# Parsing

The parser can be considered to be the heart of the compiler. It requests token from the scanner, checks that the source code complies with the grammar, builds the symbol table, performs type checking, and generates the middle code.

## The MiddleOperator Enumeration

Before we start the parsing, we need to look into the middle code generated by the parser. The MiddleOperator enumeration holds the operator of the middle code.

MiddleOperator.cs

namespace CCompiler {

public enum MiddleOperator

{Goto, AssignRegister, Compare,

InspectFlagbyte, InspectRegister, SysCall,

JumpRegister, Interrupt, Expression, Declaration,

PushZero, PushOne, PushFloat, PopFloat, TopFloat,

PopEmpty, Comma, AssignInitSize, Assign, LogicalOr, LogicalAnd,

BitwiseOr, BitwiseXOr, BitwiseAnd,

Equal, NotEqual, SignedLessThan,

SignedLessThanEqual, SignedGreaterThan,

SignedGreaterThanEqual, UnsignedLessThan,

UnsignedLessThanEqual, UnsignedGreaterThan,

UnsignedGreaterThanEqual, ShiftLeft,

ShiftRight, BinaryAdd, BinarySubtract,

SignedMultiply, SignedDivide, SignedModulo,

UnsignedMultiply, UnsignedDivide, UnsignedModulo,

UnaryAdd, UnarySubtract, Carry, NotCarry,

LogicalNot, BitwiseNot, Address, Dereference,

Call, PostCall, DecreaseStack,

ParameterInitSize, Parameter, Empty, SetReturnValue,

Return, Exit, Increment, Decrement, PreCall,

FunctionEnd, Conditional, GetReturnValue,

IntegralToIntegral, IntegralToFloating, FloatingToIntegral,

ArrayToPointer, FunctionToPointer, StringToPointer,

ValueOffset, AddressOffset, Case, CaseEnd, StackTop,

Index, Dot, Arrow, Variable, Value, Initializer, InitializerZero};

}

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 Stack<Scope> ScopeStack = new Stack<Scope>();

%}

The parser is made up by tokens and rules, where tokens correspond to operators, keywords, and characters. The following tokens are used by the parser, and they are also returned by the scanner in the Chapter 2.

A variable or function has auto, register, static, extern, or typedef storage.

%token AUTO REGISTER STATIC EXTERN TYPEDEF

A variable can also be qualified as constant or volatile.

CONSTANT VOLATILE

An integral type can be signed or unsigned, and char, short int, int, or long int.

SIGNED UNSIGNED CHAR SHORT INT LONG

A floating type can be float, double, or long double.

FLOAT DOUBLE

Moreover, there are enums, structs, and unions.

ENUM STRUCT UNION

Finally, there is also the void type, that marks the absence of a type. It is used to mark the absence of a parameter list or return type of a function, and as pointer to void.

VOID

The asterisk (’\*’) is used both for multiplication and dereferencing pointers.

PLUS MINUS ASTERRISK DIVIDE MODULO INCREMENT DECREMENT

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

GREATER\_THAN\_EQUAL

The assignment is both simple and compound.

ASSIGN ADD\_ASSIGN SUBTRACT\_ASSIGN MULTIPLY\_ASSIGN

DIVIDE\_ASSIGN MODULO\_ASSIGN AND\_ASSIGN OR\_ASSIGN XOR\_ASSIGN

LEFT\_SHIFT\_ASSIGN RIGHT\_SHIFT\_ASSIGN

LOGICAL\_OR LOGICAL\_AND LOGICAL\_NOT

The ampersand (’&’) is used both bitwise and as well as address operator.

AMPERSAND BITWISE\_XOR BITWISE\_OR BITWISE\_NOT

The ellipse is made up by threee dots (’...’) and is used when defining functions with a variable number of parameters, such as printf or scanf.

QUESTION\_MARK COLON COMMA SEMICOLON ELLIPSE DOT ARROW SIZEOF

In C, we have the regular parentheses ’(’ and ’)’, block parentheses ’{’ and ’}’, and squares ’[’ and ’]’

LEFT\_PARENTHESIS RIGHT\_PARENTHESIS LEFT\_BLOCK RIGHT\_BLOCK

LEFT\_SQUARE RIGHT\_SQUARE

We need a set of keywords for the statements of C.

IF ELSE SWITCH CASE DEFAULT FOR WHILE DO BREAK CONTINUE RETURN GOTO

The interrupt, jump\_register, syscall, and carry flag tokens are used for internal system calls only.

INTERRUPT JUMP\_REGISTER SYSCALL CARRY\_FLAG

Note that we do not need any tokens for comments, since they have already been taken care of by the preprocessor.

The next part of the scanner is the union section, where we define the attributes of the rules below.

%union {

public string name;

The Register enumeration holds the registers of the target machine.

public Register register;

Each symbol has a type, which is described by the Type class.

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 List<Symbol> symbol\_list;

public List<string> string\_list;

public Declarator declarator;

public List<Declarator> declarator\_list;

public MiddleOperator middle\_operator;

public Expression expression;

public List<Expression> expression\_list;

public Statement statement;

public Pair<List<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 declaration

declaration\_list optional\_declaration\_list

initialization\_bitfield\_simple\_declarator

%type <declarator> optional\_simple\_declarator declarator direct\_declarator

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

%type <type> pointer\_marker

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

parameter\_ellipse\_list

%type <symbol\_list> parameter\_list

%type <symbol> 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 <middle\_operator> assignment\_operator equality\_operator

relation\_operator add\_operator shift\_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\_OR\_expression

bitwise\_xor\_expression bitwise\_and\_expression

equality\_expression relation\_expression

shift\_expression add\_expression

multiply\_expression type\_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 source\_code\_file

## Declarations

%%

Finally, we have reached the last (and largest) part of the parser, where the rules are defined. The source\_code\_file rule is the start of the grammar.

source\_code\_file:

external\_declaration

| source\_code\_file external\_declaration;

external\_declaration:

function\_definition

| declaration;

### Function Definition

A function definition is made up by declarator, possible preceded by a declaration specifier list, followed by an optional declaration list, and a block with an optional statement list. For instance, in the function below unsigned long int is the declaration specifier list, square(int value) is the declarator, and return value \* value ; is the only statement in the statement list.

unsigned long int square(int value) { return value \* value; }

In C, there are two ways to define a function, the old way where the parameter list hold holds the names of the parameters and where there types are defined afterwards, and new way where the parameter list holds the names and types of the parameters. In the following examples f is defined by the old way and g is defined by the new way.

double f(a, b)

int a;

double b; {

return a + b;

}

double f(int a, double b) {

return a + b;

}

The first part of the function definition rule handles the case where there is specifier list of at least specifier present. There are three methods to handle the function definition: FunctionHeader that hanles the header of the function, CheckFunctionDefinition that makes sure the function is defined in either the old or new way, and FunctionEnd that generates the assembly code of the function and saves it in the global static set.

MainParser.gppg

function\_definition:

declaration\_specifier\_list\_x declarator {

MiddleCodeGenerator.FunctionHeader

(SpecifierStack.Pop(), $2);

}

optional\_declaration\_list {

MiddleCodeGenerator.CheckFunctionDefinition();

}

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

$8.CodeList.InsertRange(0, $7);

MiddleCodeGenerator.BackpatchGoto();

MiddleCodeGenerator.FunctionEnd($8);

}

The second rule handles the case where there is no specifier list. In that case, the function return type is assumed to be a signed integer.

| declarator {

MiddleCodeGenerator.FunctionHeader(null, $1);

}

optional\_declaration\_list {

MiddleCodeGenerator.CheckFunctionDefinition();

}

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

$7.CodeList.InsertRange(0, $6);

MiddleCodeGenerator.BackpatchGoto();

MiddleCodeGenerator.FunctionEnd($7);

};

optional\_declaration\_list:

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

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

### Specifier List

A declaration is made up by a declaration specifier list, an optional declarator list, and a semicolon. For instance, in the following declaration, static long int is the specifier list while \*p, and a[3] are declarators.

static long int \*p, a[3];

There is possible to define a declaration without a declarator list. This feature is useful when declare a named struct or union:

struct \_s {int i};

However, this feature makes it also possible to declare unnamed specifier lists. The following declaration are syntactically correct but without meaning, they will result in no generated code.

unsigned short int;

union {short s;}

However, the following declaration has meaning since the enumeration values can be used.

enum {ZERO = 0, ONE = 1, TEN = 10};

MainParser.gppg

declaration:

declaration\_specifier\_list SEMICOLON {

SpecifierStack.Push(Specifier.SpecifierList($1));

$$ = new List<MiddleCode>();

}

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

SpecifierStack.Pop();

$$ = $2;

};

optional\_declaration\_specifier\_list\_x:

/\* empty \*/

| declaration\_specifier\_list\_x;

declaration\_specifier\_list\_x:

declaration\_specifier\_list {

SpecifierStack.Push(Specifier.SpecifierList($1));

};

A declaration specifier list is made up by one or more declaration specifiers, which are stored in a list.

declaration\_specifier\_list:

declaration\_specifier {

$$ = new List<object>();

$$.Add($1);

}

| declaration\_specifier declaration\_specifier\_list {

$2.Add($1);

$$ = $2;

};

There is a set of different kind of specifiers. The following specifiers return a value of the Mask enumeration.

* Storage specifiers: auto, register, static, extern, or typedef
* Type qualifiers: constant or volatile
* Type specifiers: void, char, short, int, long, float, double, signed, or unsigned

The following specifiers return a reference to the Type class.

* Struct or union specifier
* Enum specifier
* Typedef name

declaration\_specifier:

AUTO { $$ = Mask.Auto; }

| REGISTER { $$ = Mask.Register; }

| STATIC { $$ = Mask.Static; }

| EXTERN { $$ = Mask.Extern; }

| TYPEDEF { $$ = Mask.Typedef; }

| CONSTANT { $$ = Mask.Constant; }

| VOLATILE { $$ = Mask.Volatile; }

| 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; };

### Structs and Unions

A struct or union specifier holds an optional name and a declaration list within brackets. It can also refer to an declared struct or union by its name. When the declaration list is parsed, the struct or union is given a symbol table of its own and each member becomes added to symbol table.

MainParser.gppg

struct\_or\_union\_specifier:

struct\_or\_union optional\_name {

The call to StructOrUnionHeader adds the struct or union to the symbol table if the optional name is not null.

MiddleCodeGenerator.StructUnionHeader($2, $1);

SymbolTable.CurrentTable =

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

}

The declaration list adds the members of the struct or union to the current symbol table.

LEFT\_BLOCK declaration\_list RIGHT\_BLOCK {

The call to StructOrUnionSpecifier adds the member map of the struct or union to the symbol table if the optional name is not null.

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

SymbolTable.CurrentTable =

SymbolTable.CurrentTable.ParentTable;

}

In case of a struct of union without a declaration list, we look up the name.

| 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 {

$$ = $1;

}

| declaration\_list declaration {

$1.AddRange($2);

$$ = $1;

};

The StructOrUnionHeader adds the struct or union to the tag map if the optional name is not null. We need to do this in order for recursive pointers to work. For instance:

struct Cell {

int value;

struct Cell\* next;

}

If we do not perform this action, there is a risk that the next field in the code above will point to another struct with the same name, defined in a surrounding block or in global space.

### Enumeration

An enumeration declaration may hold a list of enumeration items, which are assigned values. Each item is declared as a constant signed integer, with a value implicitly or explicitly assigned.

MainParser.gppg

enum\_specifier:

ENUM optional\_name {

EnumValueStack.Push(BigInteger.Zero);

}

LEFT\_BLOCK enum\_list RIGHT\_BLOCK {

EnumValueStack.Pop();

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

}

An enumeration declaration may also hold a name without an enumeration item list. In that case, we look up the name and returns its type, which is a constant signed integer. However, the name of the enumeration must exist; otherwise, an error is reported.

| ENUM NAME {

$$ = MiddleCodeGenerator.LookupEnum($2);

};

The result of enum\_list is a set holding the symbols of the enumeration items and a Boolean value indicating whether the item was explicitly assigned. We need the Boolean value in case the enumeration symbol is given extern storage, in which case it is not allowed to assign the enumeration item an explicit value.

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;

};

When it comes to enumerations, we have a potential problem. Each enumeration item is stored in the symbol table as a signed integer with a value. The value may be explicitly assigned or implicitly given a value. If the storage of the enumeration is extern, it is not allowed to assign values to the enumeration items. However, it is in C allowed to state the declaration specifiers in arbitrary order. For instance, the following declaration is valid:

enum {a, b} extern;

The following declaration is invalid, since it has extern storage and a is assigned a value:

enum {a = 1, b} extern;

The problem is that when we parse enumeration declaration we might not yet know if its storage is extern. Therefore, we must also store, for each item, whether it has been explicitly assigned. The handling of the storage is managed by the Specifier class (see Chapter 4) after all declaration specifiers have been parsed. The result is that for each item a pair is returned, holding the symbol of the item and a Boolean value indicating whether the item has been explicitly assigned a value.

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

As mentioned in the previous section, a declaration is made up by a declaration specifier list, an optional declarator list, and a semicolon. In the following declaration, static long int is the specifier list while \*p, and a[3] are declarators.

static long int \*p, a[3];

A declarator list is made up declarators separated by commas. There are three kinds of declarators:

* Simple, such as int i;
* With initialization, such as int i = 3;
* With bitfield, only allowed in structs, such as int i : 3; or int : 3;

Note that the declarator can be omitted in the bitfield case. Also note that it is not possible to combine initialization and bitfield.

MainParser.gppg

declarator\_list:

initialization\_bitfield\_declarator {

$$ = $1;

}

| declarator\_list COMMA initialization\_bitfield\_declarator {

$1.AddRange($3);

$$ = $1;

};

initialization\_bitfield\_declarator:

declarator {

MiddleCodeGenerator.Declarator(SpecifierStack.Peek(), $1);

$$ = new List<MiddleCode>();

}

| declarator ASSIGN initializer {

$$ = MiddleCodeGenerator.InitializedDeclarator

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

}

| optional\_declarator COLON constant\_integral\_expression {

MiddleCodeGenerator.BitfieldDeclarator

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

$$ = new List<MiddleCode>();

};

### Pointer Declarators

A pointer declarator is made up by an optional list of pointer markers and a direct declarator. A pointer marker is made up by an optional qualifier (constant or volatile) list and an asterisk (‘\*’). In other words, each asterisk can be qualified by the word constant and/or volatile, and it is possible to add more than one pointer marker to establish a pointer-to-pointer effect.

MainParser.gppg

declarator:

optional\_pointer\_list direct\_declarator {

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

};

optional\_pointer\_list:

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

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

pointer\_list:

pointer\_marker {

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

$$.Add($1);

}

| pointer\_list pointer\_marker {

$1.Add($2);

$$ = $1;

};

pointer\_marker:

ASTERRISK optional\_qualifier\_list {

$$ = Specifier.QualifierList($2);

};

optional\_qualifier\_list:

/\* Empty \*/ {

$$ = new List<Mask>();

}

| optional\_qualifier\_list qualifier {

$$ = $1;

$$.Add($2);

};

qualifier:

CONSTANT { $$ = Mask.Constant; }

VOLATILE { $$ = Mask.Volatile; };

### Direct Declarator

The next step after we have handled pointer markers is the direct declarator, which may hold an array or function declaration, or just a single name.

direct\_declarator:

NAME {

$$ = new Declarator($1);

}

Another form of direct declarator is another declarator enclosed by parenthesis, in which case we simply return the declarator. The parentheses are present only to change the precedence of the declarator.

| LEFT\_PARENTHESIS declarator RIGHT\_PARENTHESIS {

$$ = $2;

}

A direct declarator can be an array declarator, with a constant expression enclosed by brackets.

| direct\_declarator LEFT\_SQUARE

optional\_constant\_integral\_expression RIGHT\_SQUARE {

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

}

The ArrayType method handles the declaration of an array. Similar to the pointer case above is that it is possible to omit the declarator. It is also possible to omit the expression within the brackets. However, the brackets must always be present in an array declaration.

MainParser.gppg

| direct\_declarator

LEFT\_PARENTHESIS parameter\_ellipse\_list RIGHT\_PARENTHESIS {

$$ = MiddleCodeGenerator.

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

}

The next declarator is the old-style function declarator, where the parameter list is a list of strings.

MainParser.gppg

| direct\_declarator LEFT\_PARENTHESIS

optional\_name\_list RIGHT\_PARENTHESIS {

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

};

optional\_parameter\_ellipse\_list:

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

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

The parameter\_ellipse\_list rule returns a pair of the parameter list and a Boolean value representing the presence of the ellipse marker.

parameter\_ellipse\_list:

parameter\_list {

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

}

| parameter\_list COMMA ELLIPSE {

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

};

For each parameter declaration, we push the scope of the current symbol table on a stack that we pop after the parsing of the declaration. This is because there are special rules for the storage specifiers of parameters (only auto or register allowed) for the Specifier class to handle.

parameter\_list:

{ ScopeStack.Push(SymbolTable.CurrentTable.Scope);

SymbolTable.CurrentTable.Scope = Scope.Parameter;

}

parameter\_declaration {

SymbolTable.CurrentTable.Scope = ScopeStack.Pop();

$$ = new List<Symbol>();

$$.Add($2);

}

| parameter\_list COMMA {

ScopeStack.Push(SymbolTable.CurrentTable.Scope);

SymbolTable.CurrentTable.Scope = Scope.Parameter;

}

parameter\_declaration {

SymbolTable.CurrentTable.Scope = ScopeStack.Pop();

$1.Add($4);

$$ = $1;

};

A parameter declaration can be made up be declaration specifier list and a declarator or an abstract declarator, or only a declaration specifier list, since parameters do not need to be named.

parameter\_declaration:

declaration\_specifier\_list {

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

}

| declaration\_specifier\_list\_x declarator {

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

}

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

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

};

MainParser.gppg

optional\_name\_list:

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

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

name\_list:

NAME {

$$ = new List<string>();

$$.Add($1);

}

| name\_list COMMA NAME {

$1.Add($3);

$$ = $1;

};

### Initialization

The initializer can be either an assignment expression or a block, which is made up by assignment expressions or other blocks. The assignment expression makes sure that the expression cannot hold assignment or comma.

initializer:

assignment\_expression {

$$ = $1;

}

The block holds a list of initializers, which means that initializers can be nested. But in the end, there is always expressions in the initialization lists.

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

$$ = $2;

};

optional\_comma:

/\* Empty \*/

| COMMA;

initializer\_list:

initializer {

$$ = new List<object>();

$$.Add($1);

}

| initializer\_list COMMA initializer {

$1.Add($3);

$$ = $1;

};

### Abstract Declarator

An abstract declarator is a direct declarator without a name, which is allowed in function parameters and as operand of the sizeof operator. It works in the same way as the regular declarator. The only difference is that an abstract declarator cannot be made up by a name.

abstract\_declarator:

pointer\_list {

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

}

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

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

};

direct\_abstract\_declarator:

LEFT\_PARENTHESIS abstract\_declarator RIGHT\_PARENTHESIS {

$$ = $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\_PARENTHESIS optional\_parameter\_ellipse\_list RIGHT\_PARENTHESIS {

$$ = MiddleCodeGenerator.

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

}

| direct\_abstract\_declarator

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

$$ = MiddleCodeGenerator.

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

};

## Statements

The GenerateFunctionCode is called after the middle code of the function has been generated. Its task is to generate the assembly and object code of the function. If the function returns void and is the main function, the Exit middle code is added to middle code list. If the it returns void and is not the main, the Return middle code instruction is added.

Then the middle code is being optimized and a list of base blocks is generated.

The object code is actually generated twice, since we want to know which symbols are not used and can be removed from the symbol table.

If the function is the main function we have to modify the entry point of the function.

Finally, the function is added to the linker map and the symbol table is popped.

The generateAssemblyText method generates the assembly code for the function.

When a function containing goto statements has been parsed, the addresses need to be set with the help of the label map.

### The if-else Problem

The if-else problem is the problem of syntactically interpret the leftmost source code below. Semantically, the middle interpretation of the left statement is the correct one, each else shall be connected to the latest preceding if.

|  |  |  |
| --- | --- | --- |
| if (a < b)  if (c < d)  e = 1;  else  f = 2;  (a) Ambiguous C code | if (a < b) {  if (c < d)  e = 1;  else  f = 2;  }  (b) Correct Interpretation | if (a < b)  if (c < d) {  e = 1;  }  else  f = 2;  © Incorrect interpretation |

Below is a simple set of statement rules. Unfortunately, they are ambiguous in that way that the an else does not have to be connected to the latest preceding if, resulting in both the middle and rightmost semantically interpretation above, depending in which order the rules are applied.

statement ::=

IF LEFT\_PAREN logical\_expression RIGHT\_PAREN statement

| IF LEFT\_PAREN logical\_expression RIGHT\_PAREN statement ELSE statement

| ...

To solve the problem, we need a more complicated set of rules that works with open and closed statements. The following set is unambiguous in that way that it always connects each *else* with the latest preceding *if*, with complies to the semantics of C. A theoretical explanation of the open-closed statement solution is beyond the scope of this book, but I recommend the Dragon Book by Aho et al. for a closer look.

When it comes to the if-else statement we have a problem. Given the following rules:

statement:

**if** ( expression ) statement

| **if** ( expression ) statement **else** statement

| *all other statements*

Let us look at the following two examples:

if (a < b)

if (c < d)

a = 1;

else

b = 1;

if (a < b)

if (c < d)

a = 1;

else

b = 1;

The question is whether the else part matches the first of second if part. The first example suggests that it matches the first part, while the second example suggests that it matches the second part. The first suggestion is the right one, each else shall be matched to the closest if. However, the parser above supports both suggestions. We must rewrite the parser so that it only accepts the first case, which we do by introducing the opened and closed statements:

statement:

opened\_statement

| closed\_statement

opened\_statement:

if ( expression ) statement

| if ( expression ) closed\_statement else opened\_statement

| switch ( expression ) opened\_statement

| case constant\_integral\_expression : opened\_statement

| while ( expression ) opened\_statement

| for ( optional\_expression ; optional\_expression ; optional\_expression )

opened\_statement {

| name : opened\_statement

closed\_statement:

if ( expression ) closed\_statement else closed\_statement

| switch ( expression ) closed\_statement

| while ( expression ) closed\_statement

| for ( optional\_expression ; optional\_expression ; optional\_expression )

closed\_statement

| do statement while ( expression ) ;

| case constant\_integral\_expression : closed\_statement

| default : closed\_statement

| continue ;

| break ;

| { optional\_statement\_list }

| goto name ;

| return optional\_expression ;

| optional\_expression ;

| declaration ;

| jump\_register ( register\_name ) ;

| interrupt ( constant\_integral\_expression ) ;

| syscall ( ) ;

A theoretical explanation of the open-closed statement solution is beyond the scope of this book, but I recommend the Dragon Book by Aho et al. for a closer look.

MainParser.cs

statement:

opened\_statement { $$ = $1; }

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

opened\_statement:

IF LEFT\_PARENTHESIS expression RIGHT\_PARENTHESIS statement {

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

}

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

ELSE opened\_statement {

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

}

| // ...

closed\_statement:

IF LEFT\_PARENTHESIS expression RIGHT\_PARENTHESIS closed\_statement

ELSE closed\_statement {

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

}

| // ...

### The Forward-Jump Problem (Backpatching)

When generating middle code instructions for expressions or statements, a common situation is that we generate forward jump instructions without yet knowing the target address. One way to solve the problem is to work with backpatching, which means that we use sets to keep track of jump instructions with yet unknown target addresses that eventually becomes filled in when the target addresses have become known.

For instance, let us look at the while statement to the left. The parsing of expression a < b will generate the middle code in the middle and the sets to the right. The middle code line will a have undefined targets since we at the moment does not know where to jump if the expression is true or false.

|  |  |  |
| --- | --- | --- |
| while (a < b)  ++a; | 1. if a < b goto ?  2. goto ? | a < b: true set = {1}  false set = {2} |

When parsing the while statement, the sets of the expression becomes backpatched: If the a < b expression is true, line 1 shall jump to line 3, and if it is false, line 2 shall jump to line 5: whatever comes after the while statement.

|  |  |  |
| --- | --- | --- |
| 1. if a < b goto 3  2. goto 5  3. ++a  4. goto 1  5. ... |  |  |

If we instead let the while statement be surrounded by an if-else statement, we have both the true and false sets of the if expression x < y and the sets of the while expression a < b:

|  |  |  |
| --- | --- | --- |
| if (x < y)  while (a < b)  ++a;  else  ++b; | 1. if x < y goto ?  2. goto ?  3. if a < b goto ?  4. goto ?  5. ++a  6. goto 3  7. goto 9  8. ++b  9. ... | x < y: true set: {1}  false set: {2}  a < b: true set: {3}  false set: {4} |

When the if and while statements become parsed the following call will occur:

backpatch({1}, 3);

backpatch({2}, 8);

backpatch({3}, 5);

backpatch({4}, 7);

|  |  |  |
| --- | --- | --- |
| 1. if x < y goto 3  2. goto 8  3. if a < b goto 5  4. goto 7  5. ++a  6. goto 3  7. goto 9  8. ++b  9. ... |  |  |

The generated middle code above may appear ineffective, but the middle code optimizer of Chapter 0 will change it into:

|  |  |  |
| --- | --- | --- |
| 1. if x >= y goto 5  2. if a >= b goto 6  3. ++a  4. goto 2  5. ++b  6. ... |  |  |

Another example is the evaluation of a logical-and expression. Since C takes advantage of lazy evaluation, we have to insert jump instructions in the middle code.

|  |  |  |
| --- | --- | --- |
| if ((a < b) && (c < d))  ++a;  else  ++b; | 1. if a < b goto ?  2. goto ?  3. if c < d goto ?  4. goto ?  5. ++a  6. goto 7  7. ++b  8. ... |  |

When the first expression (a < b) is evaluated, its true set is {1} and its false set is {2}. When the second expression is evaluated the true set of the first expression is backpatched to the beginning of the second expression. This means that if the first expression is true we continue to evaluate the second expression. However, if the first expression is false the second expression will never be evaluated since it not necessary, the total expression will be false regardless of the value of the second expression. The result will be that the true set of the total expression is the one of the second expression and the false set is the union of the falses sets of both expressions.

|  |  |  |
| --- | --- | --- |
| 1. if a < b goto 3  2. goto ?  3. if c < d goto ?  4. goto ?  5. ++a  6. goto 7  7. ++b | true set: {3}  false set: {2, 4} |  |

In a similar way, the true set of an expression with logical-or will be the union of both true sets while the false set will the false set of the second expression.

|  |  |  |
| --- | --- | --- |
| if ((a < b) || (c < d))  ++a;  else  ++b; | 1. if a < b goto ?  2. goto 3  3. if c < d goto ?  4. goto ?  5. ++a  6. goto 7  7. ++b  8. ... | true set: {1, 3}  false set: {4} |

The Backpatch class holds a set of static methods that backpatch a single address or a set of addresses. The first two methods call the last two methods with the current size of ther middle code list as target address. In this way, the target address becomes the next middle code instruction to become generated. More specifically, the backpatching is performed by setting the third address of the middle code, since the third address is always the jump address in conditional and unconditional jump instructions.

### The Statement Class

First, let us look at the Statement class. A statement holds a list of middle code instructions, and the next set, which is a set om jump instructions that shall be backpatched to jump to the next instruction after the statement.

Statement.cs

using System.Collections.Generic;

namespace CCompiler {

public class Statement {

private List<MiddleCode> m\_codeList;

private ISet<MiddleCode> m\_nextSet;

public Statement(List<MiddleCode> codeList,

ISet<MiddleCode> nextSet = null) {

Assert.ErrorXXX(codeList != null);

m\_codeList = codeList;

m\_nextSet = (nextSet != null) ? nextSet : (new HashSet<MiddleCode>());

}

public List<MiddleCode> CodeList {

get { return m\_codeList; }

}

public ISet<MiddleCode> NextSet {

get { return m\_nextSet; }

}

}

}

Then we look at the Expression class. It handles a symbol, a register, and the short list and the long list. The short list and long list both hold the middle code of the expression. The difference is that the short list holds only the side effects of the expression, while the long holds the whole epression. For instance, in the following expression the long list holds the the function call, decrement, and addition. The short list holds the only the side effects: the function call and decrement, but not the addition.

f(x) + a--;

The following statement is correct, but in holds no side-effects, its short list will be empty, and it will in the end not generate and code.

a + (b \* c);

The register parameter is used in system calls when we need to access or assign a specific register.

### Header Rules

The open statements include the compound statements if, if-else, switch, case, while, for, and label statements. The switch, while, and for statements need to do some preparation before the parsing, which means that they need one header rule each. Each rule calls its corresponding method in the Generate class that does the actual work of type checking and middle code generation.

### The Switch Statement

The GenerateSwitchStatement method generates middle code jump instructions for each of the case statements, which are stored in the Main.CaseMapStack stack. Each entry in the stack holds an integer value (each case expression value must be possible to evaluate in compile-time) and a line number. If the original switch expression equals the constant case expression, we jump to the start line of the statement following the case expression.

If there is a default statements, we finally jump to its start line if the switch statement does not match any of the case statements. Note that the default statement does not have to be placed after the case statements, even though I can see no good reason not to do so.

MainParser.gppg

switch\_header:

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

opened\_statement:

// ...

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

opened\_statement {

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

}

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

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

}

// ...

closed\_statement:

// ...

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

closed\_statement {

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

}

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

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

}

| DEFAULT COLON closed\_statement {

$$ = MiddleCodeGenerator.DefaultStatement($3);

}

// ...

### The Case Statement

MainParser.gppg

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

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

}

### The Default Statement

The default statement have to comply with the following demands:

* Main.m\_defaultStack must not be empty. If it is empty, the default statements misses a surrounding switch statements.
* Since Main.m\_defaultStack is not empty, there is at least one surrounding switch statement. We call pop to see if the top value is minus one. If it is not, there has already been a default statement in the closest surrounding switch statement. If it is minus one, there has not been a earlier default statement and we push the line number at the beginning of the statement following the default statement. Since the line numbers are numbered from zero, it cannot be minus one.

MainParser.gppg

| DEFAULT COLON closed\_statement {

$$ = MiddleCodeGenerator.DefaultStatement($3);

}

### The While Statement

The while header prepares the while statement by pushing break and continue stacks with integers sets. The continue set will eventually be backpatched to the beginning of the while statement and break set will be backpatched to the statement following the while statement.

The true set of the while expression is backpatched to the beginning of the statement (or the first statement of a block statement) inside the while loop while the false set is backpatched to the statement following the while statement, similar to the break set. In order to make sure that the while statement is followed by jump statement, the jump\_marker rule adds a jump statement at the end of the statement surrounded by the while statement, which is backpatched to the beginning of the while expression, similar to the continue set.

MainParser.gppg

loop\_header:

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

opened\_statement:

// ...

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

opened\_statement {

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

}

// ...

closed\_statement:

// ...

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

closed\_statement {

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

}

// ...

### The Do Statement

The do statement is a weaker version of the while statement. The difference is that the while expression may be false from the beginning resulting in zero iterations while the do expression is located at the end, resulting in at least one iteration. However, the do statement may in some cases present a simpler and more intuitive solution.

Like the while and for cases, the do statement is prepared by pushing the continue and break stacks. The true set of the do expression is backpatched to the beginning of the statement surrounded by the do statement, the false set is backpatched to the statement followed the do statement and finally the continue and break stacks are popped.

MainParser.cs

closed\_statement:

// ...

| DO loop\_header statement WHILE

LEFT\_PARENTHESIS expression RIGHT\_PARENTHESIS SEMICOLON {

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

}

// ...

### The For Statement

The for statement holds three optional expression: the initialization expression, the test expression, and the increment expression. The true and false sets of the initialization and increment expressions are all backpatched to beginning of the test expression. Like the while case, in order to make sure that the for statement is followed by jump statement the jump\_marker rule add a jump statement at the end of the statement surrounded by the for statement, which is also backpatched to the beginning of the test expression. There is also a jump line inserted after the test optional expression, which is backpatched to the beginning of the for statements to make sure that the for loop works properly even if the test expression has been omitted. An omitted test expression is equivalent to an infinitive loop.

Finally, like the while case, the continue set is backpatched to the beginning of the test expression and the break set is backpatched to the beginning of the statement following the while statement.

The for statement is more complicated than the do and while statements. It holds three optional expression: the initialization expression, the test expression, and the increment expression. The true and false sets of the initialization and increment expressions are all backpatched to beginning of the test expression. An omitted test expression is equivalent to an infinitializere loop.

opened\_statement:

// ...

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

optional\_expression SEMICOLON optional\_expression RIGHT\_PARENTHESIS

opened\_statement {

$$ = MiddleCodeGenerator.ForStatement($4, $6, $8, $10);

}

// ...

closed\_statement:

// ...

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

optional\_expression SEMICOLON optional\_expression RIGHT\_PARENTHESIS

closed\_statement {

$$ = MiddleCodeGenerator.ForStatement($4, $6, $8, $10);

}

// ...

### Label and Goto Statement

The label statement is quite simple, we just add the name of the label together with the line number of the beginning of the statement following the label to Main.LabelMap and check that the label has not been added already. The labels are used as targets of the goto statements. But, as we all know, goto has no place in well-structured programs. Labels and goto are included in C of historical reasons only. More recent languages have omitted goto.

MainParser.cs

opened\_statement:

NAME COLON opened\_statement {

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

};

closed\_statement:

GOTO NAME SEMICOLON {

$$ = MiddleCodeGenerator.GotoStatement($2);

}

To keep track of the label statements we have the label map m\_labelMap with the label name as key and the first middle code instruction of the following statement as value.

### Return Statement

The return statement may have an optional expression.

MainParser.cs

closed\_statement:

RETURN optional\_expression SEMICOLON {

$$ = MiddleCodeGenerator.ReturnStatement($2);

}

### Optional Expression Statement

A statement can also be made up by an optional expression; that is, the statement is an expression followed by a semicolon, or simply a semicolon.

An expression statement is an optional expression followed by a semicolon. The expression is evaluated and its true and false sets are backpatched to the beginning of the next statement. Note that we do not use the result of the expression, the only interested part is its potential side effects. If it has no side effects, the statements will be removed by the middle code optimizer in Chapter 0.

MainParser.gppg

closed\_statement:

optional\_expression SEMICOLON {

$$ = MiddleCodeGenerator.ExpressionStatement($1);

}

### Block Statement

A statement can be an optional sequence of statements enclosed in brackets. The sequence is parsed with a new symbol table.

The block statement simple, a block is just a list of statements surrounded by brackets. Since a two variables with the same name can be defined in different blocks, we push the symbol table before the statement list.

MainParser.cs

closed\_statement:

LEFT\_BLOCK {

SymbolTable.CurrentTable =

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

}

optional\_declaration\_list optional\_statement\_list RIGHT\_BLOCK {

SymbolTable.CurrentTable =

SymbolTable.CurrentTable.ParentTable;

$4.CodeList.InsertRange(0, $3);

$$ = $4;

}

In case of an empty statement list, we return a statement with an empty code list and an empty next set.

optional\_statement\_list:

/\* Empty \*/ {

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

new HashSet<MiddleCode>());

}

In case of an non-empty statement list, we add the code list of the statements of the statement list. For each statement in the list, except the last statement, we backpatch the next set to the beginning of the code list of the next set. The result is a statement with the total middle code list and the next set of the last statement.

| 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);

};

### Jump Register Statements

When performing a call to a function which address is stored in a pointer variables we need to store the address in a register and jump to that register.

MainParser.gppg

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

$$ = MiddleCodeGenerator.JumpRegisterStatement($3);

}

### Interrupt Statements

When making system calls, an interrupt occurs. The operand is an integral value of short size (1 byte).

MainParser.gppg

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

SEMICOLON {

$$ = MiddleCodeGenerator.InterruptStatement($3);

}

### System Call Statements

System calls for Linux.

MainParser.gppg

| SYSCALL LEFT\_PARENTHESIS RIGHT\_PARENTHESIS SEMICOLON {

$$ = MiddleCodeGenerator.SyscallStatement();

};

## Expressions

The third part of the parser is the expressions. We start with the expression of lowest precedence and add a new rule for each new level of precedence.

### The Expression Class

Similar to the statement case above, we have a class to handle expressions.

Expression.cs

using System.Numerics;

using System.Collections.Generic;

namespace CCompiler {

public class Expression {

private Symbol m\_symbol;

private List<MiddleCode> m\_shortList;

private List<MiddleCode> m\_longList;

private Register? m\_register;

public Expression(Symbol symbol, List<MiddleCode> shortList,

List<MiddleCode> longList, Register? register = null) {

m\_symbol = symbol;

m\_shortList = (shortList != null) ? shortList : (new List<MiddleCode>());

m\_longList = (longList != null) ? longList : (new List<MiddleCode>());

m\_register = register;

}

public Symbol Symbol {

get { return m\_symbol; }

}

public List<MiddleCode> ShortList {

get { return m\_shortList; }

}

public List<MiddleCode> LongList {

get { return m\_longList; }

}

public Register? Register {

get { return m\_register; }

}

public override string ToString() {

return m\_symbol.ToString();

}

}

}

### The Comma Expression

MainParser.cs

optional\_expression:

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

| expression { $$ = $1; };

The operator or lowest precedence is the comma operator. A comma expression may be an assignment expression, or two expressions separated by a comma.

expression:

assignment\_expression {

$$ = $1;

}

| expression COMMA assignment\_expression {

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

};

### The Assignment Expression

There are two kinds of assignment: simple and compound. In the case of compound assignment, we first evaluate the corresponding binary expression (addition in the example below), assign its result to a temporary variable that we then assign to original left operand.

|  |  |  |
| --- | --- | --- |
| x += y; | $1 = x + y  x = $1 |  |

An assignment expression can be a conditional expression, a simple assignment, or a compound assignment.

MainParser.cs

assignment\_expression:

condition\_expression {

$$ = $1;

}

| prefix\_expression assignment\_operator assignment\_expression {

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

};

assignment\_operator:

ASSIGN { $$ = MiddleOperator.Assign; }

| ADD\_ASSIGN { $$ = MiddleOperator.BinaryAdd; }

| SUBTRACT\_ASSIGN { $$ = MiddleOperator.BinarySubtract; }

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

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

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

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

| OR\_ASSIGN { $$ = MiddleOperator.BitwiseOr; }

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

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

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

### The Condition Expression

The conditional operator applies lazy evaluation, which means that only one of the true or false expression will be evaluated.

MainParser.gppg

condition\_expression:

logical\_or\_expression {

$$ = $1;

}

| logical\_or\_expression QUESTION\_MARK expression COLON condition\_expression{

$$ = MiddleCodeGenerator.ConditionalExpression($1, $3, $5);

};

### Constant Expression

On several occasions, such array limits and enumeration values, we need to parse constant integral expressions. The expression is a conditional expression, since it cannot hold commas or assignments.

MainParser.cs

optional\_constant\_integral\_expression:

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

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

constant\_integral\_expression:

condition\_expression {

$$ = MiddleCodeGenerator.ConstantIntegralExpression($1);

};

### Logical Expressions

The logical-and expression backpatch the true set of the first operand to the beginning of the right operand. This can be interpreted as if the left expression is true, we try the right expression instead. If both the expressions are true, the result is true. However, if the left expression is false, the result is determined to be false without evaluating the right expression.

The logical-and expression backpatch the true set of the first operand to the beginning of the right operand. This can be interpreted as if the left expression is true, we try the right expression instead. If both the expressions are true, the result is true. However, if the left expression is false, the result is determined to be false without evaluating the right expression.

MainParser.cs

logical\_or\_expression:

logical\_and\_expression {

$$ = $1;

}

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

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

};

logical\_and\_expression:

bitwise\_or\_expression {

$$ = $1;

}

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

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

};

### Bitwise Expressions

Unlike the logical expressions in the previous section, the inclucise or, exclusive or, and and bitwise expression does not support lazy evaluation, which means that the right expressions is always evaluated regardless of the value of the left expression. Both the operands have to be integral and the result is integral.

The are three bitwise operators: or (inclusive or), xor (exclusive or), and and. They take two integer values and perform operations on each bit in the values:

* or: one if at least one value is one, zero otherwise
* xor: one if at exact one value is one, zero otherwise
* and: one if at both values are one, zero otherwise

The expressions have one rule each since they have different precedence, but they call the same method: BitwiseExpression in MiddleCodeGenerator.

MainParser.gppg

bitwise\_or\_expression:

bitwise\_xor\_expression {

$$ = $1;

}

| bitwise\_or\_expression BITWISE\_OR bitwise\_xor\_expression {

$$ = MiddleCodeGenerator.BitwiseExpression

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

};

bitwise\_xor\_expression:

bitwise\_and\_expression {

$$ = $1;

}

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

$$ = MiddleCodeGenerator.BitwiseExpression

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

};

bitwise\_and\_expression:

equality\_expression {

$$ = $1;

}

| bitwise\_and\_expression AMPERSAND equality\_expression {

$$ = MiddleCodeGenerator.BitwiseExpression

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

};

### Shift Expression

The left and right expression of a shift expression must hold integral types. The right expression is type cast to a one-byte integer value.

MainParser.gppg

shift\_expression:

add\_expression {

$$ = $1;

}

| shift\_expression shift\_operator add\_expression {

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

};

shift\_operator:

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

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

### Equality and Relation Expressions

Values of all types except structs or unions can be compared with equality and inequality operator. However, arrays and functions are converted to pointers before the comparation. The equality operators have higher precedence than the relation operators, but they both call RelationalExpression in MiddleCodeGenerator.

MainParser.gppg

equality\_expression:

relation\_expression {

$$ = $1;

}

| equality\_expression equality\_operator relation\_expression {

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

};

equality\_operator:

EQUAL { $$ = MiddleOperator.Equal; }

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

relation\_expression:

shift\_expression {

$$ = $1;

}

| relation\_expression relation\_operator shift\_expression {

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

};

relation\_operator:

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

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

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

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

### Addition and Subtraction Expression

The addition and subtraction expression are a bit more complicated, since we need to take pointer arithmetic into consideration.

MainParser.gppg

add\_expression:

multiply\_expression {

$$ = $1;

}

| add\_expression addition\_operator multiply\_expression {

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

};

add\_operator:

PLUS { $$ = MiddleOperator.BinaryAdd; }

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

### Multiplication Expressions

MainParser.gppg

multiply\_expression:

type\_cast\_expression {

$$ = $1;

}

| multiply\_expression multiply\_operator type\_cast\_expression {

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

};

The multiplication operators are multiply, division, and modulo.

multiply\_operator:

ASTERRISK { $$ = MiddleOperator.SignedMultiply; }

| DIVIDE { $$ = MiddleOperator.SignedDivide; }

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

### Cast Expressions

A type cast expression is a type name within parentheses followed by a type cast expression.

MainParser.gppg

type\_cast\_expression:

prefix\_expression {

$$ = $1;

}

| LEFT\_PARENTHESIS type\_name RIGHT\_PARENTHESIS type\_cast\_expression {

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

};

The type name is a declaration specifier list with or without an abstract declarator.

type\_name:

declaration\_specifier\_list {

$$ = MiddleCodeGenerator.TypeName(Specifier.SpecifierList($1), null);

}

| declaration\_specifier\_list {

SpecifierStack.Push(Specifier.SpecifierList($1));

}

abstract\_declarator {

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

};

### Prefix Expression

In C, there are several prefix expressions:

* Unary add and minus
* Logical not
* Bitwise not
* The sizeof operator
* The address operator
* The dereference operator

### Unary Addition Expressions

MainParser.gppg

prefix\_expression:

postfix\_expression {

$$ = $1;

}

| prefix\_add\_operator type\_cast\_expression {

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

};

prefix\_add\_operator:

PLUS { $$ = MiddleOperator.UnaryAdd; }

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

### Logical Not Expression

The logical not operator just swaps the true and false sets of its operand.

MainParser.gppg

prefix\_expression:

LOGICAL\_NOT type\_cast\_expression {

$$ = MiddleCodeGenerator.LogicalNotExpression($2);

};

### Bitwise Not Expression

The bitwise not operator is quite simple since it only accepts integral values.

MainParser.gppg

prefix\_expression:

BITWISE\_NOT type\_cast\_expression {

$$ = MiddleCodeGenerator.BitwiseNotExpression($2);

};

### The sizeof Expression

The sizeof-operator takes an expression or a type inside parenthesis as operand. The operator is not allowed on function or bitfields.

MainParser.gppg

prefix\_expression:

SIZEOF prefix\_expression {

$$ = MiddleCodeGenerator.SizeOfExpression($2);

};

The sizeof operator can also be applied to a type name within parentheses.

MainParser.gppg

prefix\_expression:

SIZEOF LEFT\_PARENTHESIS type\_name RIGHT\_PARENTHESIS {

$$ = MiddleCodeGenerator.SizeOfType($3);

};

### Address Expression

The address operator takes the address of its operand and the return type is a pointer to the operand type, unless it is an array, in which case the result is a pointer to the array type. The operand cannot have register storage.

MainParser.gppg

prefix\_expression:

AMPERSAND type\_cast\_expression {

$$ = MiddleCodeGenerator.AddressExpression($2);

};

The address operator does not apply to bitfields symbols with register storage. A symbol is addressable if it does not have register storage and is not a bitfield in a struct or union.

### Dereference Expression

The dereference operator takes a pointer or array as operand.

MainParser.gppg

prefix\_expression:

ASTERRISK type\_cast\_expression {

$$ = MiddleCodeGenerator.DereferenceExpression($2);

};

### Prefix Increment and Decrement Expression

The postfix increment and decrement change the operand and return the original value. The operand has to be a left-value of arithmetic or pointer type.

The increment and decrement operators come in two forms: prefix and postfix. The difference is that in the prefix case the resulting value is value after the operator has been applied, while in the postfix case the resulting value is the original value before the operator has been applied. However, the side effects of the operators are the same: in the increment case the value is added by one, and in the decrement case the value is subtracted by one.

MainParser.gppg

prefix\_expression:

increment\_operator prefix\_expression {

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

};

increment\_operator:

INCREMENT { $$ = MiddleOperator.Increment; }

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

### Postfix Increment and Decrement Expression

MainParser.gppg

postfix\_expression:

primary\_expression {

$$ = $1;

}

| postfix\_expression increment\_operator {

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

};

### Arrow Expression

The arrow operator takes a pointer to a struct or union as operand. The result symbol has the operand as address symbol and the member offset as offset.

MainParser.gppg

postfix\_expression:

postfix\_expression ARROW NAME {

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

};

### Index Expression

The index operator takes a pointer or array and an integral. We have three cases:

1. The index is constant, in which case we treat the operator as if it was the arrow operator.
2. The index is not constant and the array or pointer type size is one, in which case we generate that add the index to the pointer or array.
3. The index is not constant and the array or pointer type size is greater than, in which case we also need to generate code that multiply the index with the type size.

MainParser.gppg

postfix\_expression:

postfix\_expression LEFT\_SQUARE expression RIGHT\_SQUARE {

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

};

### Dot Expression

The dot operator generates different result is the operand has and address symbol. If it has, it is the result of the arrow, dereference, or index operator. In that case, it inherits the address symbol and increase the address offset with the member offset.

If the operand does not have an address symbol, it is a regular variable. Then we just create a temporary symbol with the same storage and increase the offset with the member offset.

MainParser.gppg

postfix\_expression:

postfix\_expression DOT NAME {

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

};

### Function Call Expression

When calling a function, first we need to check the number of actual and formal parameters, which shall be equal. Unless the function is elliptic, in which case the actual parameter list can be longer.

Each regular actual parameter is transformed to the type of the formal parameter, and each extra actual parameter is subjected to the type promotion described in Section **Fel! Hittar inte referenskälla.**. Unless the return type is void, we place the result value in a temporary symbol.

Each actual parameter is checked. For a regular parameter (paramType is not null), a function is converted to a pointer to a function, otherwise implicit type conversion occurs. For an extra parameter, a function, array, or string is converted to a pointer, otherwise argument promotion occurs.

For an extra parameter: a char is converted to an integer and a float is converted to a double.

MainParser.gppg

postfix\_expression:

postfix\_expression {

MiddleCodeGenerator.FunctionPreCall($1);

}

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

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

};

### Argument Expression List

Each parameter is converted from a possible logical type to an integer by calling Generate.generateIntegerExpression.

MainParser.gppg

argument\_expression\_list:

assignment\_expression {

$$ = new List<Expression>();

$$.Add(MiddleCodeGenerator.ArgumentExpression(0, $1));

}

| argument\_expression\_list COMMA assignment\_expression {

$1.Add(MiddleCodeGenerator.ArgumentExpression($1.Count, $3));

$$ = $1;

};

### Primary Expressions

We have finally reached the last kind of expression. A primary expression may be another expression within parentheses, a value, a symbol, a register, or the carry flag.

A primary expression is an identifier, which symbol we look up in the symbol table, a value, which we store in the global space, or an expression surrounded by parenthesis.

MainParser.gppg

primary\_expression:

LEFT\_PARENTHESIS expression RIGHT\_PARENTHESIS {

$$ = $2;

}

VALUE {

$$ = MiddleCodeGenerator.ValueExpression($1);

};

primary\_expression:

NAME {

$$ = MiddleCodeGenerator.NameExpression($1);

};

MainParser.gppg

primary\_expression:

REGISTER\_NAME {

$$ = MiddleCodeGenerator.RegisterExpression($1);

};

The carry flag is also only used internally in conjunction with system calls. On some occasion the interrupt call returns information stored in the carry flag.

MainParser.gppg

primary\_expression:

CARRY\_FLAG {

$$ = MiddleCodeGenerator.CarryFlagExpression();

};

MainParser.gppg

primary\_expression:

STACK\_TOP {

$$ = MiddleCodeGenerator.StackTopExpression();

};