# C Compiler

## Introduction

### Parsing

Parsing is the process of determine whether the code is correct …

MainParser.gppg

%namespace CCompiler\_Main

%partial

%using CCompiler;

%using System.Numerics;

%{

public static Stack<Specifier> SpecifierStack = new Stack<Specifier>();

public static Stack<BigInteger> EnumValueStack = new Stack<BigInteger>();

public static int CallDepth = 0;

%}

The first part is the tokens of the grammar

%token AUTO REGISTER STATIC EXTERN TYPEDEF CONSTANT VOLATILE

SIGNED UNSIGNED CHAR WCHAR\_T SHORT INT LONG

FLOAT DOUBLE VOID ENUM STRUCT UNION

PLUS MINUS DIVIDE MODULO INCREMENT DECREMENT

EQUAL NOT\_EQUAL LESS\_THAN LESS\_THAN\_EQUAL GREATER\_THAN

GREATER\_THAN\_EQUAL LEFT\_SHIFT RIGHT\_SHIFT

ASSIGN ADD\_ASSIGN SUBTRACT\_ASSIGN MULTIPLY\_ASSIGN

DIVIDE\_ASSIGN MODULO\_ASSIGN LEFT\_SHIFT\_ASSIGN

RIGHT\_SHIFT\_ASSIGN AND\_ASSIGN IOR\_ASSIGN XOR\_ASSIGN

COLON COMMA SEMICOLON ELLIPSE DOT ARROW ASTERRISK AMPERSAND

LEFT\_PAREN RIGHT\_PAREN LEFT\_BLOCK RIGHT\_BLOCK

LEFT\_SQUARE RIGHT\_SQUARE SIZEOF

LOGICAL\_OR LOGICAL\_AND LOGICAL\_NOT BITWISE\_XOR BITWISE\_IOR

BITWISE\_NOT QUESTION\_MARK IF ELSE SWITCH CASE DEFAULT

FOR WHILE DO CONTINUE BREAK RETURN GOTO INTERRUPT

JUMP\_REGISTER SYSCALL CARRY\_FLAG

%union {

public string name;

public Register register;

public CCompiler.Type type;

public List<CCompiler.Type> type\_list;

public Sort sort;

public Symbol symbol;

public IDictionary<string,Symbol> symbol\_map;

public ISet<Pair<Symbol,bool>> symbol\_bool\_pair\_set;

public Pair<Symbol,bool> symbol\_bool\_pair;

public Pair<string,Symbol> string\_symbol\_pair;

public List<Pair<string,Symbol>> string\_symbol\_pair\_list;

public List<string> string\_list;

public Declarator declarator;

public List<Declarator> declarator\_list;

public MiddleOperator middleOperator;

public Expression expression;

public List<Expression> expression\_list;

public Statement statement;

public Pair<List<Pair<string,Symbol>>,Boolean> parameter\_pair;

public List<MiddleCode> middle\_code\_list;

public object obj;

public List<object> object\_list;

}

%token <name> NAME

%token <register> REGISTER\_NAME

%token <type> TYPEDEF\_NAME

%token <symbol> VALUE

%type <obj> declaration\_specifier declaration\_specifier\_list\_x

%type <object\_list> declaration\_specifier\_list

%type <name> optional\_name

%type <type> struct\_or\_union\_specifier

%type <sort> struct\_or\_union

%type <type> enum\_specifier

%type <symbol\_bool\_pair\_set> enum\_list

%type <symbol\_bool\_pair> enum

%type <middle\_code\_list> declarator\_list

init\_bitfield\_declarator declaration

%type <declarator> declarator direct\_declarator

%type <type\_list> optional\_pointer\_list pointer\_list

%type <type> pointer

%type <parameter\_pair> optional\_parameter\_ellipse\_list

parameter\_ellipse\_list

%type <string\_symbol\_pair\_list> parameter\_list

%type <string\_symbol\_pair> parameter\_declaration

%type <string\_list> optional\_name\_list identifier\_list

%type <object\_list> initializer\_list

%type <obj> initializer

%type <type> type\_name

%type <declarator> abstract\_declarator direct\_abstract\_declarator

%type <middleOperator> equality\_operator relation\_operator

shift\_operator shift\_assignment\_operator

multiply\_assignment\_operator

bitwise\_assignment\_operator

multiply\_operator prefix\_add\_operator

increment\_operator

%type <expression> optional\_constant\_integral\_expression

constant\_integral\_expression optional\_expression

expression assignment\_expression

condition\_expression logical\_or\_expression

logical\_and\_expression bitwise\_ior\_expression

bitwise\_xor\_expression bitwise\_and\_expression

equality\_expression relation\_expression

shift\_expression add\_expression

multiply\_expression cast\_expression

prefix\_expression postfix\_expression

primary\_expression

%type <statement> optional\_statement\_list statement

closed\_statement opened\_statement

%type <expression\_list> optional\_argument\_expression\_list

argument\_expression\_list

%start translation\_unit

%%

The first rule of the grammer is translation\_unit. It represent the code of a single file, and is made up by one or several exertal declarations, which in turn is a function definition, a function prototype, or a global varaible or constant.

translation\_unit:

external\_declaration

| translation\_unit external\_declaration;

external\_declaration:

function\_definition

| declaration;

function\_definition:

declarator {

MiddleCodeGenerator.FunctionHeader(null, $1);

}

optional\_declaration\_list {

MiddleCodeGenerator.CheckFunctionDefinition();

}

LEFT\_BLOCK optional\_statement\_list RIGHT\_BLOCK {

MiddleCodeGenerator.BackpatchGoto();

MiddleCodeGenerator.FunctionEnd($6);

}

| declaration\_specifier\_list\_x declarator {

MiddleCodeGenerator.FunctionHeader

(SpecifierStack.Pop(), $2);

}

optional\_declaration\_list {

MiddleCodeGenerator.CheckFunctionDefinition();

}

LEFT\_BLOCK optional\_statement\_list RIGHT\_BLOCK {

MiddleCodeGenerator.BackpatchGoto();

MiddleCodeGenerator.FunctionEnd($7);

};

optional\_declaration\_list:

/\* Empty \*/

| optional\_declaration\_list declaration;

declaration:

declaration\_specifier\_list SEMICOLON {

SpecifierStack.Push(new Specifier($1, false));

$$ = new List<MiddleCode>();

}

| declaration\_specifier\_list\_x declarator\_list SEMICOLON {

SpecifierStack.Pop();

$$ = $2;

};

declaration\_specifier\_list\_x:

declaration\_specifier\_list {

SpecifierStack.Push(new Specifier($1, false));

};

declaration\_specifier\_list:

declaration\_specifier {

List<object> specList = new List<object>();

specList.Add($1);

$$ = specList;

}

| declaration\_specifier declaration\_specifier\_list {

List<object> specList = $2;

specList.Add($1);

$$ = specList;

};

declaration\_specifier:

CONSTANT { $$ = Mask.Constant; }

| VOLATILE { $$ = Mask.Volatile; }

| AUTO { $$ = Mask.Auto; }

| REGISTER { $$ = Mask.Register; }

| STATIC { $$ = Mask.Static; }

| EXTERN { $$ = Mask.Extern; }

| TYPEDEF { $$ = Mask.Typedef; }

| VOID { $$ = Mask.Void; }

| CHAR { $$ = Mask.Char; }

| SHORT { $$ = Mask.Short; }

| INT { $$ = Mask.Int; }

| LONG { $$ = Mask.Long; }

| FLOAT { $$ = Mask.Float; }

| DOUBLE { $$ = Mask.Double; }

| SIGNED { $$ = Mask.Signed; }

| UNSIGNED { $$ = Mask.Unsigned; }

| struct\_or\_union\_specifier { $$ = $1; }

| enum\_specifier { $$ = $1; }

| TYPEDEF\_NAME { $$ = $1; };

struct\_or\_union\_specifier:

struct\_or\_union optional\_name {

MiddleCodeGenerator.StructUnionHeader($2, $1);

SymbolTable.CurrentTable =

new SymbolTable(SymbolTable.CurrentTable, (Scope) $1);

}

LEFT\_BLOCK declaration\_list RIGHT\_BLOCK {

$$ = MiddleCodeGenerator.StructUnionSpecifier($2, $1);

SymbolTable.CurrentTable =

SymbolTable.CurrentTable.ParentTable;

}

| struct\_or\_union NAME {

$$ = MiddleCodeGenerator.LookupStructUnionSpecifier($2, $1);

};

struct\_or\_union:

STRUCT { $$ = Sort.Struct; }

| UNION { $$ = Sort.Union; };

optional\_name:

/\* Empty \*/ { $$ = null; }

| NAME { $$ = $1; };

declaration\_list:

declaration

| declaration\_list declaration;

enum\_specifier:

ENUM optional\_name {

EnumValueStack.Push(((BigInteger) 0));

}

LEFT\_BLOCK enum\_list RIGHT\_BLOCK {

EnumValueStack.Pop();

$$ = MiddleCodeGenerator.EnumSpecifier($2, $5);

}

| ENUM NAME {

$$ = MiddleCodeGenerator.LookupEnum($2);

};

enum\_list:

enum {

ISet<Pair<Symbol,bool>> memberSet =

new HashSet<Pair<Symbol,bool>>();

memberSet.Add($1);

$$ = memberSet;

}

| enum\_list COMMA enum {

ISet<Pair<Symbol,bool>> memberSet = $1;

memberSet.Add($3);

$$ = memberSet;

};

enum:

NAME {

Symbol symbol = MiddleCodeGenerator.EnumItem($1, null);

$$ = new Pair<Symbol,bool>(symbol, false);

}

| NAME ASSIGN constant\_integral\_expression {

Symbol symbol = MiddleCodeGenerator.EnumItem($1, $3.Symbol);

$$ = new Pair<Symbol,bool>(symbol, true);

};

declarator\_list:

init\_bitfield\_declarator {

$$ = $1;

}

| declarator\_list COMMA init\_bitfield\_declarator {

$1.AddRange($3);

$$ = $1;

};

init\_bitfield\_declarator:

declarator {

MiddleCodeGenerator.Declarator

(SpecifierStack.Peek(), $1);

$$ = new List<MiddleCode>();

}

| declarator ASSIGN initializer {

$$ = MiddleCodeGenerator.AssignmentDeclarator

(SpecifierStack.Peek(), $1, $3);

}

| declarator COLON constant\_integral\_expression {

MiddleCodeGenerator.BitfieldDeclarator

(SpecifierStack.Peek(), $1, $3.Symbol);

$$ = new List<MiddleCode>();

}

| COLON constant\_integral\_expression {

MiddleCodeGenerator.BitfieldDeclarator

(SpecifierStack.Peek(), null, $2.Symbol);

$$ = new List<MiddleCode>();

};

declarator:

optional\_pointer\_list direct\_declarator {

$$ = MiddleCodeGenerator.PointerDeclarator($1, $2);

};

direct\_declarator:

NAME {

$$ = new Declarator($1);

}

| LEFT\_PAREN declarator RIGHT\_PAREN {

$$ = $2;

}

| direct\_declarator LEFT\_SQUARE

optional\_constant\_integral\_expression RIGHT\_SQUARE {

$$ = MiddleCodeGenerator.ArrayType($1, $3);

}

| direct\_declarator LEFT\_PAREN parameter\_ellipse\_list RIGHT\_PAREN{

$$ = MiddleCodeGenerator.

NewFunctionDeclaration($1, $3.First, $3.Second);

}

| direct\_declarator LEFT\_PAREN

optional\_name\_list RIGHT\_PAREN {

$$ = MiddleCodeGenerator.OldFunctionDeclaration($1, $3);

};

optional\_pointer\_list:

/\* Empty \*/ { $$ = new List<CCompiler.Type>(); }

| pointer\_list { $$ = $1; };

pointer\_list:

pointer {

$$ = new List<CCompiler.Type>();

((List<CCompiler.Type>) $$).Add($1);

}

| pointer\_list pointer {

$1.Add($2);

$$ = $1;

};

pointer:

ASTERRISK {

$$ = new CCompiler.Type((CCompiler.Type) null);

}

| ASTERRISK declaration\_specifier\_list {

$$ = (new Specifier($2, true)).Type;

};

optional\_parameter\_ellipse\_list:

/\* Empty \*/ { $$ = null; }

| parameter\_ellipse\_list { $$ = $1; };

parameter\_ellipse\_list:

parameter\_list {

$$ = new Pair<List<Pair<string,Symbol>>,Boolean>($1, false);

}

| parameter\_list COMMA ELLIPSE {

$$ = new Pair<List<Pair<string,Symbol>>,Boolean>($1, true);

};

parameter\_list:

{ ++CallDepth; }

parameter\_declaration {

--CallDepth;

List<Pair<string,Symbol>> pairList =

new List<Pair<string,Symbol>>();

pairList.Add($2);

$$ = pairList;

}

| parameter\_list COMMA {

++CallDepth;

}

parameter\_declaration {

--CallDepth;

$1.Add($4);

$$ = $1;

};

parameter\_declaration:

declaration\_specifier\_list {

$$ = MiddleCodeGenerator.Parameter(new Specifier($1, false), null);

}

| declaration\_specifier\_list\_x declarator {

$$ = MiddleCodeGenerator.Parameter(SpecifierStack.Pop(), $2);

}

| declaration\_specifier\_list\_x abstract\_declarator {

$$ = MiddleCodeGenerator.Parameter(SpecifierStack.Pop(), $2);

};

optional\_name\_list:

/\* Empty \*/ { $$ = new List<string>(); }

| identifier\_list { $$ = $1; };

identifier\_list:

NAME {

List<string> nameList = new List<string>();

nameList.Add($1);

$$ = nameList;

}

| identifier\_list COMMA NAME {

$1.Add($3);

$$ = $1;

};

initializer:

assignment\_expression {

$$ = $1;

}

| LEFT\_BLOCK initializer\_list optional\_comma RIGHT\_BLOCK {

$$ = $2;

};

optional\_comma:

/\* Empty \*/

| COMMA;

initializer\_list:

initializer {

List<object> initializerList = new List<object>();

initializerList.Add($1);

$$ = initializerList;

}

| initializer\_list COMMA initializer {

$1.Add($3);

$$ = $1;

};

type\_name:

declaration\_specifier\_list {

$$ = MiddleCodeGenerator.

TypeName(new Specifier($1, false), null);

}

| declaration\_specifier\_list {

SpecifierStack.Push(new Specifier($1, false));

}

abstract\_declarator {

$$ = MiddleCodeGenerator.

TypeName(SpecifierStack.Pop(), $3);

};

abstract\_declarator:

pointer\_list {

$$ = MiddleCodeGenerator.PointerDeclarator($1, null);

}

| optional\_pointer\_list direct\_abstract\_declarator {

$$ = MiddleCodeGenerator.PointerDeclarator($1, $2);

};

direct\_abstract\_declarator:

LEFT\_PAREN abstract\_declarator RIGHT\_PAREN {

$$ = $2;

}

| LEFT\_SQUARE optional\_constant\_integral\_expression RIGHT\_SQUARE {

$$ = MiddleCodeGenerator.ArrayType(null, $2);

}

| direct\_abstract\_declarator

LEFT\_SQUARE optional\_constant\_integral\_expression RIGHT\_SQUARE {

$$ = MiddleCodeGenerator.ArrayType($1, $3);

}

| LEFT\_PAREN optional\_parameter\_ellipse\_list RIGHT\_PAREN {

$$ = MiddleCodeGenerator.

NewFunctionDeclaration(null, $2.First, $2.Second);

}

| direct\_abstract\_declarator

LEFT\_PAREN optional\_parameter\_ellipse\_list RIGHT\_PAREN {

$$ = MiddleCodeGenerator.

NewFunctionDeclaration($1, $3.First, $3.Second);

};

optional\_statement\_list:

/\* Empty \*/ {

$$ = new Statement(new List<MiddleCode>(),

new HashSet<MiddleCode>());

}

| optional\_statement\_list statement {

MiddleCodeGenerator.Backpatch($1.NextSet(), $2.CodeList());

List<MiddleCode> codeList = new List<MiddleCode>();

codeList.AddRange($1.CodeList());

codeList.AddRange($2.CodeList());

$$ = new Statement(codeList, $2.NextSet());

};

statement:

opened\_statement { $$ = $1; }

| closed\_statement { $$ = $1; };

switch\_header:

/\* Empty. \*/ { MiddleCodeGenerator.SwitchHeader(); };

loop\_header:

/\* Empty. \*/ { MiddleCodeGenerator.LoopHeader(); };

opened\_statement:

IF LEFT\_PAREN expression RIGHT\_PAREN statement {

$$ = MiddleCodeGenerator.IfStatement($3, $5);

}

| IF LEFT\_PAREN expression RIGHT\_PAREN closed\_statement

ELSE opened\_statement {

$$ = MiddleCodeGenerator.IfElseStatement($3, $5, $7);

}

| SWITCH switch\_header LEFT\_PAREN expression RIGHT\_PAREN

opened\_statement {

$$ = MiddleCodeGenerator.SwitchStatement($4, $6);

}

| CASE constant\_integral\_expression COLON opened\_statement {

$$ = MiddleCodeGenerator.CaseStatement($2, $4);

}

| WHILE loop\_header LEFT\_PAREN expression RIGHT\_PAREN

opened\_statement {

$$ = MiddleCodeGenerator.WhileStatement($4, $6);

}

| FOR loop\_header LEFT\_PAREN optional\_expression SEMICOLON

optional\_expression SEMICOLON optional\_expression RIGHT\_PAREN

opened\_statement {

$$ = MiddleCodeGenerator.

ForStatement($4, $6, $8, $10);

}

| NAME COLON opened\_statement {

$$ = MiddleCodeGenerator.LabelStatement($1, $3);

};

closed\_statement:

IF LEFT\_PAREN expression RIGHT\_PAREN closed\_statement

ELSE closed\_statement {

$$ = MiddleCodeGenerator.IfElseStatement($3, $5, $7);

}

| SWITCH switch\_header LEFT\_PAREN expression RIGHT\_PAREN

closed\_statement {

$$ = MiddleCodeGenerator.SwitchStatement($4, $6);

}

| WHILE loop\_header LEFT\_PAREN expression RIGHT\_PAREN

closed\_statement {

$$ = MiddleCodeGenerator.WhileStatement($4, $6);

}

| FOR loop\_header LEFT\_PAREN optional\_expression SEMICOLON

optional\_expression SEMICOLON optional\_expression RIGHT\_PAREN

closed\_statement {

$$ = MiddleCodeGenerator.

ForStatement($4, $6, $8, $10);

}

| DO loop\_header statement WHILE LEFT\_PAREN expression RIGHT\_PAREN

SEMICOLON {

$$ = MiddleCodeGenerator.DoStatement($3, $6);

}

| CASE constant\_integral\_expression COLON closed\_statement {

$$ = MiddleCodeGenerator.CaseStatement($2, $4);

}

| DEFAULT COLON closed\_statement {

$$ = MiddleCodeGenerator.DefaultStatement($3);

}

| CONTINUE SEMICOLON {

$$ = MiddleCodeGenerator.ContinueStatement();

}

| BREAK SEMICOLON {

$$ = MiddleCodeGenerator.BreakStatement();

}

| LEFT\_BLOCK {

SymbolTable.CurrentTable =

new SymbolTable(SymbolTable.CurrentTable, Scope.Block);

}

optional\_statement\_list RIGHT\_BLOCK {

SymbolTable.CurrentTable =

SymbolTable.CurrentTable.ParentTable;

$$ = $3;

}

| GOTO NAME SEMICOLON {

$$ = MiddleCodeGenerator.GotoStatement($2);

}

| RETURN optional\_expression SEMICOLON {

$$ = MiddleCodeGenerator.ReturnStatement($2);

}

| optional\_expression SEMICOLON {

$$ = MiddleCodeGenerator.ExpressionStatement($1);

}

| declaration {

$$ = new Statement($1, new HashSet<MiddleCode>());

}

| JUMP\_REGISTER LEFT\_PAREN REGISTER\_NAME RIGHT\_PAREN SEMICOLON {

$$ = MiddleCodeGenerator.JumpRegisterStatement($3);

}

/\*| REGISTER\_NAME ASSIGN assignment\_expression {

$$ = MiddleCodeGenerator.RegisterExpression($1, $3);

}\*/

| INTERRUPT LEFT\_PAREN constant\_integral\_expression RIGHT\_PAREN

SEMICOLON {

$$ = MiddleCodeGenerator.InterruptStatement($3);

}

| SYSCALL LEFT\_PAREN RIGHT\_PAREN SEMICOLON {

$$ = MiddleCodeGenerator.SyscallStatement();

};

optional\_constant\_integral\_expression:

/\*/ Empty \*/ { $$ = null; }

| constant\_integral\_expression { $$ = $1; };

constant\_integral\_expression:

condition\_expression {

$$ = MiddleCodeGenerator.

ConstantIntegralExpression($1);

};

optional\_expression:

/\* Empty \*/ { $$ = null; }

| expression { $$ = $1; };

expression:

assignment\_expression {

$$ = $1;

}

| expression COMMA assignment\_expression {

$$ = MiddleCodeGenerator.CommaExpression($1, $3);

};

assignment\_expression:

condition\_expression {

$$ = $1;

}

| prefix\_expression ASSIGN assignment\_expression {

$$ = MiddleCodeGenerator.AssignmentExpression($1, $3, true);

}

| prefix\_expression ADD\_ASSIGN assignment\_expression {

Assert.Error($1.Symbol.Assignable, $1.Symbol.Name,

Message.Not\_assignable);

Expression tempExpr =

MiddleCodeGenerator.AdditionExpression($1, $3);

$$ = MiddleCodeGenerator.

AssignmentExpression($1, tempExpr, false);

}

| prefix\_expression SUBTRACT\_ASSIGN assignment\_expression {

Assert.Error($1.Symbol.Assignable, $1.Symbol.Name,

Message.Not\_assignable);

Expression tempExpr =

MiddleCodeGenerator.SubtractionExpression($1, $3);

$$ = MiddleCodeGenerator.

AssignmentExpression($1, tempExpr, false);

}

| prefix\_expression multiply\_assignment\_operator

assignment\_expression {

Assert.Error($1.Symbol.Assignable, $1.Symbol.Name,

Message.Not\_assignable);

Expression tempExpr =

MiddleCodeGenerator.MultiplyExpression($1, $2, $3);

$$ = MiddleCodeGenerator.

AssignmentExpression($1, tempExpr, false);

}

| prefix\_expression bitwise\_assignment\_operator

assignment\_expression {

Assert.Error($1.Symbol.Assignable, $1.Symbol.Name,

Message.Not\_assignable);

Expression tempExpr =

MiddleCodeGenerator.BitwiseExpression($1, $2, $3);

$$ = MiddleCodeGenerator.

AssignmentExpression($1, tempExpr, false);

}

| prefix\_expression shift\_assignment\_operator

assignment\_expression {

Assert.Error($1.Symbol.Assignable, $1.Symbol.Name,

Message.Not\_assignable);

Expression tempExpr =

MiddleCodeGenerator.ShiftExpression($1, $2, $3);

$$ = MiddleCodeGenerator.

AssignmentExpression($1, tempExpr, false);

};

multiply\_assignment\_operator:

MULTIPLY\_ASSIGN { $$ = MiddleOperator.SignedMultiply; }

| DIVIDE\_ASSIGN { $$ = MiddleOperator.SignedDivide; }

| MODULO\_ASSIGN { $$ = MiddleOperator.SignedModulo; };

bitwise\_assignment\_operator:

AND\_ASSIGN { $$ = MiddleOperator.BitwiseAnd; }

| IOR\_ASSIGN { $$ = MiddleOperator.BitwiseIOr; }

| XOR\_ASSIGN { $$ = MiddleOperator.BitwiseXOr; };

shift\_assignment\_operator:

LEFT\_SHIFT\_ASSIGN { $$ = MiddleOperator.ShiftLeft; }

| RIGHT\_SHIFT\_ASSIGN { $$ = MiddleOperator.ShiftRight; };

condition\_expression:

logical\_or\_expression {

$$ = $1;

}

| logical\_or\_expression QUESTION\_MARK expression COLON

condition\_expression {

$$ = MiddleCodeGenerator.

ConditionalExpression($1, $3, $5);

};

logical\_or\_expression:

logical\_and\_expression {

$$ = $1;

}

| logical\_or\_expression LOGICAL\_OR logical\_and\_expression {

$$ = MiddleCodeGenerator.LogicalOrExpression($1, $3);

};

logical\_and\_expression:

bitwise\_ior\_expression {

$$ = $1;

}

| logical\_and\_expression LOGICAL\_AND bitwise\_ior\_expression {

$$ = MiddleCodeGenerator.LogicalAndExpression($1, $3);

};

bitwise\_ior\_expression:

bitwise\_xor\_expression {

$$ = $1;

}

| bitwise\_ior\_expression BITWISE\_IOR bitwise\_xor\_expression {

$$ = MiddleCodeGenerator.BitwiseExpression

($1, MiddleOperator.BitwiseIOr, $3);

};

bitwise\_xor\_expression:

bitwise\_and\_expression {

$$ = $1;

}

| bitwise\_xor\_expression BITWISE\_XOR bitwise\_and\_expression {

$$ = MiddleCodeGenerator.BitwiseExpression

($1, MiddleOperator.BitwiseXOr, $3);

};

bitwise\_and\_expression:

equality\_expression {

$$ = $1;

}

| bitwise\_and\_expression AMPERSAND equality\_expression {

$$ = MiddleCodeGenerator.BitwiseExpression

($1, MiddleOperator.BitwiseAnd, $3);

};

equality\_expression:

relation\_expression {

$$ = $1;

}

| equality\_expression equality\_operator relation\_expression {

$$ = MiddleCodeGenerator.

RelationalExpression($1, $2, $3);

};

equality\_operator:

EQUAL { $$ = MiddleOperator.Equal; }

| NOT\_EQUAL { $$ = MiddleOperator.NotEqual; };

relation\_expression:

shift\_expression {

$$ = $1;

}

| relation\_expression relation\_operator shift\_expression {

$$ = MiddleCodeGenerator.

RelationalExpression($1, $2, $3);

};

relation\_operator:

LESS\_THAN { $$ = MiddleOperator.SignedLessThan; }

| LESS\_THAN\_EQUAL { $$ = MiddleOperator.SignedLessThanEqual; }

| GREATER\_THAN { $$ = MiddleOperator.SignedGreaterThan; }

| GREATER\_THAN\_EQUAL { $$=MiddleOperator.SignedGreaterThanEqual;};

shift\_expression:

add\_expression {

$$ = $1;

}

| shift\_expression shift\_operator add\_expression {

$$ = MiddleCodeGenerator.ShiftExpression($1, $2, $3);

};

shift\_operator:

LEFT\_SHIFT { $$ = MiddleOperator.ShiftLeft; }

| RIGHT\_SHIFT { $$ = MiddleOperator.ShiftRight; };

add\_expression:

multiply\_expression {

$$ = $1;

}

| add\_expression PLUS multiply\_expression {

$$ = MiddleCodeGenerator.AdditionExpression($1, $3);

}

| add\_expression MINUS multiply\_expression {

$$ = MiddleCodeGenerator.

SubtractionExpression($1, $3);

};

multiply\_expression:

cast\_expression {

$$ = $1;

}

| multiply\_expression multiply\_operator cast\_expression {

$$ = MiddleCodeGenerator.

MultiplyExpression($1, $2, $3);

};

multiply\_operator:

ASTERRISK { $$ = MiddleOperator.SignedMultiply; }

| DIVIDE { $$ = MiddleOperator.SignedDivide; }

| MODULO { $$ = MiddleOperator.SignedModulo; };

cast\_expression:

prefix\_expression {

$$ = $1;

}

| LEFT\_PAREN type\_name RIGHT\_PAREN cast\_expression {

$$ = MiddleCodeGenerator.CastExpression($2, $4);

};

prefix\_add\_operator:

PLUS { $$ = MiddleOperator.UnaryAdd; }

| MINUS { $$ = MiddleOperator.UnarySubtract; };

increment\_operator:

INCREMENT { $$ = MiddleOperator.Increment; }

| DECREMENT { $$ = MiddleOperator.Decrement; };

prefix\_expression:

postfix\_expression {

$$ = $1;

}

| prefix\_add\_operator cast\_expression {

$$ = MiddleCodeGenerator.UnaryExpression($1, $2);

}

| increment\_operator prefix\_expression {

$$ = MiddleCodeGenerator.

PrefixIncrementExpression($1, $2);

}

| LOGICAL\_NOT cast\_expression {

$$ = MiddleCodeGenerator.

LogicalNotExpression($2);

}

| BITWISE\_NOT cast\_expression {

$$ = MiddleCodeGenerator.

BitwiseNotExpression($2);

}

| SIZEOF prefix\_expression {

$$ = MiddleCodeGenerator.SizeOfExpression($2);

}

| SIZEOF LEFT\_PAREN type\_name RIGHT\_PAREN {

$$ = MiddleCodeGenerator.SizeOfType($3);

}

| AMPERSAND cast\_expression {

$$ = MiddleCodeGenerator.AddressExpression($2);

}

| ASTERRISK cast\_expression {

$$ = MiddleCodeGenerator.DerefExpression($2);

};

postfix\_expression:

primary\_expression {

$$ = $1;

}

| postfix\_expression increment\_operator {

$$ = MiddleCodeGenerator.

PostfixIncrementExpression($1, $2);

}

| postfix\_expression DOT NAME {

$$ = MiddleCodeGenerator.DotExpression($1, $3);

}

| postfix\_expression ARROW NAME {

$$ = MiddleCodeGenerator.ArrowExpression($1, $3);

}

| postfix\_expression LEFT\_SQUARE expression RIGHT\_SQUARE {

$$ = MiddleCodeGenerator.IndexExpression($1, $3);

}

| postfix\_expression {

MiddleCodeGenerator.CallHeader($1);

}

LEFT\_PAREN optional\_argument\_expression\_list RIGHT\_PAREN {

$$ = MiddleCodeGenerator.CallExpression($1, $4);

};

primary\_expression:

NAME {

$$ = MiddleCodeGenerator.SymbolExpression($1);

}

| VALUE {

$$ = MiddleCodeGenerator.ValueExpression($1);

}

| REGISTER\_NAME {

$$ = MiddleCodeGenerator.InspectRegisterExpression($1);

}

| CARRY\_FLAG {

$$ = MiddleCodeGenerator.CarryFlagExpression();

}

| LEFT\_PAREN expression RIGHT\_PAREN {

$$ = $2;

};

optional\_argument\_expression\_list:

/\* Empty \*/ { $$ = new List<Expression>(); }

| argument\_expression\_list { $$ = $1; };

argument\_expression\_list:

assignment\_expression {

List<Expression> argList = new List<Expression>();

argList.Add(MiddleCodeGenerator.

ParameterExpression($1));

$$ = argList;

}

| argument\_expression\_list COMMA assignment\_expression {

List<Expression> argList = $1;

argList.Add(MiddleCodeGenerator.

ParameterExpression($3));

$$ = argList;

};

%%