    { $$ = _TCN ("arguments",3, $1,$2,$3); }
        | LPAREN RPAREN                 { $$ = _TCN ("arguments",2, $1,$2); }
        ;
```

```
arrayLiteral:
        LBRACK elementList RBRACK       { $$ = _TCN ("arrayLiteral",3, $1,$2,$3); }
        | LBRACK  RBRACK                { $$ = _TCN ("arrayLiteral",2, $1,$2); }
        ;

elementList:
        COMMA                   { $$ = _TCN ("elementList",1, $1); }
        | nonemptyElementList   { $$ = _TCN ("elementList",1, $1); }
        ;

moreAssignmentExpressions:
        COMMA assignmentExpression moreAssignmentExpressions    { $$ = _TCN ("moreAs
signmentExpressions",3, $1,$2,$3); }
        | /*empty*/                                             { $$ = NULL; }
        ;

nonemptyElementList:
        assignmentExpression moreAssignmentExpressions          { $$ = _TCN ("nonemp
tyElementList",1, $1,$2); }
        ;

element :
        assignmentExpression    { $$ = _TCN ("element",1, $1); }
        ;

objectLiteral:
        LCURLY fieldList RCURLY { $$ = _TCN ("objectLiteral",2, $1,$2,$3); }
        | LCURLY  RCURLY        { $$ = _TCN ("objectLiteral",2, $1,$2); }
        ;

moreLiteralFields:
        COMMA literalField moreLiteralFields    { $$ = _TCN ("moreLiteralFields",3,
$1,$2,$3); }
        | COMMA                                 { $$ = _TCN ("moreLiteralFields",1,
$1); }
        |  /*empty*/                            { $$ = NULL; }
        ;

fieldList:
        literalField moreLiteralFields          { $$ = _TCN ("fieldList",2, $1,$2);
}
        ;

literalField:
        fieldName COLON element         { $$ = _TCN ("literalFields",3, $1,$2,$3); }
        ;

fieldName:
        ident           { $$ = _TCN ("fieldName",1, $1); }
        | number        { $$ = _TCN ("fieldName",1, $1); }
        ;

expression:
        assignmentExpression    { $$ = _TCN ("assignmentExpression",1, $1); }
        ;

expressionList:
        assignmentExpression moreAssignmentExpressions  { $$ = _TCN ("expressionList
",2, $1,$2); }
        ;

assignmentExpression:
        conditionalExpression assignmentOperator assignmentExpression   { $$ = _TCN
("assignmentExpression",3, $1,$2,$3); }
        | conditionalExpression                                         { $$ = _TCN
("assignmentExpression",1, $1); }
        ;

assignmentOperator:
                ASSIGN          { $$ = _TCN ("assignmentOperator",1, $1); }
        |       STAR_ASSIGN     { $$ = _TCN ("assignmentOperator",1, $1); }
        |       DIV_ASSIGN      { $$ = _TCN ("assignmentOperator",1, $1); }
        |       MOD_ASSIGN      { $$ = _TCN ("assignmentOperator",1, $1); }
```

```
        |       PLUS_ASSIGN     { $$ = _TCN ("assignmentOperator",1, $1); }
        |       MINUS_ASSIGN    { $$ = _TCN ("assignmentOperator",1, $1); }
        |       SL_ASSIGN       { $$ = _TCN ("assignmentOperator",1, $1); }
        |       SR_ASSIGN       { $$ = _TCN ("assignmentOperator",1, $1); }
        |       BSR_ASSIGN      { $$ = _TCN ("assignmentOperator",1, $1); }
        |       BAND_ASSIGN     { $$ = _TCN ("assignmentOperator",1, $1); }
        |       BXOR_ASSIGN     { $$ = _TCN ("assignmentOperator",1, $1); }
        |       BOR_ASSIGN      { $$ = _TCN ("assignmentOperator",1, $1); }
        |       LAND_ASSIGN     { $$ = _TCN ("assignmentOperator",1, $1); }
        |       LOR_ASSIGN      { $$ = _TCN ("assignmentOperator",1, $1); }
        ;

conditionalExpression:
                logicalOrExpression                                     { $$ = _TCN
("conditionalExpression",1, $1); }
        |       logicalOrExpression QUESTION conditionalSubExpression   { $$ = _TCN
("conditionalExpression",3, $1,$2,$3); }
        ;

conditionalSubExpression:
        assignmentExpression COLON assignmentExpression                 { $$ = _TCN
("conditionalSubExpression",1, $1,$2,$3); }
        ;

logicalOrExpression:
        logicalAndExpression                                            { $$ = _TCN
("logicalOrExpression",1, $1); }
        | logicalAndExpression logicalOrOperator logicalOrExpression    { $$ = _TCN
("logicalOrExpression",3, $1,$2,$3); }
        ;

logicalOrOperator:
        LOR             { $$ = _TCN ("logicalOrOperator",1, $1); }
        | OR            { $$ = _TCN ("logicalOrOperator",1, $1); }
        ;

logicalAndExpression:
                bitwiseOrExpression                                             { $$
 = _TCN ("logicalAndExpression",1, $1); }
        |       bitwiseOrExpression logicalAndOperator logicalAndExpression     { $$
 = _TCN ("logicalAndExpression",1, $1); }
        ;

logicalAndOperator:
        LAND            { $$ = _TCN ("logicalAndOperator",1, $1); }
        | AND           { $$ = _TCN ("logicalAndOperator",1, $1); }
        ;

bitwiseOrExpression:
                bitwiseXorExpression                            { $$ = _TCN ("bitwis
eOrExpression",1, $1); }
        |       bitwiseXorExpression BOR bitwiseOrExpression    { $$ = _TCN ("bitwis
eOrExpression",3, $1,$2,$3); }
        ;

bitwiseXorExpression:
                bitwiseAndExpression                            { $$ = _TCN ("bitwis
eXorExpression",1, $1); }
        |       bitwiseAndExpression BXOR bitwiseXorExpression  { $$ = _TCN ("bitwis
eXorExpression",3, $1,$2,$3); }
        ;

bitwiseAndExpression:
                equalityExpression                              { $$ = _TCN ("bitwis
eAndExpression",1, $1); }
        |       equalityExpression BAND bitwiseAndExpression    { $$ = _TCN ("bitwis
eAndExpression",3, $1,$2,$3); }
        ;

equalityExpression:
                relationalExpression                                            { $$
 = _TCN ("equalityExpression",1, $1); }
        |       relationalExpression equalityOperator equalityExpression       { $$
```

```
 = _TCN ("equalityExpression",3, $1,$2,$3); }
        ;

equalityOperator:
                STRICT_EQUAL            { $$ = _TCN ("equalityOperator",1, $1); }
        |       STRICT_NOT_EQUAL        { $$ = _TCN ("equalityOperator",1, $1); }
        |       NOT_EQUAL               { $$ = _TCN ("equalityOperator",1, $1); }
        |       EQUAL                   { $$ = _TCN ("equalityOperator",1, $1); }
        ;

relationalExpression:
                shiftExpression                                                 { $$ = _TCN
("rationalExpression",1, $1); }
        |       shiftExpression relationalOperator relationalExpression { $$ = _TCN
("rationalExpression",3, $1,$2,$3); }
        ;

relationalOperator:
                IN              { $$ = _TCN ("rationalOperator",1, $1); }
        |       LT              { $$ = _TCN ("rationalOperator",1, $1); }
        |       GT              { $$ = _TCN ("rationalOperator",1, $1); }
        |       LE              { $$ = _TCN ("rationalOperator",1, $1); }
        |       GE              { $$ = _TCN ("rationalOperator",1, $1); }
        |       IS              { $$ = _TCN ("rationalOperator",1, $1); }
        |       AS              { $$ = _TCN ("rationalOperator",1, $1); }
        |       INSTANCEOF      { $$ = _TCN ("rationalOperator",1, $1); }
        ;

shiftExpression:
                additiveExpression                                              { $$ = _TCN
("shiftExpression",1, $1); }
        |       additiveExpression shiftOperator shiftExpression        { $$ = _TCN
("shiftExpression",3, $1,$2,$3); }
        ;

shiftOperator:
                SL      { $$ = _TCN ("shiftOperator",1, $1); }
        |       SR      { $$ = _TCN ("shiftOperator",1, $1); }
        |       BSR     { $$ = _TCN ("shiftOperator",1, $1); }
        ;

additiveExpression:
                multiplicativeExpression                                        { $$
 = _TCN ("additiveExpression",1, $1); }
        |       multiplicativeExpression additiveOperator additiveExpression    { $$
 = _TCN ("additiveExpression",3, $1,$2,$3); }
        ;

additiveOperator:
                PLUS    { $$ = _TCN ("additiveOperator",1, $1); }
        |       MINUS   { $$ = _TCN ("additiveOperator",1, $1); }
        ;

multiplicativeExpression:
                unaryExpression                                                 { $$
 = _TCN ("multiplicativeExpression",1, $1); }
        |       unaryExpression multiplicativeOperator multiplicativeExpression { $$
 = _TCN ("multiplicativeExpression",3, $1,$2,$3); }
        ;

multiplicativeOperator:
                STAR    { $$ = _TCN ("multiplicativeOperator",1, $1); }
        |       DIV     { $$ = _TCN ("multiplicativeOperator",1, $1); }
        |       MOD     { $$ = _TCN ("multiplicativeOperator",1, $1); }
        ;

unaryExpression:
                INC unaryExpression             { $$ = _TCN ("unaryExpression",2, $1
,$2); }
        |       DEC unaryExpression             { $$ = _TCN ("unaryExpression",2, $1
,$2); }
        |       MINUS unaryExpression           { $$ = _TCN ("unaryExpression",2, $1
,$2); }
```

```
        |       PLUS unaryExpression            { $$ = _TCN ("unaryExpression",2, $1
,$2); }
        |       unaryExpressionNotPlusMinus     { $$ = _TCN ("unaryExpression",1, $1
); }
        ;

unaryExpressionNotPlusMinus:
                DELETE postfixExpression        { $$ = _TCN ("unaryExpressionNotPlus
Minus",2, $1,$2); }
        |       VOID unaryExpression            { $$ = _TCN ("unaryExpressionNotPlus
Minus",2, $1,$2); }
        |       TYPEOF unaryExpression          { $$ = _TCN ("unaryExpressionNotPlus
Minus",2, $1,$2); }
        |       LNOT unaryExpression            { $$ = _TCN ("unaryExpressionNotPlus
Minus",2, $1,$2); }
        |       BNOT unaryExpression            { $$ = _TCN ("unaryExpressionNotPlus
Minus",2, $1,$2); }
        |       postfixExpression               { $$ = _TCN ("unaryExpressionNotPlus
Minus",1, $1); }
        ;

postfixExpression:
        postfixExpression2                      { $$ = _TCN ("postfixExpression",1,
$1); }
        |  postfixExpression2 INC               { $$ = _TCN ("postfixExpression",2,
$1,$2); }
        |  postfixExpression2 DEC               { $$ = _TCN ("postfixExpression",2,
$1,$2); }
        ;

postfixExpression2:
                primaryExpression                               { $$ = _TCN ("postfi
xExpression2",1, $1); }
        |       postfixExpression LBRACK expression RBRACK      { $$ = _TCN ("postfi
xExpression2",4, $1,$2,$3,$4); }
        |       postfixExpression E4X_DESC qualifiedIdentifier  { $$ = _TCN ("postfi
xExpression2",3, $1,$2,$3); }
        |       postfixExpression DOT LPAREN expression RPAREN  { $$ = _TCN ("postfi
xExpression2",5, $1,$2,$3,$4,$5); }
        |       postfixExpression DOT e4xAttributeIdentifier    { $$ = _TCN ("postfi
xExpression2",3, $1,$2,$3); }
        |       postfixExpression DOT STAR                      { $$ = _TCN ("postfi
xExpression2",3, $1,$2,$3); }
        |       postfixExpression arguments                     { $$ = _TCN ("postfi
xExpression2",2, $1,$2); }
        ;

e4xAttributeIdentifier:
                E4X_ATTRI qualifiedIdent                { $$ = _TCN ("e4xAttributeId
entifier",2, $1,$2); }
        |       E4X_ATTRI STAR                          { $$ = _TCN ("e4xAttributeId
entifier",2, $1,$2); }
        |       E4X_ATTRI LBRACK expression RBRACK      { $$ = _TCN ("e4xAttributeId
entifier",2, $1,$2); }
        ;

primaryExpression:
                UNDEFINED               { $$ = _TCN ("primaryExpression",1, $1); }
        |       constant                { $$ = _TCN ("primaryExpression",1, $1); }
        |       arrayLiteral            { $$ = _TCN ("primaryExpression",1, $1); }
        |       objectLiteral           { $$ = _TCN ("primaryExpression",1, $1); }
        |       functionExpression      { $$ = _TCN ("primaryExpression",1, $1); }
        |       newExpression           { $$ = _TCN ("primaryExpression",1, $1); }
        |       encapsulatedExpression  { $$ = _TCN ("primaryExpression",1, $1); }
        |       e4xAttributeIdentifier  { $$ = _TCN ("primaryExpression",1, $1); }
        |       qualifiedIdent          { $$ = _TCN ("primaryExpression",1, $1); }
        ;


/*
propOrIdent:
                DOT qualifiedIdent      { $$ = _TCN ("propOrIdent",2, $1,$2); }
        ;
```

```
*/

constant:
                number                  { $$ = _TCN ("constant",1, $1); }
        |       STRING_LITERAL          { $$ = _TCN ("constant",1, $1); }
        |       TRUE                    { $$ = _TCN ("constant",1, $1); }
        |       FALSE                   { $$ = _TCN ("constant",1, $1); }
        |       NULL_VAL                { $$ = _TCN ("constant",1, $1); }
        ;

number:
                HEX_LITERAL             { $$ = _TCN ("number",1, $1); }
        |       DECIMAL_LITERAL         { $$ = _TCN ("number",1, $1); }
        |       OCTAL_LITERAL           { $$ = _TCN ("number",1, $1); }
        |       FLOAT_LITERAL           { $$ = _TCN ("number",1, $1); }
        ;

newExpression:
                NEW fullNewSubexpression arguments      { $$ = _TCN ("newExpression"
,3, $1,$2,$3); }
        |       NEW fullNewSubexpression                { $$ = _TCN ("newExpression"
,2, $1,$2); }
        ;

fullNewSubexpression:
        primaryExpression                               { $$ = _TCN ("fullNewSubexpr
ession",1, $1); }
        | fullNewSubexpression DOT qualifiedIdent       { $$ = _TCN ("fullNewSubexpr
ession",3, $1,$2,$3); }
        | fullNewSubexpression brackets                 { $$ = _TCN ("fullNewSubexpr
ession",3, $1,$2); }
        ;

brackets:
        LBRACK expressionList RBRACK    { $$ = _TCN ("brackets",1, $1,$2,$3); }
        ;

encapsulatedExpression:
        LPAREN assignmentExpression RPAREN      { $$ = _TCN ("encapsulatedExpression
",3, $1,$2,$3); }
        ;

functionSignature:
        parameterDeclarationList optionalTypeExpression         { $$ = _TCN ("functi
onSignature",2, $1,$2); }
        ;

functionCommon:
        functionSignature block         { $$ = _TCN ("functionCommon",2, $1,$2); }
        ;

functionExpression:
                FUNCTION IDENT functionCommon   { $$ = _TCN("functionExpression",3,
$1, $2, $3); }
        |       FUNCTION functionCommon         { $$ = _TCN("functionExpression",3,
$1, $2); }
        ;


ident
        :       IDENT           { $$ = _TCN ("ident",1, $1); }
        |       USE             { $$ = _TCN ("ident",1, $1); }
        |       XML             { $$ = _TCN ("ident",1, $1); }
        |       DYNAMIC         { $$ = _TCN ("ident",1, $1); }
        |       NAMESPACE       { $$ = _TCN ("ident",1, $1); }
        |       IS              { $$ = _TCN ("ident",1, $1); }
        |       AS              { $$ = _TCN ("ident",1, $1); }
        |       GET             { $$ = _TCN ("ident",1, $1); }
        |       SET             { $$ = _TCN ("ident",1, $1); }
        ;
```