# Middle Code Generation

The rules of the parser call corresponding methods of the MiddleCodeGenerator class to perform type checking, build the symbol table, and generate the middle code.

MiddleCodeGenerator.cs

using System;

using System.IO;

using System.Linq;

using System.Numerics;

using System.Collections.Generic;

The AddMiddleCode method adds a MiddleCode object to the given code list.

namespace CCompiler {

public class MiddleCodeGenerator {

public static MiddleCode AddMiddleCode(List<MiddleCode> codeList,

MiddleOperator op, object operand0 = null,

object operand1 = null, object operand2 = null) {

MiddleCode middleCode = new MiddleCode(op, operand0, operand1, operand2);

codeList.Add(middleCode);

return middleCode;

}

### Backpatching

When generating middle code, there are numerous occasions that we generate jump instructions without knowing where to jump. In those cases, we store the instructions in sets and later go back and fill in the jump address. The process is called backpatching.

The first Backpatch methods take set of jump instructions and a list of instructions, where its first instruction is the target of the jump instructions. If the code list is empty, we add an empty instruction in order to make sure there is always a jump target. Then we call the second Backpatch method with the set of jump instruction and the target instruction.

MiddleCodeGenerator.cs

public static void Backpatch(ISet<MiddleCode> sourceSet,

List<MiddleCode> list) {

if (list.Count == 0) {

AddMiddleCode(list, MiddleOperator.Empty);

}

Backpatch(sourceSet, list[0]);

}

The second Backpatch method iterates through the set of jump instructions and set its first operand (index 0) to the target. The first operand of each jump instruction is its target address.

public static void Backpatch(ISet<MiddleCode> sourceSet,

MiddleCode target) {

foreach (MiddleCode source in sourceSet) {

source[0] = target;

}

}

### Function Definition

The GenerateFunctionHeader is called before the middle code of the function has been generated. Its task is to store the symbol of the function in the symbol table, and to create a new symbol table for the symbols of the function.

MiddleCodeGenerator.cs

public static void FunctionHeader(Specifier specifier,

Declarator declarator) {

Storage? storage;

Type returnType;

If specifier is not null, we extract the storage and return type of the function from the specifier.

if (specifier != null) {

storage = specifier.Storage;

returnType = specifier.Type;

}

If specifier is null, there was no storage specifier or return type defined, and we set the storage to extern and the return type to signed int.

else {

storage = Storage.Extern;

returnType = Type.SignedIntegerType;

}

We add the return type to the pointer declarator. In this way the type become complete: a function with a return type.

declarator.Add(returnType);

Then we perform some error checking. The function may lack a name in a function declaration. However, in a function definition it must have a name.

Assert.Error(declarator.Name != null,

Message.Unnamed\_function\_definition);

Technically, the declaration may have any type, and we must check that it really is a function declaration.

Assert.Error(declarator.Type.IsFunction(), declarator.Name,

Message.Not\_a\_function);

The public and static member variable CurrentFunction is a reference to the symbol holding the current function. This reference is used on every occasion during the parsing of the function body when we add or look up a symbol.

SymbolTable.CurrentFunction =

new Symbol(declarator.Name, storage, declarator.Type);

A function definition must be static or extern, it cannot be auto, register, or typedef.

Assert.Error(SymbolTable.CurrentFunction.IsStaticOrExtern(),

declarator.Name,

Message.A\_function\_must\_be\_static\_or\_extern);

Every function definition, as well as static variables, are added to the global static set. They will in the end be translated into assembly code.

SymbolTable.CurrentFunction.FunctionDefinition = true;

SymbolTable.CurrentTable.AddSymbol(SymbolTable.CurrentFunction);

If this function is the main function of the C code, the return type must be void or (signed or unsigned) integer.

if (SymbolTable.CurrentFunction.UniqueName.Equals("main")) {

Assert.Error(returnType.IsVoid() || returnType.IsInteger(), "main",

Message.Function\_main\_must\_return\_void\_or\_integer);

}

Finally, we create a new symbol for the function. Every symbol defined in this function will be stored in the new symbol table, or in one of its sub tables.

SymbolTable.CurrentTable =

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

}

The CheckFunctionDefinition method checks that the definition is correct, especially in case of old-style definition. In the old-style function definition, the parameter list must match the declaration list. In the new-style function definition, the declaration list must be empty since it is not allowed to mix the old and new style.

public static void CheckFunctionDefinition() {

Type funcType = SymbolTable.CurrentFunction.Type;

if (funcType.Style == Type.FunctionStyle.Old) {

List<string> nameList = funcType.NameList;

IDictionary<string,Symbol> entryMap =

SymbolTable.CurrentTable.EntryMap;

In the old-style case, we must make sure the number of parameters equals the number of declarations.

Assert.Error(nameList.Count == entryMap.Count,

SymbolTable.CurrentFunction.Name, Message.

Unmatched\_number\_of\_parameters\_in\_old\_\_style\_function\_definition);

Then we iterate through the parameter list to make sure they are all declared, and to assigned them the correct offset. When the parameters were declared they were given offsets. However, since the parameters and declaration may come in different order, we need to reassign the declarations’ offsets.

int offset = SymbolTable.FunctionHeaderSize;

foreach (string name in nameList) {

Symbol symbol;

If one of the parameters has not been declared, we report an error.

if (!entryMap.TryGetValue(name, out symbol)) {

Assert.Error(name, Message.

Undefined\_parameter\_in\_old\_\_style\_function\_definition);

}

symbol.Offset = offset;

offset += symbol.Type.Size();

}

}

In case of a new-style function definition, we need to make sure the entry map of the current function is empty. If it is not empty, we have a mix of old-style and new-style function definition, which is not allowed.

else {

Assert.Error(SymbolTable.CurrentTable.EntryMap.Count == 0,

Message.New\_and\_old\_style\_mixed\_function\_definition);

Then we iterate through the parameters list and add each parameter to the current symbol table of the function. In this way, each parameter is located at a memory offset so they can be assigned in function calls and be treated like regular variables inside the function. The AddSymbol method will also report an error if two parameters have the same name.

foreach (Symbol symbol in funcType.ParameterList) {

SymbolTable.CurrentTable.AddSymbol(symbol);

}

}

When the parameter list has been settled, we check if the function is the main function of the C code. In that case, some special rules apply.

if (SymbolTable.CurrentFunction.UniqueName.Equals("main")) {

First, we call the InitializationCodeList method that adds a few lines of code that initializes the system and will be placed before the code of the main function.

AssemblyCodeGenerator.InitializationCodeList();

We extract the parameter type list from the current function. In case of old-style definition, this list will be null. In case of new-style definition, there is only two allowed parameter list:

* No parameters. Marked with void.

int main(void) { /\* ... \*/ }

* Command line arguments.

int main(int argc, char \*argv[]) { /\* ... \*/ }

or

int main(int argc, char \*\*argv) { /\* ... \*/ }

The two cases are actually the same case, since arrays are changed to pointers in parameter types. Also note that do not check the names of the parameters, the parameters may have other names. However, the names cannot be omitted.

MiddleCodeGenerator.cs

List<Type> typeList =

SymbolTable.CurrentFunction.Type.TypeList;

If typelist is not null, and there are two parameters, we check if the command line argument case applies. The first parameter shall be a signed or unsigned integer while the second parameter shall be a pointer to pointer to signed or unsigned character.

if ((typeList != null) && (typeList.Count == 2)) {

Assert.Error(typeList[0].IsInteger() &&

typeList[1].IsPointer() &&

typeList[1].PointerType.IsPointer() &&

typeList[1].PointerType.PointerType.IsChar(),

"main", Message.Invalid\_parameter\_list);

AssemblyCodeGenerator.ArgumentCodeList();

}

If typelist is null, we have the old-style function definition, in which case we do not perform any type checking at all. If typelist is empty, we have the void-marked empty parameter list.

else {

Assert.Error((typeList == null) || (typeList.Count == 0),

"main", Message.Invalid\_parameter\_list);

}

}

}

The FunctionEnd method is called when the statements of the function have been parsed. Its task is to check the return statement at the end of the function and generate a static symbol holding the assembly code of the function.

The first step is to backpatch the next set of the statement. Each statement has a next set holding jump instructions. For instance, a for or while loop has a next sets holding jump instructions jumping out of the loop. We add a new empty statement and backpatch the next set to it.

public static void FunctionEnd(Statement statement) {

MiddleCode nextCode =

AddMiddleCode(statement.CodeList, MiddleOperator.Empty);

Backpatch(statement.NextSet, nextCode);

If the return type of the function is void, we need to add a Return instruction at the end of the function.

if (SymbolTable.CurrentFunction.Type.ReturnType.IsVoid()) {

AddMiddleCode(statement.CodeList, MiddleOperator.Return);

if (SymbolTable.CurrentFunction.UniqueName.Equals

(AssemblyCodeGenerator.MainName)) {

AddMiddleCode(statement.CodeList, MiddleOperator.Exit);

}

}

We then add a FunctionEnd middle code instruction. Its only purpose is that the middle code optimizer will report an error if it is possible to reach the EndFunction instruction in a function not returning void.

AddMiddleCode(statement.CodeList, MiddleOperator.FunctionEnd,

SymbolTable.CurrentFunction);

When we finally have added all middle code instructions, we optimize the middle code of the function. The optimizer transforms the middle code in several ways, see Chapter 10.

MiddleCodeOptimizer middleCodeOptimizer =

new MiddleCodeOptimizer(statement.CodeList);

middleCodeOptimizer.Optimize();

When the middle code has been optimized, we generate assembly code of the function, see Chapter 11.

List<AssemblyCode> assemblyCodeList = new List<AssemblyCode>();

AssemblyCodeGenerator.GenerateAssembly(statement.CodeList,

assemblyCodeList);

As mention in the introduction chapter of this book, we generate two kinds of target code. In the first part of the book we generate assembly code text for 64-bit Linux on Intel processors, which is than assembled and linked in traditional manner. In the second part of the book we generate an executable target file for 16-bit Windows. In the case where we need to distinguish between the two cases, we test whether the Start.Linux or Start.Windows static variable is true. In this chapter, we perform the actions for the Linux target system, see Chapter 12 for the Windows target system.

In the Linux case, we need to generate the assembly text from the assembly code list. We also need to generate the extern set; that is, the set of accesses of static variables with external linkage and the calls to all functions with external linkage.

if (Start.Linux) {

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

if (SymbolTable.CurrentFunction.UniqueName.

Equals(AssemblyCodeGenerator.MainName)) {

textList.AddRange(SymbolTable.InitSymbol.TextList);

if (SymbolTable.ArgsSymbol != null) {

textList.AddRange(SymbolTable.ArgsSymbol.TextList);

}

}

else {

textList.Add("section .text");

}

ISet<string> externSet = new HashSet<string>();

AssemblyCodeGenerator.LinuxTextList(assemblyCodeList, textList,

externSet);

When the text list and access set has been generated, we create a static symbol that we add to the global static set. In this way, the assembly code text of the function will be added to final assembly file. The symbols of the global static set is then handled by the Main method of Section 1.2

StaticSymbol staticSymbol =

new StaticSymbolLinux(SymbolTable.CurrentFunction.UniqueName,

textList, externSet);

SymbolTable.StaticSet.Add(staticSymbol);

}

if (Start.Windows) {

// ...

}

Finally, we set the former symbol table, which is the parent table of this function’s table, to be the current table. We also set the current function to null, which strictly speaking in not necessary. However, it is logical to set it to null when the parser does not parse the code of a function.

SymbolTable.CurrentTable = SymbolTable.CurrentTable.ParentTable;

SymbolTable.CurrentFunction = null;

}

### Specifier List

#### Structs and Unions

MiddleCodeGenerator.cs

public static void StructUnionHeader(string optionalName, Sort sort) {

if (optionalName != null) {

Type type = new Type(sort);

SymbolTable.CurrentTable.AddTag(optionalName, type);

}

}

The StructUnionSpecifier method adds a struct or union. The sort parameter is either Sort.Struct or Sort.Union. If the optional name is not null, we look up the type (it has been added by StructOrUnionHeader above) and sets its member map, which is given by the entry map of the current symbol table.

public static Type StructUnionSpecifier(string optionalName, Sort sort) {

if (optionalName != null) {

Type type = SymbolTable.CurrentTable.LookupTag(optionalName, sort);

type.MemberMap = SymbolTable.CurrentTable.EntryMap;

type.MemberList = SymbolTable.CurrentTable.EntryList;

return type;

}

If the optional name is null, we create and return a type with the entry map of the current symbol table.

else {

return (new Type(sort, SymbolTable.CurrentTable.EntryMap,

SymbolTable.CurrentTable.EntryList));

}

}

The LookupStructUnion method looks a struct or union. The sort parameter is either Sort.Struct or Sort.Union.

public static Type LookupStructUnion(string name, Sort sort) {

Type type = SymbolTable.CurrentTable.LookupTag(name, sort);

If the struct or union exists, we simply return its type.

if (type != null) {

return type;

}

If the struct or union does not exist, we create a new type and add it to the tag map. However, the type lacks a member map, which means that it is not yet possible to define variables of the type. It is only when the struct or union becomes properly defined, with a member map, this it will be possible to define variables.

else {

type = new Type(sort);

SymbolTable.CurrentTable.AddTag(name, type);

return type;

}

}

#### Enumeration

The EnumItem method stores the item in the symbol table and checks the optional initialization symbol. The type of the item is constant signed integer.

MiddleCodeGenerator.cs

public static Symbol EnumItem(string itemName,

Symbol optInitializerSymbol) {

Type itemType = new Type(Sort.SignedInt, true);

itemType.Constant = true;

The value of the item is either set by the initialization symbol value, if present, or by the enumeration value stack. The stack holds the value of the current enumeration list. It is popped and pushed with the item value plus one. In that way will each item hold the value of the previous item plus one, the first item holds the value zero.

BigInteger value;

if (optInitializerSymbol != null) {

Assert.Error(optInitializerSymbol.Type.IsIntegral(), itemName,

Message.Non\_\_integral\_enum\_value);

Assert.Error(optInitializerSymbol.IsValue(), itemName,

Message.Non\_\_constant\_enum\_value);

CCompiler\_Main.Parser.EnumValueStack.Pop();

value = (BigInteger) optInitializerSymbol.Value;

}

else {

value = CCompiler\_Main.Parser.EnumValueStack.Pop();

}

Symbol itemSymbol = new Symbol(itemName, null, itemType, value);

SymbolTable.CurrentTable.AddSymbol(itemSymbol);

CCompiler\_Main.Parser.EnumValueStack.Push(value + 1);

return itemSymbol;

}

The EnumSpecifier method add the enumerator to the current symbol table if optional name is not null.

public static Type EnumSpecifier(string optionalName,

ISet<Pair<Symbol,bool>> enumSet) {

Type enumType = new Type(Sort.Enumeration, enumSet);

if (optionalName != null) {

SymbolTable.CurrentTable.AddTag(optionalName, enumType);

}

return enumType;

}

The LookupEnum method looks up the name of an enumeration. Technically, there is no meaning by looking up an enumeration name since an enumeration value has the signed integer type. However, if an unknown name is refereed, an error shall be reported. The following code is value only if an enumeration with the name CarMake has earlier been defined.

Enum CarMake car;

MiddleCodeGenerator.cs

public static Type LookupEnum(string name) {

Type type = SymbolTable.CurrentTable.LookupTag(name, Sort.Enumeration);

Assert.Error(type != null, name, Message.Tag\_not\_found);

return type;

}

## Declarator

The Declarator method handles the cases without assignment or bitfield. The declarator is possible made up by a sequence of pointer, array, or functions declarations, and we add the type of the specifier as the end type of the declarator.

MiddleCodeGenerator.cs

public static void Declarator(Specifier specifier, Declarator declarator){

declarator.Add(specifier.Type);

A function can only hold extern or static storage. This is in fact the only storage check that is performed outside the Specifier class of Chapter 4, since we need the declarator to decide that it is in fact a function declaration.

Storage storage = specifier.Storage;

if (declarator.Type.IsFunction()) {

Assert.Error((storage == Storage.Static) ||

(storage == Storage.Extern), storage, Message.

Only\_extern\_or\_static\_storage\_allowed\_for\_functions);

}

We create the symbol holding the name, external linkage, storage, and type of the declarator, and add it to the current symbol table.

Symbol symbol = new Symbol(declarator.Name, specifier.ExternalLinkage,

storage, declarator.Type);

SymbolTable.CurrentTable.AddSymbol(symbol);

If the symbol has static storage and is not a function, we add it to the global static set. We call the Value method in the ConstantExpression class to generate the global static set. If it is a function definition, it becomes added by the FunctionEnd method of Section 3.2.2. If it is function declaration, is shall not be stored in the global static set at all. Declarations are stored in the symbol table only, not in the global static set.

if ((storage == Storage.Static) && !type.IsFunction()) {

SymbolTable.StaticSet.Add(ConstantExpression.Value(symbol));

}

}

The InitializedDeclarator method handles the cases where the declarator becomes initialized with a value. Similar to the Declarator case above, we begin by adding the type of the specifier to the declarator. Then we obtain the

public static List<MiddleCode> InitializedDeclarator(Specifier specifier,

Declarator declarator, object initializer){

declarator.Add(specifier.Type);

Type type = declarator.Type;

Storage storage = specifier.Storage;

string name = declarator.Name;

A function cannot be initialized, neither can a symbol of extern or typedef storage, or a member of a struct or union.

Assert.Error(!type.IsFunction(), null,

Message.Functions\_cannot\_be\_initialized);

Assert.Error(storage != Storage.Extern, name,

Message.Extern\_cannot\_be\_initialized);

Assert.Error(storage != Storage.Typedef, name,

Message.Typedef\_cannot\_be\_initialized);

Assert.Error((SymbolTable.CurrentTable.Scope != Scope.Struct) &&

(SymbolTable.CurrentTable.Scope != Scope.Union),

name, Message.Struct\_or\_union\_field\_cannot\_be\_initialized);

In case of static storage, we call the GenerateStatic method in the GenerateStaticInitializer class to generate a middle code list holding the initialization code.

if (storage == Storage.Static) {

List<MiddleCode> middleCodeList =

GenerateStaticInitializer.GenerateStatic(type, initializer);

We create a symbol with the name and type of the declarator and the external linkage and storage of the specifier and add it to the current symbol table.

Symbol symbol = new Symbol(name, specifier.ExternalLinkage,

storage, type);

SymbolTable.CurrentTable.AddSymbol(symbol);

Since the storage is static, we also create a static symbol that we add to the global static set. Similar to the Declarator case above, we call the Value method in the ConstantExpression class to generate the value. However, we return an empty code list after we have added the static symbol to the global static set.

StaticSymbol staticSymbol =

ConstantExpression.Value(symbol.UniqueName, type, middleCodeList);

SymbolTable.StaticSet.Add(staticSymbol);

return (new List<MiddleCode>());

}

In the non-static cases, we call the GenerateAuto method in the GenerateAutoInitializer class, which return a list of middle code instructions. The value in this case is not a static value. Instead, it is alist of middle code instructions that assign the value to the symbol in run-time.

else {

Symbol symbol =

new Symbol(name, specifier.ExternalLinkage, storage, type);

symbol.Offset = SymbolTable.CurrentTable.CurrentOffset;

List<MiddleCode> codeList =

GenerateAutoInitializer.GenerateAuto(symbol, initializer);

SymbolTable.CurrentTable.AddSymbol(symbol);

return codeList;

}

}

The BitfieldDeclarator handles the case with bitfields. What is special in this case is that the declarator can actually be omitted.

public static void BitfieldDeclarator(Specifier specifier,

Declarator declarator, Symbol bitsSymbol) {

Storage storage = specifier.Storage;

Bitfields are only allowed in structs and can only have auto or register storage.

Assert.Error(SymbolTable.CurrentTable.Scope == Scope.Struct,

bitsSymbol, Message.Bitfields\_only\_allowed\_on\_structs);

Assert.Error((storage == Storage.Auto) || (storage == Storage.Register),

null, Message.

Only\_auto\_or\_register\_storage\_allowed\_in\_struct\_or\_union);

We calculate the number of bits by parsing the text of the

object bitsValue = bitsSymbol.Value;

int bits = int.Parse(bitsValue.ToString());

If the declarator is not null (it has been present in the code) we add the specifier type to it and extract its final type, which must be integral. The number of bits must be a value between one and the type size in bites (eight times the type size in bytes), inclusive.

if (declarator != null) {

declarator.Add(specifier.Type);

Type type = declarator.Type;

Assert.Error(type.IsIntegral(), type,

Message.Non\_\_integral\_bits\_expression);

Assert.Error((bits >= 1) && (bits <= (8 \* type.Size())),

bitsValue, Message.Bits\_value\_out\_of\_range);

If the number of bits is less then the type size in bits, we need to define the bit mask that is used to delete the bits not covered by the mask.

if (bits < (8 \* type.Size())) {

type.SetBitfieldMask(bits);

}

Symbol symbol = new Symbol(declarator.Name, specifier.ExternalLinkage,

storage, type);

SymbolTable.CurrentTable.AddSymbol(symbol);

Contrary to the previous cases, we do not add the symbol to the global static set since it cannot be static.

}

If the declarator is null (it has been omitted in the code) we just check that the number of bits is in range: at least one and at most the type size in bits.

else {

Assert.Error((bits >= 1) && (bits <= (8 \* 4)), bitsValue,

Message.Bits\_value\_out\_of\_range);

}

}

### Pointer declarator

The PointerDeclarator method takes a type list and a declarator.

MiddleCodeGenerator.cs

public static Declarator PointerDeclarator(List<Type> typeList,

Declarator declarator) {

If the declarator is null, we simply create a new declarator since we always need a declarator, even in the cases where is has been omitted.

if (declarator == null) {

declarator = new Declarator(null);

}

We iterate through the pointer type list and add each pointer type to the declarator. In this way, the declarator is pointer.

foreach (Type pointerType in typeList) {

Type pointerType = new Type((Type) null);

pointerType.Constant = type.Constant;

pointerType.Volatile = type.Volatile;

declarator.Add(pointerType);

}

We will always have at least a declarator with a certain type or a pointer type, since it is syntactically impossible to omit both the declarator and the pointer list.

return declarator;

}

### Direct Declarator

MiddleCodeGenerator.cs

public static Declarator ArrayType(Declarator declarator,

Expression optionalSizeExpression) {

if (declarator == null) {

declarator = new Declarator(null);

}

If the array size expression is present, the parser has already made sure it is a constant expression of integral type. But we must also check that the array (the value of the expression) is positive.

Type arrayType;

if (optionalSizeExpression != null) {

Symbol optSizeSymbol = optionalSizeExpression.Symbol;

int arraySize = (int) ((BigInteger) optSizeSymbol.Value);

Assert.Error(arraySize > 0, arraySize,

Message.Non\_\_positive\_array\_size);

arrayType = new Type(arraySize, null);

}

If the array size expression is not present, we create a new array type of zero size.

else {

arrayType = new Type(0, null);

}

We add the array type to the declarator, which we return. Note that the type of the array is yet unknown, which is why it is null.

declarator.Add(arrayType);

return declarator;

}

The next kind of direct declarator is the new-style function declarator, with parentheses enclosing a variable parameter list. A variable parameter list is a parameter list that may be finished with the variable marker (‘…’).

The NewFunctionDeclaration handles a new-style function declaration.

MiddleCodeGenerator.cs

public static Declarator NewFunctionDeclaration(Declarator declarator,

List<Symbol> parameterList,

bool ellipse) {

It is not allowed to add a variable marker to an empty parameter list, it must always be at least one parameter.

if (parameterList.Count == 0) {

Assert.Error(!ellipse, "...",

Message.An\_elliptic\_function\_must\_have\_at\_least\_one\_parameter);

}

It is allowed to have a void parameter if it is the only parameter in the list, and there is no variable marker.

else if ((parameterList.Count == 1) && parameterList[0].Type.IsVoid()) {

Assert.Error(parameterList[0].Name == null,

parameterList[0].Name,

Message.A\_void\_parameter\_cannot\_be\_named);

Assert.Error(!ellipse, "...", Message.

An\_elliptic\_function\_cannot\_have\_a\_void\_parameter);

parameterList.Clear();

}

Otherwise, a void parameter is not allowed.

else {

foreach (Symbol symbol in parameterList) {

Assert.Error(!symbol.Type.IsVoid(),

Message.Invalid\_void\_parameter);

}

}

Finally, we create a new-style function type with the parameter list and ellipse status, add it to the declarator, and return the declarator. Note that we do not add a return type (the first parameter is null), since it will be added to the declarator by the specifier.

declarator.Add(new Type(null, parameterList, ellipse));

return declarator;

}

The OldFunctionDeclaration method handles the old-style function declaration. The only thing we need to test is that the same name does not occur twice in the name list.

MiddleCodeGenerator.cs

public static Declarator OldFunctionDeclaration(Declarator declarator,

List<string> nameList) {

We create a set and add the names of the name list to the set. The Add method returns true as long as the name does not already occur in the set.

ISet<string> nameSet = new HashSet<string>();

foreach (string name in nameList) {

Assert.Error(nameSet.Add(name), name, Message.Name\_already\_defined);

}

Finally, we create an old-style function type with the name list, add it to the declarator, and return the declarator. Again, note that we do not add a return type since it will be added to the declarator by the specifier.

declarator.Add(new Type(null, nameList));

return declarator;

}

### Parameters

MiddleCodeGenerator.cs

public static Symbol Parameter(Specifier specifier,

Declarator declarator) {

Storage storage = specifier.Storage;

Type specifierType = specifier.Type;

string name;

Type type;

if (declarator != null) {

name = declarator.Name;

declarator.Add(specifierType);

type = declarator.Type;

}

else {

name = null;

type = specifierType;

}

If the parameter is an array, we set the type to a pointer to the array type, and if it is a function, we set the type to a pointer to the function. In both cases the type becomes constant.

if (type.IsArray()) {

type = new Type(type.ArrayType);

type.Constant = true;

}

else if (type.IsFunction()) {

type = new Type(type);

type.Constant = true;

}

We create and return a symbol, with the name, storeage, and type of the parameter. The second value is false since parameters do not have external linkage. The last value is true since it is a parameter, which we need to know that in the assembly code generation phase in Chapter 11.

return (new Symbol(name, false, storage, type, true));

}

#### Abstract Declarator

## Statements

### The If Statement

The if statement backpatches the true set to the beginning of the statement and the false set to the middle code instruction following the statement. The if-else statement backpatches the true set to the beginning of the true statement and the false set to the beginning of the false statement.

The IfStatement method handles the if statement of the parser. Since the if-statement accepts logical expressions only, we need to start with making sure the expression is or becomes a logical expression.

MiddleCodeGenerator.cs

public static Statement IfStatement(Expression expression,

Statement innerStatement){

expression = TypeCast.ToLogical(expression);

Since we use the value of the expression in the if-statement, we only use the long list.

List<MiddleCode> codeList = expression.LongList;

AddMiddleCode(codeList, MiddleOperator.CheckTrackMapFloatStack);

We backpatch the true value of the expression to the first instruction in the middle code list of the inner statement. This means that when the expression is true, the program will jump to the beginning of the code list of the inner statement.

Backpatch(expression.Symbol.TrueSet(), innerStatement.CodeList);

codeList.AddRange(innerStatement.CodeList);

We add a jump instruction at the end of the middle code that shall jump to the instruction following this if-statement.

MiddleCode nextCode = AddMiddleCode(codeList, MiddleOperator.Goto);

We define the next set of the if-statement, it holds the instructions that jump to the instruction after the if- statement. It is the union of the next set of the inner statement, the false set of the expression, and the jump instruction just added to the list.

ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();

nextSet.UnionWith(innerStatement.NextSet);

nextSet.UnionWith(expression.Symbol.FalseSet());

nextSet.Add(nextCode);

Finally, we return on object of the Statement class holding the code list and next set of the if-statement.

return (new Statement(codeList, nextSet));

}

The IfElseStatement methods is like the IfStatement method above. The difference is that we have two inter statements. On statement that the program jumps to in case the expression is evaluated to a true value, and one statement where it jumps to in case of a false value.

public static Statement IfElseStatement(Expression expression,

Statement trueStatement,

Statement falseStatement) {

expression = TypeCast.ToLogical(expression);

List<MiddleCode> codeList = expression.LongList;

AddMiddleCode(codeList, MiddleOperator.CheckTrackMapFloatStack);

We backpatch the true and false set to the beginning of the code list of the true and false statement, respectively.

Backpatch(expression.Symbol.TrueSet(), trueStatement.CodeList);

Backpatch(expression.Symbol.FalseSet(), falseStatement.CodeList);

codeList.AddRange(trueStatement.CodeList);

We add a jump instruction after the end of the true statement code list for it to jump out of the if-else-statement.

MiddleCode gotoCode = AddMiddleCode(codeList, MiddleOperator.Goto);

codeList.AddRange(falseStatement.CodeList);

The next set of the if-else-statement is the union of the next sets of the true and false statement, and the jump instruction just added to the code.

ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();

nextSet.UnionWith(trueStatement.NextSet);

nextSet.UnionWith(falseStatement.NextSet);

nextSet.Add(gotoCode);

Finally, we return on object of the Statement class holding the code list and next set of the if-statement.

return (new Statement(codeList, nextSet));

}

### The Switch Statement

We need a map to keep track of the case statements. Since the switch statements can be nested, we need a stack to keep track of the case maps (m\_caseMapStack). In the same way, we need a stack keep track of the default statements (m\_caseMapStack), and a stack of sets to keep track of the break statements (m\_breakStack).

MiddleCodeGenerator.cs

private static Stack<IDictionary<BigInteger, MiddleCode>> m\_caseMapStack =

new Stack<IDictionary<BigInteger, MiddleCode>>();

private static Stack<MiddleCode> m\_defaultStack = new Stack<MiddleCode>();

private static Stack<ISet<MiddleCode>> m\_breakSetStack =

new Stack<ISet<MiddleCode>>();

The SwitchHeader method is called before the parsing of the switch statement. Its task is to push a new map to the case map stack, a null reference to the default stack, and a new set to the break set stack.

public static void SwitchHeader() {

m\_caseMapStack.Push(new Dictionary<BigInteger,MiddleCode>());

DefaultStack.Push(null);

BreakSetStack.Push(new HashSet<MiddleCode>());

}

The SwitchStatement method is called after the parsing of the switch statement. Its task is to generate middle code for the case and default statements, and to backpatch the break statements.

public static Statement SwitchStatement(Expression switchExpression,

Statement innerStatement) {

Since the switch statement need an integral value rather than a logical value we need to, contrary to the if- statement above, type cast a potential logical value to an integral value.

switchExpression = TypeCast.LogicalToIntegral(switchExpression);

Since we need the value of the switch statement, we are only interested of the long list of the switch expression.

List<MiddleCode> codeList = switchExpression.LongList;

Each case value is type casted to the type of the switch expression.

Type switchType = switchExpression.Symbol.Type;

foreach (KeyValuePair<BigInteger,MiddleCode> entry

in m\_caseMapStack.Pop()) {

BigInteger caseValue = entry.Key;

MiddleCode caseTarget = entry.Value;

Symbol caseSymbol = new Symbol(switchType, caseValue);

We add code where we compare the case value with the switch value and jump to the matching middle code instruction if the values are equal.

AddMiddleCode(codeList, MiddleOperator.Case, caseTarget,

switchExpression.Symbol, caseSymbol);

}

After the case values, we add a case end instruction. The reason for this is that the register used for the case comparison in the generated assembly code shall be unallocated.

AddMiddleCode(codeList, MiddleOperator.CaseEnd,

switchExpression.Symbol);

Similar to the if- statement, the switch- statement has a next set; that is, a set of middle code instruction that shall jump to the instruction following the switch statement.

ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();

If there is a default statement, we jump to it if the switch statement does not match any of the case statement. In that case, the next set becomes empty.

MiddleCode defaultCode = DefaultStack.Pop();

if (defaultCode != null) {

AddMiddleCode(codeList, MiddleOperator.Goto, defaultCode);

}

If there is no default statement, we instead add a jump instruction to the next set.

else {

nextSet.Add(AddMiddleCode(codeList, MiddleOperator.Goto));

}

We finally add the code list of the inner statement to the resulting code list.

codeList.AddRange(innerStatement.CodeList);

The next set is the union of the next set of the inner statement and the break statements of the switch statement.

nextSet.UnionWith(innerStatement.NextSet);

nextSet.UnionWith(BreakSetStack.Pop());

return (new Statement(codeList, nextSet));

}

### The Case Statement

The case statement must comply with the following demands:

* Main.CaseMapStack must not be empty. If it is empty, the case statements misses a surrounding switch statements.
* The case expression must be integral and constant and thereby possible to evaluate by the compiler.
* Since Main.CaseMapStack is not empty, there is at least one surrounding switch statement. We call peek on the stack to obtain the map holding the case expression value of the closest surrounding switch statement, add the case value with the line number of the case statement, and check that there is no earlier cases expression with the same value (put returns the old value if the key is already present in the map, null if is not). Technically, it is possible to store null values in the map, in which case put would return null even when the key is present in the map. However, in this book, we never store null values in any map.

If the case map stack is empty, we have a case statement without a enclosing switch statement and an error is reported.

public static Statement CaseStatement(Expression expression,

Statement statement) {

Assert.Error(m\_caseMapStack.Count > 0, Message.Case\_without\_switch);

expression = TypeCast.LogicalToIntegral(expression);

If the value of the symbol is null, we have a non-constant case value, and an error is reported.

Assert.Error(expression.Symbol.Value != null, expression.Symbol.Name,

Message.Non\_\_constant\_case\_value);

BigInteger caseValue = (BigInteger) expression.Symbol.Value;

We extract the current case map from the top of the case map stack and adds the current case value to the map.

IDictionary<BigInteger, MiddleCode> caseMap = m\_caseMapStack.Peek();

I the map already contains the value we have two case values with the same value and an error is reported.

Assert.Error(!caseMap.ContainsKey(caseValue), caseValue,

Message.Repeated\_case\_value);

Finally, we add the case value and the first instruction of the statement’s code

caseMap.Add(caseValue, GetFirst(statement.CodeList));

return statement;

}

The GetFirst method returns the first instruction in the middle code instruction list. It also adds an empty instruction in case the list is empty. This is done for the backpatching to always have an instruction to set as the target.

private static MiddleCode GetFirst(List<MiddleCode> list) {

if (list.Count == 0) {

AddMiddleCode(list, MiddleOperator.Empty);

}

return list[0];

}

### The Default Statement

Technically, the default statement does not have to be placed after the last case statements. However, we cannot have more than one default statement in a switch statement.

public static Statement DefaultStatement(Statement statement) {

If the default stack is empty, we have a default statement without an enclosing switch statement and an error is reported.

Assert.Error(DefaultStack.Count > 0, Message.Default\_without\_switch);

If the value on the top of the stack is not null, we have a switch statement with two default statement and an error is reported.

Assert.Error(DefaultStack.Pop() == null, Message.Repeted\_default);

Finally, we add the default middle code instruction at the top of the stack.

DefaultStack.Push(GetFirst(statement.CodeList));

return statement;

}

### The While Statement

Similar to the break set stack above, we also need a continue set stack.

MiddleCodeGenerator.cs

private static Stack<ISet<MiddleCode>> m\_continueSetStack =

new Stack<ISet<MiddleCode>>();

The LoopHeader method is called before the parsing of a while, do, or for statement. Its task is to add empty sets at the top of the break set stack and continue set stack.

public static void LoopHeader() {

m\_breakSetStack.Push(new HashSet<MiddleCode>());

m\_continueSetStack.Push(new HashSet<MiddleCode>());

}

The WhileStatement method is called after the while statement has been parsed. Similar to the if statement above, it starts by type casting the expression into logical type.

public static Statement WhileStatement(Expression expression,

Statement innerStatement) {

expression = TypeCast.ToLogical(expression);

List<MiddleCode> codeList = expression.LongList;

AddMiddleCode(codeList, MiddleOperator.CheckTrackMapFloatStack);

We backpatchar the true set of the expression to the code list of the inner statement; that is, if the expression is true the program shall jump to the beginning of the inner statement.

Backpatch(expression.Symbol.TrueSet, innerStatement.CodeList);

codeList.AddRange(innerStatement.CodeList);

We define the next set of the while statement, and a jump instruction.

ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();

nextSet.Add(AddMiddleCode(codeList, MiddleOperator.Goto,

GetFirst(codeList)));

We also add the false set of the expression and the break set of the inner statement to the next set.

nextSet.UnionWith(expression.Symbol.FalseSet);

nextSet.UnionWith(m\_breakSetStack.Pop());

We backpatch the next set of the inner statement and its continue set to the code list; that is, the jump instruction of the sets shall jump back to the beginning of the while statement.

Backpatch(innerStatement.NextSet, codeList);

Backpatch(m\_continueSetStack.Pop(), codeList);

return (new Statement(codeList, nextSet));

}

### The Do Statement

MiddleCodeGenerator.cs

public static Statement DoStatement(Statement innerStatement,

Expression expression) {

List<MiddleCode> codeList = innerStatement.CodeList;

Backpatch(innerStatement.NextSet, codeList);

AddMiddleCode(codeList, MiddleOperator.CheckTrackMapFloatStack);

codeList.AddRange(expression.LongList);

Backpatch(expression.Symbol.TrueSet, codeList);

Backpatch(m\_continueSetStack.Pop(), codeList);

ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();

nextSet.UnionWith(expression.Symbol.FalseSet);

nextSet.UnionWith(m\_breakSetStack.Pop());

AddMiddleCode(codeList, MiddleOperator.Goto, GetFirst(codeList));

return (new Statement(codeList, nextSet));

}

### The For Statement

In the ForStatement method, we make difference of the short and long list of the expressions. If the test expression is present, we want its value, and therefore we re interested in its long list. On the other hand, if the initializer or next expressions are present we are only interested in their potential side effekts.

MiddleCodeGenerator.cs

public static Statement ForStatement(Expression initializerExpression,

Expression testExpression, Expression nextExpression,

Statement innerStatement) {

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

ISet<MiddleCode> nextSet = new HashSet<MiddleCode>();

We add an empty instruction as the target of the test expression, since we do not know if the test expression is not null.

MiddleCode testTarget = AddMiddleCode(codeList, MiddleOperator.Empty);

If the initializer expression is not null, we add its short list. Note that we the short list instead of the long list since we are only interested of the side effects of the expression, not its value. We also backpatch both its true set and false set to the test target; that is, the beginning of the test expression code.

if (initializerExpression != null) {

codeList.AddRange(initializerExpression.ShortList);

Backpatch(initializerExpression.Symbol.TrueSet, testTarget);

Backpatch(initializerExpression.Symbol.FalseSet, testTarget);

}

If the test expression is not null, we start by type casting it into a logical value and add its long list to the code. Not that we add the long list rather than the short list in this case, since we need the value of the expression, rather than just its side effects.

if (testExpression != null) {

testExpression = TypeCast.ToLogical(testExpression);

codeList.AddRange(testExpression.LongList);

AddMiddleCode(codeList, MiddleOperator.CheckTrackMapFloatStack);

We backpatch the true set of the test expression to the beginning or the code list of the inner statement.

Backpatch(testExpression.Symbol.TrueSet, innerStatement.CodeList);

nextSet.UnionWith(testExpression.Symbol.FalseSet);

}

We add the code list of the inner statement. We then add the next target instruction and backpatch the next set of the code of the inner statement since we do not know if the next expression is not null.

codeList.AddRange(innerStatement.CodeList);

MiddleCode nextTarget = AddMiddleCode(codeList, MiddleOperator.Empty);

Backpatch(innerStatement.NextSet, nextTarget);

If the next expression is not null, we add its short list and backpatch its true set and false set to the

if (nextExpression != null) {

codeList.AddRange(nextExpression.ShortList);

Backpatch(nextExpression.Symbol.TrueSet, testTarget);

Backpatch(nextExpression.Symbol.FalseSet, testTarget);

}

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.

AddMiddleCode(codeList, MiddleOperator.Goto, testTarget);

Backpatch(m\_continueSetStack.Pop(), nextTarget);

nextSet.UnionWith(m\_breakSetStack.Pop());

return (new Statement(codeList, nextSet));

}

### The Label and Goto Statements

MiddleCodeGenerator.cs

public static IDictionary<string, MiddleCode> m\_labelMap =

new Dictionary<string, MiddleCode>();

public static Statement LabelStatement(string labelName,

Statement statement) {

If the label map already hold the label name as a key, we have two labels with the same name and an error is reported.

Assert.Error(!m\_labelMap.ContainsKey(labelName),

labelName, Message.Defined\_twice);

m\_labelMap.Add(labelName, GetFirst(statement.CodeList));

return statement;

}

We also have the goto set map m\_gotoSetMap with the label name as key and the goto middle code instruction as value.

public static IDictionary<string, ISet<MiddleCode>> m\_gotoSetMap =

new Dictionary<string, ISet<MiddleCode>>();

In the GotoStatement method we start by adding middle code holding a goto instruction.

public static Statement GotoStatement(string labelName) {

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

MiddleCode gotoCode = AddMiddleCode(gotoList, MiddleOperator.Goto);

If the label name is already a key in the goto set map, we look up the goto set and a the goto middle code instruction. This situation occurs in when we encounter the first jump to the label.

if (m\_gotoSetMap.ContainsKey(labelName)) {

ISet<MiddleCode> gotoSet = m\_gotoSetMap[labelName];

gotoSet.Add(gotoCode);

}

If the label name is not already a key in the goto set map, we create a new goto set to which we add the goto instruction, and then we add the label name and goto set to the the goto set map. This situation occurs when we encounter the the jumps following the first jump to the label. In this way, we have a map where each label name is associated with the goto instruction jump to that label.

else {

ISet<MiddleCode> gotoSet = new HashSet<MiddleCode>();

gotoSet.Add(gotoCode);

m\_gotoSetMap.Add(labelName, gotoSet);

}

return (new Statement(gotoList));

}

The BackpatchGoto method is called at the end of each function definition. Its iterates through the goto set map and, for each label name, we look up the label middle code instruction and backpatch the goto set to that instruction.

public static void BackpatchGoto() {

foreach (KeyValuePair<string,ISet<MiddleCode>> entry in m\_gotoSetMap) {

string labelName = entry.Key;

ISet<MiddleCode> gotoSet = entry.Value;

If the label name is not a key in the label map, we have a goto statement to an unknown label and we report an error.

MiddleCode labelCode;

Assert.Error(m\_labelMap.TryGetValue(labelName, out labelCode),

labelName, Message.Missing\_goto\_address);

Backpatch(gotoSet, labelCode);

}

}

### Return Statement

The GenerateReturnStatement method is surprisely complicated, which due to the fact the return statement shall be interpreted as an exit statement in case of the main function.

If the surrounding function returns a pointer to a function and the return expression is a or a pointer to a function, we check the they are equal; that is, that they return equal types and have equal parameter lists.

If the sorrounding function returns a pointer and the expression type is array, we compare the pointer type with array type. If the return expression is a string we check that the surrounding function’s return type is a pointer to a (signed or unsigned) character. Otherwise, we just cast the return expression to the surrounding function’s return type.

If the surrounding function is the main function, we cast the return expression to a signed short integer, since we already checked that main does not return void, it has to return an integral type. If the surrounding function is not main, we generate middle code instructions for the return value and function return. Note that there are two different instructions, where the first one sets the return value and the second one performs the return jump.

On the other hand, if there is no return expression we check that the surrounding function returns void. If it is the main function, we add the exit middle code instruction with argument zero. If it is not, we just add the return middle code instruction without precede it with the setting of a return value.

We need to regard whether there is an expression and whether the return statement is located inside the main function.

MiddleCodeGenerator.cs

public static Statement ReturnStatement(Expression expression) {

List<MiddleCode> codeList;

If the expression is not null, we need the check that the function does not return void. If it does we report an error.

if (expression != null) {

Assert.Error(!SymbolTable.CurrentFunction.Type.ReturnType.IsVoid(),

Message.Non\_\_void\_return\_from\_void\_function);

We cast the return expression to the return type of the function.

expression = TypeCast.ImplicitCast(expression,

SymbolTable.CurrentFunction.Type.ReturnType);

codeList = expression.LongList;

If the function is the main function, we shall not return a value. Instead, we shall exit the program execution and return an integer value to the enclosing system.

If the function is not the main function, we return the value of the expression.

AddMiddleCode(codeList, MiddleOperator.Return,

null, expression.Symbol);

if (SymbolTable.CurrentFunction.UniqueName.Equals

(AssemblyCodeGenerator.MainName)) {

AddMiddleCode(codeList, MiddleOperator.Exit);

}

}

If the expression is null, we check that the function returns void. If it does not, we report and error.

else {

Assert.Error(SymbolTable.CurrentFunction.Type.ReturnType.IsVoid(),

Message.Void\_returned\_from\_non\_\_void\_function);

codeList = new List<MiddleCode>();

If the function is the main function, we exit the execution of the program without a return value.

If the function is not the main function, return the function without a return value.

AddMiddleCode(codeList, MiddleOperator.Return);

if (SymbolTable.CurrentFunction.UniqueName.Equals

(AssemblyCodeGenerator.MainName)) {

AddMiddleCode(codeList, MiddleOperator.Exit);

}

}

return (new Statement(codeList));

}

### Optional Expression Statement

If there is an expression, we add its short list to the code list. We choose the short list rather than the long list since we are only interested in the side effects of the expression.

MiddleCodeGenerator.cs

public static Statement ExpressionStatement(Expression expression) {

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

if (expression != null) {

codeList.AddRange(expression.ShortList);

}

return (new Statement(codeList));

}

### Jump Register Statements

MiddleCodeGenerator.cs

public static Statement JumpRegisterStatement(Register register) {

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

AddMiddleCode(codeList, MiddleOperator.JumpRegister, register);

return (new Statement(codeList));

}

### Interrupt Statements

MiddleCodeGenerator.cs

public static Statement InterruptStatement(Expression expression) {

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

AddMiddleCode(codeList, MiddleOperator.Interrupt,

expression.Symbol.Value);

return (new Statement(codeList));

}

### System Call Statements

MiddleCodeGenerator.cs

public static Statement SyscallStatement() {

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

AddMiddleCode(codeList, MiddleOperator.SysCall);

return (new Statement(codeList));

}

## Expressions

The value of a comma expression is the value of the right expression, the value of the left expression is discarded. However, we keep the side effects of both the left and right expressions.

MiddleCodeGenerator.cs

public static Expression CommaExpression(Expression leftExpression,

Expression rightExpression) {

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

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

Note that we add the short list of the left expression to the middle code list. The reason for this is that we are not interested in the value of the left expression, only its side effects.

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

longList.AddRange(leftExpression.ShortList);

longList.AddRange(rightExpression.LongList);

We return the symbol of the right expression, and discard the symbol of the left expression.

return (new Expression(rightExpression.Symbol, shortList, longList));

}

### The Assignment Expression

MiddleCodeGenerator.cs

public static Expression AssignmentExpression(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression){

rightExpression =

TypeCast.ImplicitCast(rightExpression, leftExpression.Symbol.Type);

switch (middleOp) {

case MiddleOperator.Assign:

return Assignment(leftExpression, rightExpression, true);

In cases of compound assignment, we call the matching method to perform the operation, and the Assignment method for the final assignment.

case MiddleOperator.BinaryAdd:

case MiddleOperator.BinarySubtract:

return Assignment(leftExpression,

AdditionExpression(middleOp, leftExpression,

rightExpression));

case MiddleOperator.SignedMultiply:

case MiddleOperator.SignedDivide:

case MiddleOperator.SignedModulo:

return Assignment(leftExpression,

MultiplyExpression(middleOp, leftExpression,

rightExpression));

case MiddleOperator.BitwiseAnd:

case MiddleOperator.BitwiseOr:

case MiddleOperator.BitwiseXOr:

return Assignment(leftExpression,

BitwiseExpression(middleOp, leftExpression,

rightExpression));

default: // shift left, shift right

return Assignment(leftExpression,

ShiftExpression(middleOp, leftExpression,

rightExpression));

}

}

The Assignment method performs simple or compound assignment.

public static Expression Assignment(Expression leftExpression,

Expression rightExpression,

bool simpleAssignment = false) {

In cases of system calls, a specific register is assigned a value.

Register? register = leftExpression.Register;

if (register != null) {

Symbol rightSymbol = rightExpression.Symbol;

We check that the register has the same size (in bytes) as the expression it is assigned to. If it is not, we report an error.

Assert.Error(AssemblyCode.SizeOfRegister(register.Value) ==

rightExpression.Symbol.Type.Size(),

Message.Unmatched\_register\_size);

We add the long list of the right expression, since we need its value, to the final middle code list.

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

longList.AddRange(rightExpression.LongList);

We add the assignment of the value of right expression to the register.

AddMiddleCode(longList, MiddleOperator.AssignRegister,

register, rightExpression.Symbol);

Since this an assignment, the code list holds both the assignment of the register and the side effect of the assignment. Therefore, we add the long list as both the short list and the long list of the result expression.

return (new Expression(rightExpression.Symbol, longList, longList));

}

If the left expression is not a register, we first check that the left expression is assignable. If it is not, we report an error.

else {

Assert.Error(leftExpression.Symbol.Assignable,

leftExpression, Message.Not\_assignable);

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

In case of a simple assignment we add the long list of the left expression to the code list. In case of a compound assignment, the long list of the left expression has already been added to long list of the right expression. If we added the long list in case of a compound assignment, we would add the same code twice.

if (simpleAssignment) {

longList.AddRange(leftExpression.LongList);

If the left expression is of floating type, we need to pop the floating value stack in case of a simple assignment. In case of a compound assignment, we let the value stay on the stack.

if (leftExpression.Symbol.Type.IsFloating()) {

AddMiddleCode(longList, MiddleOperator.PopFloat);

}

}

We type cast the right expression to the type of the left expression and add its long list to the final code list.

rightExpression = TypeCast.ImplicitCast(rightExpression,

leftExpression.Symbol.Type);

longList.AddRange(rightExpression.LongList);

If the expressions hold floating type, we really do not perform an assignment, we simply top the current value on the floating value stack to the left expression.

if (leftExpression.Symbol.Type.IsFloating()) {

AddMiddleCode(longList, MiddleOperator.TopFloat,

leftExpression.Symbol);

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

shortList.AddRange(longList);

In case of floating type, there is a difference between the short list and long list. In the long list, the resulting value of the assignment (which equals the new value of the left expression) shall be used. Therefore, it is preserved on the stack. In the short list, the value is popped from the stack, since the value shall not be used, only the side effect of the assignment.

AddMiddleCode(shortList, MiddleOperator.PopFloat);

return (new Expression(leftExpression.Symbol, shortList, longList));

}

In case of integral type, we add an assign instruction to the final code.

else {

if (leftExpression.Symbol.Type.IsStructOrUnion()) {

AddMiddleCode(longList, MiddleOperator.AssignInit,

leftExpression.Symbol, rightExpression.Symbol);

}

AddMiddleCode(longList, MiddleOperator.Assign,

leftExpression.Symbol, rightExpression.Symbol);

BigInteger? bitFieldMask =

leftExpression.Symbol.Type.GetBitfieldMask();

In case of bitfield, we add an add operation to set the non-relevant bits to zero.

if (bitFieldMask != null) {

Symbol maskSymbol = new Symbol(leftExpression.Symbol.Type,

bitFieldMask);

AddMiddleCode(longList, MiddleOperator.BitwiseAnd,

leftExpression.Symbol, leftExpression.Symbol,

maskSymbol);

}

In the integral case, we set the code list as both the short list and long list in the resulting expression since the assignment is a side effect.

return (new Expression(leftExpression.Symbol, longList, longList));

}

}

}

### The Condition Expression

MiddleCodeGenerator.cs

public static Expression ConditionalExpression(Expression testExpression,

Expression trueExpression,

Expression falseExpression) {

We type cast the test expression to logical type.

testExpression = TypeCast.ToLogical(testExpression);

If the test expression is constant, we simple return the true or false expression depending on whether the test expression is true.

if (ConstantExpression.IsConstant(testExpression)) {

return ConstantExpression.IsTrue(testExpression)

? falseExpression : trueExpression;

}

If both the true and false expressions hold logical types, we keep their types.

if (trueExpression.Symbol.Type.IsLogical() &&

falseExpression.Symbol.Type.IsLogical()) {

We start by backpathing the true and false set of the test expression to the beginning of the true and false expression’s code list.

Backpatch(testExpression.Symbol.TrueSet, trueExpression.LongList);

Backpatch(testExpression.Symbol.FalseSet, falseExpression.LongList);

The resulting true set is the union of the true sets of the true and false expression, and the resulting false set is the union of the false sets of the true and false expression.

ISet<MiddleCode> trueSet = new HashSet<MiddleCode>(),

falseSet = new HashSet<MiddleCode>();

trueSet.UnionWith(trueExpression.Symbol.TrueSet);

trueSet.UnionWith(falseExpression.Symbol.TrueSet);

falseSet.UnionWith(trueExpression.Symbol.FalseSet);

falseSet.UnionWith(falseExpression.Symbol.FalseSet);

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

If the short list of both the true and false expression is empty, it does not matter if the test expression is true or false, and we let the resulting short list be the short list of the true expression (which may be empty).

if (IsEmpty(trueExpression.ShortList) &&

IsEmpty(falseExpression.ShortList)) {

shortList.AddRange(testExpression.ShortList);

}

If the short list of the test expression is not empty, the situation becomes a bit more complicated. We add the long list, rather than the short list, of the test expression to the final short list since we need the value of the test expression in order to jump to the short list of the true of false expression.

else {

shortList.AddRange(testExpression.LongList);

shortList.AddRange(trueExpression.ShortList);

shortList.AddRange(falseExpression.ShortList);

}

We add the long list of the test, true, and false expression to the final long list.

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

longList.AddRange(testExpression.LongList);

longList.AddRange(trueExpression.LongList);

longList.AddRange(falseExpression.LongList);

Finally, we create a new symbol with logical type and the resulting true and false sets.

Symbol symbol = new Symbol(trueSet, falseSet);

return (new Expression(symbol, shortList, longList));

}

If at least one of the true or false expression does not hold logical type, we define maxType as their largest type, and type cast both expressions to that type.

else {

Type maxType = TypeCast.MaxType(trueExpression.Symbol.Type,

falseExpression.Symbol.Type);

trueExpression = TypeCast.ImplicitCast(trueExpression, maxType);

Backpatch(testExpression.Symbol.TrueSet, trueExpression.LongList);

We create a new temporary symbol to hold the result of the condition expression.

Symbol symbol = new Symbol(maxType);

In case of non-floating type, we add the assignment instruction. In case of a floating type, the value is already placed at the floating value stack and we do not need to do anything.

if (maxType.IsFloating()) {

AddMiddleCode(trueExpression.LongList,

MiddleOperator.DecreaseStack);

}

else {

if (trueExpression.Symbol.IsTemporary()) {

foreach (MiddleCode middleCode in trueExpression.LongList) {

if (middleCode[0] == trueExpression.Symbol) {

middleCode[0] = symbol;

}

}

}

else {

AddMiddleCode(trueExpression.LongList, MiddleOperator.Assign,

symbol, trueExpression.Symbol);

}

}

We add the jump to a target code to both the short and long list of the true expression. The target code will be added after the code of the false expression. Its purpose is to jump over the false expression code.

MiddleCode targetCode = new MiddleCode(MiddleOperator.Empty);

AddMiddleCode(trueExpression.ShortList,

MiddleOperator.Goto, targetCode);

AddMiddleCode(trueExpression.LongList,

MiddleOperator.Goto, targetCode);

Similar to the true expression, we type cast and false expression, and backpatch the true set of the test expression to the beginning of the false expression code.

falseExpression = TypeCast.ImplicitCast(falseExpression, maxType);

Backpatch(testExpression.Symbol.FalseSet, falseExpression.LongList);

We also assign the value of the false expression to the temporary symbol if it does not hold floating type. If it holds floating type, the value is already placed at the top of the floating value stack, and we do nothing.

if (!maxType.IsFloating()) {

if (falseExpression.Symbol.IsTemporary()) {

foreach (MiddleCode middleCode in falseExpression.LongList) {

if (middleCode[0] == falseExpression.Symbol) {

middleCode[0] = symbol;

}

}

}

else {

AddMiddleCode(falseExpression.LongList, MiddleOperator.Assign,

symbol, falseExpression.Symbol);

}

}

If both the short lists of the true and false expressions are empty, we just add the short list of the test expression (which may be empty).

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

if (IsEmpty(trueExpression.ShortList) &&

IsEmpty(falseExpression.ShortList)) {

shortList.AddRange(testExpression.ShortList);

}

If not both the short lists of the true and false expressions are empty, we add the long list of the test expression, rather than the short list, since we need to value of the test expression, as well as the short lists of the true and false expressions.

else {

shortList.AddRange(testExpression.LongList);

shortList.AddRange(trueExpression.ShortList);

shortList.AddRange(falseExpression.ShortList);

shortList.Add(targetCode);

}

Finally, add the long lists and return the expression.

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

longList.AddRange(testExpression.LongList);

longList.AddRange(trueExpression.LongList);

longList.AddRange(falseExpression.LongList);

longList.Add(targetCode);

return (new Expression(symbol, shortList, longList));

}

}

### Constant Expression

The ConstantIntegralExpression methods makes sure the expression is indeed constant and either integral or pointer.

MiddleCodeGenerator.cs

public static Expression ConstantIntegralExpression(Expression expression)

{ expression = ConstantExpression.Cast(expression,

Type.SignedLongIntegerType);

Assert.Error(expression != null, expression,

Message.Non\_\_constant\_expression);

Assert.Error(expression.Symbol.Type.IsIntegralOrPointer(),

expression.Symbol, Message.Non\_\_integral\_expression);

return expression;

}

### Logical Expressions

The LogicalOrExpression method returns a true expression is at least one of the left or right expression is true, and a false expression if both the left and right expression is false.

MiddleCodeGenerator.cs

public static Expression LogicalOrExpression(Expression leftExpression,

Expression rightExpression) {

Since we have passed the constant integral expression parsing, we need to check if the expression is constant. If it is constant, we return the constant expression.

Expression constantExpression =

ConstantExpression.Logical(MiddleOperator.LogicalOr,

leftExpression, rightExpression);

if (constantExpression != null) {

return constantExpression;

}

We type cast both the expression to logical type.

leftExpression = TypeCast.ToLogical(leftExpression);

rightExpression = TypeCast.ToLogical(rightExpression);

For the resulting expression to be true, it is enough that one of the left or right expression is true. Therefore, the true set of the resulting expression is the union of the true set of the left and right expression. This results in laze evaluation, if the left expression is evaluated to true, the right expression (including its side effects) shall not evaluated. The false sets, on the other hand, are different. If the left expression is evaluated to false, we need to evaluate the right expression. Therefore, we backpatch the false set of the left expression to the beginning of the right expression code. The false set of the resulting expression is the false set of the right expression.

ISet<MiddleCode> trueSet = new HashSet<MiddleCode>();

trueSet.UnionWith(leftExpression.Symbol.TrueSet);

trueSet.UnionWith(rightExpression.Symbol.TrueSet);

Backpatch(leftExpression.Symbol.FalseSet, rightExpression.LongList);

Symbol symbol = new Symbol(trueSet, rightExpression.Symbol.FalseSet);

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

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

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

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

return (new Expression(symbol, shortList, longList));

}

### Bitwise Expressions

MiddleCodeGenerator.cs

public static Expression BitwiseExpression(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

If the expression can be evaluated to a constant value, we return the constant expression.

Expression constantExpression = ConstantExpression.

Arithmetic(middleOp, leftExpression, rightExpression);

if (constantExpression != null) {

return constantExpression;

}

We find the maximal type of the left and right expression.

Type maxType = TypeCast.MaxType(leftExpression.Symbol.Type,

rightExpression.Symbol.Type);

Symbol resultSymbol = new Symbol(maxType);

The type of the left and right expression must be integral or pointer. Array, string, and functions are converted to pointers.

Assert.Error(maxType.IsIntegralPointerArrayStringOrFunction(),

maxType, Message.Invalid\_type\_in\_bitwise\_expression);

We type cast both expressions into the maximal type.

leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);

rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);

The short list of the resulting expression is simply the short list of the expressions.

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

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

The long list of the resulting expression is the long list of the expressions, and the bitwise operation.

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

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

AddMiddleCode(longList, middleOp, resultSymbol,

leftExpression.Symbol, rightExpression.Symbol);

return (new Expression(resultSymbol, shortList, longList));

}

### Shift Expression

**MiddleCodeGenerator.cs**

public static Expression ShiftExpression(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

First, we check that the left expression holds integral or pointer type. An array, a string, or a function is cast to a pointer.

Assert.Error(leftExpression.Symbol.Type.

IsIntegralPointerArrayStringOrFunction(),

leftExpression, Message.Invalid\_type\_in\_shift\_expression);

If the expression can be evaluated to a constant expression, we return the constant expression.

Expression constantExpression =

ConstantExpression.Arithmetic(middleOp, leftExpression,

rightExpression);

if (constantExpression != null) {

return constantExpression;

}

The right expression is type cast to an unsigned character, which always have a size of one byte.

rightExpression =

TypeCast.ImplicitCast(rightExpression, Type.UnsignedCharType);

The final short list is simple the short lists of the left and right expression.

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

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

The final short list is simple the short lists of the left and right expression.

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

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

The final long list is the short lists of the left and right expression, and the shift operation.

Symbol resultSymbol = new Symbol(leftExpression.Symbol.Type);

AddMiddleCode(longList, middleOp, resultSymbol,

leftExpression.Symbol, rightExpression.Symbol);

return (new Expression(resultSymbol, shortList, longList));

}

### Equality and Relation Expressions

MiddleCodeGenerator.cs

public static Expression RelationalExpression(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression){

First, we check the types of the expression. Everything except struct or union can be compared.

Assert.Error(!leftExpression.Symbol.Type.IsStructOrUnion(),

leftExpression,

Message.Invalid\_type\_in\_expression);

Assert.Error(!rightExpression.Symbol.Type.IsStructOrUnion(),

rightExpression,

Message.Invalid\_type\_in\_expression);

Expression constantExpression =

ConstantExpression.Relation(middleOp, leftExpression,

rightExpression);

if (constantExpression != null) {

return constantExpression;

}

The find the maximal type of the left and right expression types, and type cast both the expressions to that type.

Type maxType = TypeCast.MaxType(leftExpression.Symbol.Type,

rightExpression.Symbol.Type);

leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);

rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);

The final short list is simply the short lists of the left end right expression.

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

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

If the maximal type is unsigned, we change to operator from signed to unsigned.

if (maxType.IsUnsigned()) {

string name = Enum.GetName(typeof(MiddleOperator), middleOp);

middleOp = (MiddleOperator) Enum.Parse(typeof(MiddleOperator),

name.Replace("Signed", "Unsigned"));

}

The final long list is the long lists of the left end right expression, to begin with.

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

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

We add two instruction to the long list. The first is an if-goto instruction that we add to the true set. If the expression is true, this instruction will jump to the target that later will be backpatched into the instruction. In the same way, we add a goto instruction to the false set. If the expression is false, the instruction will jump to the target later backpatched to the instruction.

ISet<MiddleCode> trueSet = new HashSet<MiddleCode>(),

falseSet = new HashSet<MiddleCode>();

trueSet.Add(AddMiddleCode(longList, middleOp, null,

leftExpression.Symbol,

rightExpression.Symbol));

falseSet.Add(AddMiddleCode(longList, MiddleOperator.Goto));

The final symbol holds logical type with true and false set.

Symbol symbol = new Symbol(trueSet, falseSet);

return (new Expression(symbol, shortList, longList));

}

### Addition and Subtraction Expression

MiddleCodeGenerator.cs

public static Expression AdditionExpression(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

Type leftType = leftExpression.Symbol.Type,

rightType = rightExpression.Symbol.Type;

Similar to the other cases above, the return the possible constant expression.

Expression constantExpression =

ConstantExpression.Arithmetic(middleOp, leftExpression,

rightExpression);

if (constantExpression != null) {

return constantExpression;

}

In the addition case, we also check if the expression is a static expression, in which case we return the expression.

Expression staticExpression =

StaticExpression.Binary(MiddleOperator.BinarySubtract,

leftExpression, rightExpression);

if (staticExpression != null) {

return staticExpression;

}

The final short list is simple the short lists of the left and right expression.

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

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

If at least on the expressions has pointer or array type, we call the PointerArithmetic methods that handles pointer arithmetic.

if (leftType.IsPointerOrArray()) {

return PointerArithmetics(middleOp, leftExpression, rightExpression);

}

In the right expression has pointer type, the operator must be addition, not subtraction. Note that we swap the expressions in the call to PointerArithmetic, so that the pointer is the first expression. XXX

else if (rightType.IsPointerOrArray()) {

Assert.Error(middleOp == MiddleOperator.BinaryAdd, middleOp,

Message.Invalid\_types\_in\_subtraction\_expression);

return PointerArithmetics(middleOp, rightExpression, leftExpression);

}

If none of the left or right expressions are pointers, we perform an arithmetic operation. First, we make sure that both expressions have arithmetic types (integral or floating).

else {

Assert.Error(leftExpression.Symbol.Type.IsArithmetic(),

leftExpression, Message.Non\_\_arithmetic\_expression);

Assert.Error(rightExpression.Symbol.Type.IsArithmetic(),

rightExpression, Message.Non\_\_arithmetic\_expression);

The we find the maximal type and cast both expressions to the type. The resulting expression does also have the maximal type.

Type maxType = TypeCast.MaxType(leftType, rightType);

leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);

rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);

Symbol resultSymbol = new Symbol(maxType);

The final long list is the long lists of the two expressions, and the operation (addition or subtraction).

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

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

AddMiddleCode(longList, middleOp, resultSymbol,

leftExpression.Symbol, rightExpression.Symbol);

return (new Expression(resultSymbol, shortList, longList));

}

}

The PointerArithmetic methods performs pointer arithmetic. We have two cases: addition of a pointer and an integral value, and subtraction of two pointer values.

private static Expression PointerArithmetic(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

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

shortList.AddRange(leftExpression.LongList);

shortList.AddRange(rightExpression.LongList);

Type leftType = leftExpression.Symbol.Type,

rightType = rightExpression.Symbol.Type;

If the left type is pointer and the right type is integral, we add an integer values to and pointer. What makes this different compared to regular addition is that we need take the size of the pointer type into consideration.

if (leftType.IsPointerOrArray() && rightType.IsIntegral()) {

We make sure that the left pointer type is not void, since pointer arithmetic is not allowed on void pointers.

Assert.Error(!leftType.PointerOrArrayType.IsVoid(),

leftExpression, Message.Pointer\_to\_void);

We type cast the integral right expression to the left pointer type for the arithmetic to be performed on values of the same size.

rightExpression = TypeCast.ImplicitCast(rightExpression, leftType);

The final long list is the long lists of the left and right expression, and the pointer arithmetic.

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

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

The result symbol has the same type as the pointer expression.

Symbol resultSymbol =

new Symbol(new Type(leftType.PointerOrArrayType));

The pointer arithmetic is performed in two steps. First, we multiply the integral value with the pointer size, and then we add the product to the

int pointerSize = leftType.PointerOrArrayType.Size();

Symbol multSymbol = new Symbol(Type.PointerTypeX),

sizeSymbol = new Symbol(Type.PointerTypeX,

(BigInteger) pointerSize);

We multiply the integral value with the size of the pointer type and add the product to the pointer value. The multSymbol value be the result of two constant values, in case of a constant integral value, or the integral value times one, in case of a pointer size of one, which is inefficient. However, the Middle Code Optimizer in Chapter 10 will take care of those cases.

AddMiddleCode(longList, MiddleOperator.UnsignedMultiply, multSymbol,

rightExpression.Symbol, sizeSymbol);

AddMiddleCode(longList, middleOp, resultSymbol,

leftExpression.Symbol, multSymbol);

return (new Expression(resultSymbol, shortList, longList));

}

If it is not the case of a pointer and an integral value, then there must be two pointer values. We make sure that none of them pointes at void. Moreover, we also make sure that their pointer types have the same size.

else {

Assert.Error(!leftType.PointerOrArrayType.IsVoid(),

leftExpression, Message.Pointer\_to\_void);

Assert.Error(!rightType.PointerOrArrayType.IsVoid(),

rightExpression, Message.Pointer\_to\_void);

Assert.Error(leftType.PointerOrArrayType.Size() ==

rightType.PointerOrArrayType.Size(),

leftType + " and " + rightType,

Message.Invalid\_expression);

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

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

Symbol resultSymbol = new Symbol(Type.VoidPointerType);

int pointerSize = rightType.PointerOrArrayType.Size();

Symbol additionSymbol = new Symbol(Type.IntegerPointerType),

sizeSymbol = new Symbol(Type.IntegerPointerType,

(BigInteger) pointerSize);

We first subtract the pointer values, and then divide the difference by the size of the pointer types. There may be a division by one, in case the pointer types have size one, which is inefficient. However, the Middle Code Optimizer in Chapter 10 will take care of the case.

AddMiddleCode(longList, middleOp, additionSymbol,

leftExpression.Symbol, rightExpression.Symbol);

AddMiddleCode(longList, MiddleOperator.UnsignedDivide, resultSymbol,

additionSymbol, sizeSymbol);

Expression resultExpression = new Expression(resultSymbol,

shortList, longList);

Finally, we type cast the result of the division into signed integer.

return TypeCast.ImplicitCast(resultExpression,

Type.SignedIntegerType);

}

}

### Multiplication Expressions

MiddleCodeGenerator.cs

public static Expression MultiplyExpression(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

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

constLongList.AddRange(leftExpression.LongList);

constLongList.AddRange(rightExpression.LongList);

Expression constantExpression =

ConstantExpression.Arithmetic(middleOp, leftExpression,

rightExpression);

if (constantExpression != null) {

return constantExpression;

}

Type maxType = TypeCast.MaxType(leftExpression.Symbol.Type,

rightExpression.Symbol.Type);

if ((middleOp == MiddleOperator.SignedModulo) ||

(middleOp == MiddleOperator.UnsignedModulo)) {

Assert.Error(maxType.IsIntegralPointerArrayStringOrFunction(),

maxType, Message.Invalid\_type\_in\_expression);

}

else {

Assert.Error(maxType.IsArithmeticPointerArrayStringOrFunction(),

maxType, Message.Invalid\_type\_in\_expression);

}

leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);

rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);

Symbol resultSymbol = new Symbol(maxType);

if (maxType.IsUnsigned()) {

string name = Enum.GetName(typeof(MiddleOperator), middleOp);

middleOp = (MiddleOperator) Enum.Parse(typeof(MiddleOperator),

name.Replace("Signed", "Unsigned"));

}

List<MiddleCode> shortList = new List<MiddleCode>(),

longList = new List<MiddleCode>();

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

AddMiddleCode(longList, middleOp, resultSymbol,

leftExpression.Symbol, rightExpression.Symbol);

return (new Expression(resultSymbol, shortList, longList));

}

### Cast Expressions

The specifier parameter holds the resulting type of the declaration specifier list.

MiddleCodeGenerator.cs

public static Type TypeName(Specifier specifier, Declarator declarator) {

Type specifierType = specifier.Type;

If the declarator is not null, we add the specifier’s type to the declarator, and returns its type.

if (declarator != null) {

declarator.Add(specifierType);

return declarator.Type;

}

If the declarator is null, we just return the specifer’s type.

else {

return specifierType;

}

}

### Prefix Expression

### Unary Addition Expressions

MiddleCodeGenerator.cs

public static Expression UnaryExpression(MiddleOperator middleOp,

Expression expression) {

Type type = expression.Symbol.Type;

Assert.Error(type.IsLogical() ||

type.IsArithmeticPointerArrayStringOrFunction(),

expression, Message.Non\_\_arithmetic\_expression);

Expression constantExpression =

ConstantExpression.Arithmetic(middleOp, expression);

if (constantExpression != null) {

return constantExpression;

}

Symbol resultSymbol = new Symbol(expression.Symbol.Type);

AddMiddleCode(expression.LongList, middleOp,

resultSymbol, expression.Symbol);

return (new Expression(resultSymbol, expression.ShortList,

expression.LongList));

}

### Logical Not Expression

MiddleCodeGenerator.cs

public static Expression LogicalNotExpression(Expression expression) {

Expression constantExpression =

ConstantExpression.LogicalNot(expression);

if (constantExpression != null) {

return constantExpression;

}

The resulting expression is the original expression with swapped true and false sets.

expression = TypeCast.ToLogical(expression);

Symbol notSymbol =

new Symbol(expression.Symbol.FalseSet, expression.Symbol.TrueSet);

return (new Expression(notSymbol, expression.ShortList,

expression.LongList));

}

### Bitwise Not Expression

MiddleCodeGenerator.cs

public static Expression BitwiseNotExpression(Expression expression) {

expression = TypeCast.LogicalToIntegral(expression);

Assert.Error(expression.Symbol.Type.IsIntegralPointerArrayOrFunction(),

Message.Only\_integral\_values\_for\_bitwise\_not);

Expression constantExpression =

ConstantExpression.Arithmetic(MiddleOperator.BitwiseNot, expression);

if (constantExpression != null) {

return constantExpression;

}

expression = TypeCast.LogicalToIntegral(expression);

Symbol resultSymbol = new Symbol(expression.Symbol.Type);

AddMiddleCode(expression.LongList, MiddleOperator.BitwiseNot,

resultSymbol, expression.Symbol);

return (new Expression(resultSymbol, expression.ShortList,

expression.LongList));

}

### The sizeof Expression

MiddleCodeGenerator.cs

public static Expression SizeOfExpression(Expression expression) {

We check that the storage of the expression is not register, since it is not allowed.

Assert.Error(!expression.Symbol.IsRegister(), expression,

Message.Register\_storage\_not\_allowed\_in\_sizof\_expression);

Moreover, we check that it is not a function or a bitfield, since they do not have sizes.

Type type = expression.Symbol.Type;

Assert.Error(!type.IsFunction(),

Message.Sizeof\_applied\_to\_function\_not\_allowed);

Assert.Error(!type.IsBitfield(),

Message.Sizeof\_applied\_to\_bitfield\_not\_allowed);

We create a symbol of signed integer types with the size as its value.

Symbol symbol = new Symbol(Type.SignedIntegerType,

(BigInteger) (expression.Symbol.Type.Size()));

symbol.StaticSymbol =

ConstantExpression.Value(symbol.UniqueName, Type.SignedIntegerType,

(BigInteger) (expression.Symbol.Type.Size()));

SymbolTable.StaticSet.Add(symbol.StaticSymbol);

return (new Expression(symbol, new List<MiddleCode>(),

new List<MiddleCode>()));

}

In the case of the type name, we also check that the expression is not a function or a bitfield.

MiddleCodeGenerator.cs

public static Expression SizeOfType(Type type) {

Assert.Error(!type.IsFunction(),

Message.Sizeof\_applied\_to\_function\_not\_allowed);

Assert.Error(!type.IsBitfield(),

Message.Sizeof\_applied\_to\_bitfield\_not\_allowed);

Symbol symbol =

new Symbol(Type.SignedIntegerType, (BigInteger) type.Size());

return (new Expression(symbol, new List<MiddleCode>(),

new List<MiddleCode>()));

}

### Address Expression

MiddleCodeGenerator.cs

public static Expression AddressExpression(Expression expression) {

Assert.Error(!symbol.IsRegister() && !symbol.Type.IsBitfield(),

expression, Message.Not\_addressable);

Assert.Error(!expression.Symbol.IsRegister(), expression,

Message.Invalid\_address\_of\_register\_storage);

The address operator may result in a static address.

Expression staticExpression =

StaticExpression.Unary(MiddleOperator.Address, expression);

if (staticExpression!= null) {

return staticExpression ;

}

If the expression has floating type, we need to pop the value from the floating value stack.

if (expression.Symbol.Type.IsFloating()) {

AddMiddleCode(expression.LongList, MiddleOperator.PopFloat);

}

The type of the resulting expression is a pointer to the type of the original expression

Type pointerType = new Type(expression.Symbol.Type);

Symbol resultSymbol = new Symbol(pointerType);

AddMiddleCode(expression.LongList, MiddleOperator.Address,

resultSymbol, expression.Symbol);

return (new Expression(resultSymbol, expression.ShortList,

expression.LongList));

}

### Dereference Expression

The dereference operator applies to pointers, arrays, strings, and functions. In the function case, the expression is regarded as a pointer to a function.

MiddleCodeGenerator.cs

public static Expression DereferenceExpression(Expression expression) {

Assert.Error(expression.Symbol.Type.IsPointerArrayStringOrFunction(),

Message.Invalid\_dereference\_of\_non\_\_pointer);

Symbol resultSymbol =

new Symbol(expression.Symbol.Type.PointerOrArrayType);

return Dereference(expression, resultSymbol, 0);

}

### Prefix Increment and Decrement Expression

MiddleCodeGenerator.cs

private static IDictionary<MiddleOperator,MiddleOperator> m\_incrementMap =

new Dictionary<MiddleOperator, MiddleOperator>();

private static IDictionary<MiddleOperator,MiddleOperator>

m\_incrementInverseMap = new Dictionary<MiddleOperator,MiddleOperator>();

static MiddleCodeGenerator() {

m\_incrementMap.Add(MiddleOperator.Increment,

MiddleOperator.BinaryAdd);

m\_incrementMap.Add(MiddleOperator.Decrement,

MiddleOperator.BinarySubtract);

m\_incrementInverseMap.Add(MiddleOperator.Increment,

MiddleOperator.BinarySubtract);

m\_incrementInverseMap.Add(MiddleOperator.Decrement,

MiddleOperator.BinaryAdd);

}

public static Expression PrefixIncrementExpression

(MiddleOperator middleOp, Expression expression){

Symbol symbol = expression.Symbol;

Assert.Error(symbol.Assignable, Message.Not\_assignable);

Assert.Error(symbol.Type.IsArithmeticOrPointer(),

expression, Message.Invalid\_type\_in\_increment\_expression);

if (symbol.Type.IsIntegralOrPointer()) {

AddMiddleCode(expression.ShortList, middleOp, null, symbol);

AddMiddleCode(expression.LongList, middleOp, null, symbol);

BigInteger? bitFieldMask = symbol.Type.GetBitfieldMask();

if (bitFieldMask != null) {

Symbol maskSymbol = new Symbol(symbol.Type, bitFieldMask.Value);

MiddleCode maskCode = new MiddleCode(MiddleOperator.BitwiseAnd,

symbol, symbol, maskSymbol);

expression.ShortList.Add(maskCode);

expression.LongList.Add(maskCode);

}

Symbol resultSymbol = new Symbol(symbol.Type);

AddMiddleCode(expression.LongList, MiddleOperator.Assign,

resultSymbol, symbol);

return (new Expression(resultSymbol, expression.ShortList,

expression.LongList));

}

The increment and decrement operator apply not only to integral values, but also to floating values. The operation. We start by pushing the value one at the floating value stack, the value to be incremented or decremented has already been pushed at the stack by earlier operations. We perform the operation, which is addition or subtraction. We preform the operation on both the short list and list of the expression. The difference is that in the short list case we pop the value off the stack since we do not need it anymore. In the long list case we do nothing, we just let the value stay on the stack to be used by later operations.

else {

AddMiddleCode(expression.ShortList, MiddleOperator.PushOne);

Symbol oneSymbol = new Symbol(symbol.Type, (decimal) 1);

AddMiddleCode(expression.ShortList, m\_incrementMap[middleOp],

symbol, symbol, oneSymbol);

AddMiddleCode(expression.ShortList, MiddleOperator.PopFloat, symbol);

AddMiddleCode(expression.LongList, MiddleOperator.PushOne);

AddMiddleCode(expression.LongList, m\_incrementMap[middleOp],

symbol, symbol, oneSymbol);

AddMiddleCode(expression.LongList, MiddleOperator.TopFloat, symbol);

Symbol resultSymbol = new Symbol(symbol.Type);

return (new Expression(resultSymbol, expression.ShortList,

expression.LongList));

}

}

### Postfix Increment and Decrement Expression

MiddleCodeGenerator.cs

public static Expression PostfixIncrementExpression

(MiddleOperator middleOp, Expression expression) {

Symbol symbol = expression.Symbol;

Assert.Error(symbol.Assignable, Message.Not\_assignable);

Assert.Error(symbol.Type.IsArithmeticOrPointer(),

expression, Message.Invalid\_type\_in\_increment\_expression);

if (symbol.Type.IsIntegralOrPointer()) {

Symbol resultSymbol = new Symbol(symbol.Type);

AddMiddleCode(expression.LongList, MiddleOperator.Assign,

resultSymbol, symbol);

AddMiddleCode(expression.ShortList, middleOp, null, symbol);

AddMiddleCode(expression.LongList, middleOp, null, symbol);

BigInteger? bitFieldMask = symbol.Type.GetBitfieldMask();

if (bitFieldMask != null) {

Symbol maskSymbol = new Symbol(symbol.Type, bitFieldMask.Value);

AddMiddleCode(expression.ShortList, MiddleOperator.BitwiseAnd,

symbol, symbol, maskSymbol);

AddMiddleCode(expression.LongList, MiddleOperator.BitwiseAnd,

symbol, symbol, maskSymbol);

}

return (new Expression(resultSymbol, expression.ShortList,

expression.LongList));

}

else {

AddMiddleCode(expression.ShortList, MiddleOperator.PushOne);

Symbol oneSymbol = new Symbol(symbol.Type, (decimal) 1);

AddMiddleCode(expression.ShortList, m\_incrementMap[middleOp],

symbol, symbol, oneSymbol);

AddMiddleCode(expression.ShortList, MiddleOperator.PopFloat, symbol);

In the long list case, the situation becomes a little more complicated, since the result of the operations shall be the original value, not the resulting value. Therefore, we must perform the inverse operation on the value on the stack in order to return to the original value.

AddMiddleCode(expression.LongList, MiddleOperator.PushOne);

AddMiddleCode(expression.LongList, m\_incrementMap[middleOp],

symbol, symbol, oneSymbol);

AddMiddleCode(expression.ShortList, MiddleOperator.TopFloat, symbol);

AddMiddleCode(expression.LongList, MiddleOperator.PushOne);

AddMiddleCode(expression.LongList, m\_incrementInverseMap[middleOp],

symbol, symbol, oneSymbol);

Symbol resultSymbol = new Symbol(symbol.Type);

return (new Expression(resultSymbol, expression.ShortList,

expression.LongList));

}

}

### Arrow Expression

MiddleCodeGenerator.cs

public static Expression ArrowExpression(Expression expression,

string memberName) {

Assert.Error(expression.Symbol.Type.IsPointer() &&

expression.Symbol.Type.PointerType.IsStructOrUnion(),

expression,

Message.Not\_a\_pointer\_to\_a\_struct\_or\_union\_in\_arrow\_expression);

Symbol memberSymbol =

expression.Symbol.Type.PointerType.MemberMap[memberName];

Assert.Error(memberSymbol != null, memberName,

Message.Unknown\_member\_in\_arrow\_expression);

Symbol resultSymbol = new Symbol(memberSymbol.Type);

return Dereference(expression, resultSymbol, memberSymbol.Offset);

}

### Index Expression

MiddleCodeGenerator.cs

public static Expression IndexExpression(Expression leftExpression,

Expression rightExpression) {

Expression staticExpression =

StaticExpression.Binary(MiddleOperator.Index, leftExpression,

rightExpression);

if (staticExpression != null) {

return staticExpression;

}

Expression arrayExpression, indexExpression;

if (leftExpression.Symbol.Type.IsPointerOrArray()) {

arrayExpression = leftExpression;

indexExpression = rightExpression;

}

else {

indexExpression = leftExpression;

arrayExpression = rightExpression;

}

Type arrayType = arrayExpression.Symbol.Type,

indexType = indexExpression.Symbol.Type;

Assert.Error(arrayType.IsPointerOrArray() &&

!arrayType.PointerOrArrayType.IsVoid(),

arrayType, Message.Invalid\_type\_in\_index\_expression);

Assert.Error(indexType.IsIntegral(), indexExpression,

Message.Invalid\_type\_in\_index\_expression);

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

shortList.AddRange(arrayExpression.ShortList);

shortList.AddRange(indexExpression.ShortList);

Symbol resultSymbol =

new Symbol(arrayExpression.Symbol.Type.PointerOrArrayType);

if (indexExpression.Symbol.Value is BigInteger) {

int indexValue = (int) ((BigInteger)indexExpression.Symbol.Value),

indexSize = arrayExpression.Symbol.Type.PointerOrArrayType.Size();

return Dereference(arrayExpression, resultSymbol,

indexValue \* indexSize);

}

else {

indexExpression =

TypeCast.ImplicitCast(indexExpression, arrayExpression.Symbol.Type);

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

longList.AddRange(arrayExpression.LongList);

longList.AddRange(indexExpression.LongList);

Symbol arraySymbol = arrayExpression.Symbol,

indexSymbol = indexExpression.Symbol;

Symbol sizeSymbol = new Symbol(indexSymbol.Type, (BigInteger)

arraySymbol.Type.PointerOrArrayType.Size()),

multSymbol = new Symbol(arraySymbol.Type);

AddMiddleCode(longList, MiddleOperator.UnsignedMultiply,

multSymbol, indexSymbol, sizeSymbol);

Symbol addSymbol = new Symbol(arraySymbol.Type);

AddMiddleCode(longList, MiddleOperator.BinaryAdd,

addSymbol, arraySymbol, multSymbol);

Expression addExpression =

new Expression(addSymbol, shortList, longList);

return Dereference(addExpression, resultSymbol, 0);

}

}

private static Expression Dereference(Expression expression,

Symbol resultSymbol, int offset) {

resultSymbol.AddressSymbol = expression.Symbol;

resultSymbol.AddressOffset = offset;

AddMiddleCode(expression.LongList, MiddleOperator.Dereference,

resultSymbol, expression.Symbol, 0);

If the resulting expression holds floating type, we need to pop the value from the floating value stack.

if (resultSymbol.Type.IsFloating()) {

AddMiddleCode(expression.LongList, MiddleOperator.PushFloat,

resultSymbol);

}

return (new Expression(resultSymbol, expression.ShortList,

expression.LongList));

}

### Dot Expression

MiddleCodeGenerator.cs

public static Expression DotExpression(Expression expression,

string memberName) {

Symbol parentSymbol = expression.Symbol;

Assert.Error(parentSymbol.Type.IsStructOrUnion(), expression,

Message.Not\_a\_struct\_or\_union\_in\_dot\_expression);

Assert.Error(parentSymbol.Type.MemberMap != null, expression,

Message.Member\_access\_of\_uncomplete\_struct\_or\_union);

Symbol memberSymbol;

Assert.Error(parentSymbol.Type.MemberMap.TryGetValue(memberName,

out memberSymbol), memberName,

Message.Unknown\_member\_in\_dot\_expression);

Symbol resultSymbol;

if (parentSymbol.AddressSymbol != null) {

string name = parentSymbol.Name + "." + memberSymbol.Name +

Symbol.SeparatorId + memberSymbol.Offset;

resultSymbol = new Symbol(name, parentSymbol.ExternalLinkage,

parentSymbol.Storage, memberSymbol.Type,

parentSymbol.IsParameter());

resultSymbol.UniqueName = parentSymbol.UniqueName;

resultSymbol.AddressSymbol = parentSymbol.AddressSymbol;

resultSymbol.AddressOffset = parentSymbol.AddressOffset;

resultSymbol.Offset = parentSymbol.Offset + memberSymbol.Offset;

}

else {

resultSymbol = new Symbol(memberSymbol.Type);

resultSymbol.Name = parentSymbol.Name + Symbol.SeparatorId +

memberName;

resultSymbol.UniqueName = parentSymbol.UniqueName;

resultSymbol.Storage = parentSymbol.Storage;

resultSymbol.Offset = parentSymbol.Offset + memberSymbol.Offset;

}

return (new Expression(resultSymbol, expression.ShortList,

expression.LongList));

}

### Function Call Expression

The FunctionPreCall method is called before the argument list is parsed. It checks that the expression is either a function or a pointer to a function, adds the parameter type list of the function to the type list stack, adds zero to the current offset stack, and adds the call header instruction to the short and long list of the expression.

MiddleCodeGenerator.cs

public static void FunctionPreCall(Expression expression) {

Type type = expression.Symbol.Type;

Assert.Error((type.IsPointer() && type.PointerType.IsFunction()) ||

type.IsFunction(), expression.Symbol,

Message.Not\_a\_function);

Type functionType = type.IsFunction() ? type : type.PointerType;

TypeListStack.Push(functionType.TypeList);

ParameterOffsetStack.Push(0);

AddMiddleCode(expression.ShortList, MiddleOperator.FunctionPreCall,

SymbolTable.CurrentTable.CurrentOffset);

AddMiddleCode(expression.LongList, MiddleOperator.FunctionPreCall,

SymbolTable.CurrentTable.CurrentOffset);

}

The CallExpression method is called after the argument list is parsed. It checks that the expression is either a function or a pointer to a function, adds the parameter type list of the function to the type list stack, adds zero to the current offset stack, and adds the call header instruction to the short and long list of the expression.

public static Expression CallExpression(Expression functionExpression,

List<Expression> argumentList){

List<Type> typeList = TypeListStack.Pop();

ParameterOffsetStack.Pop();

The reason we keep track of the current offset is that we in case of nested function calls need to allocate space for the previous arguments of the functions call outside the nested function. If the current offset is more than zero, we add it together with the size of the standard function top the total offset. If the current offset is zero, no argument has yet been passed, and we do not need to increase the total offset.

int totalOffset = 0;

foreach (int currentOffset in ParameterOffsetStack) {

if (currentOffset > 0) {

totalOffset += (SymbolTable.FunctionHeaderSize + currentOffset);

}

}

Type functionType = functionExpression.Symbol.Type.IsPointer() ?

functionExpression.Symbol.Type.PointerType :

functionExpression.Symbol.Type;

We check that there are not too few parameters in the call. If the type list is null, the number of arguments does not matter. However, if the type list is not null, the number of arguments must be at least the number of parameters in the type list.

Assert.Error((typeList == null) ||

(argumentList.Count >= typeList.Count),

functionExpression,

Message.Too\_few\_actual\_parameters\_in\_function\_call);

We also check that there are not too many parameters in the call. If the type list is null or if the callee function has an elliptic parameter list, the number of arguments does not matter. There may be more arguments than parameters, we already know that the arguments are not fewer than the parameters. However, if the type list is not null and the function does not have an elliptic parameter list, the number of arguments must equal the number of parameters.

Assert.Error(functionType.IsEllipse() || (typeList == null) ||

(argumentList.Count == typeList.Count),

functionExpression,

Message.Too\_many\_parameters\_in\_function\_call);

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

longList.AddRange(functionExpression.LongList);

When iterate through the, we keep track of the extra size. In elliptic function calls we need to allocate memory for the extra parameters that is not included in the regular parameter list.

int count = 0, offset = SymbolTable.FunctionHeaderSize, extra = 0;

foreach (Expression argumentExpression in argumentList) {

longList.AddRange(argumentExpression.LongList);

Type type;

if ((typeList != null) && (count++ < typeList.Count)) {

type = typeList[count];

}

else {

type = ParameterType(argumentExpression.Symbol);

extra += type.Size();

}

We add the argument to the parameter list and update the current offset.

if (type.IsStructOrUnion()) {

AddMiddleCode(longList, MiddleOperator.ParameterInitSize,

SymbolTable.CurrentTable.CurrentOffset + totalOffset +

offset, type, argumentExpression.Symbol);

}

AddMiddleCode(longList, MiddleOperator.Parameter,

SymbolTable.CurrentTable. CurrentOffset + totalOffset +

offset, type, argumentExpression.Symbol);

offset += type.Size();

}

We add both the function call and post call instructions to the long list. The post call instruction is for

Symbol functionSymbol = functionExpression.Symbol;

AddMiddleCode(longList, MiddleOperator.Call,

SymbolTable.CurrentTable.CurrentOffset + totalOffset,

functionSymbol, extra);

AddMiddleCode(longList, MiddleOperator.PostCall,

SymbolTable.CurrentTable.CurrentOffset + totalOffset);

Type returnType = functionType.ReturnType;

Symbol returnSymbol = new Symbol(returnType);

We make the short list be a copy of the long list. However, there is one difference between the lists if the function returns a floating value. In that case, we pop the value off the floating value stack in the short list, since the value is not interesting in the short list, only the side effect of the function call. However, we keep the value on the stack in the long list, since the value shall be used in the long list.

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

shortList.AddRange(longList);

if (!returnType.IsVoid()) {

if (returnType.IsStructOrUnion()) {

Type pointerType = new Type(returnType);

Symbol addressSymbol = new Symbol(pointerType);

returnSymbol.AddressSymbol = addressSymbol;

}

AddMiddleCode(longList, MiddleOperator.GetReturnValue,

returnSymbol);

if (returnType.IsFloating()) {

AddMiddleCode(shortList, MiddleOperator.PopEmpty);

}

}

return (new Expression(returnSymbol, shortList, longList));

}

### Argument Expression List

The ArgumentExpression method type casts the argument into an appropriate type.

MiddleCodeGenerator.cs

public static Expression ArgumentExpression(int index,

Expression expression) {

We need to type list on top of the type list stack to compare the argument expression with the parameter type.

List<Type> typeList = TypeListStack.Peek();

If the argument is within the parameter list, we type cast the argument into the parameter type.

if ((typeList != null) && (index < typeList.Count)) {

expression = TypeCast.ImplicitCast(expression, typeList[index]);

}

else {

If the argument is not within the parameter list (elliptic call), we type cast the argument into an appropriate type: character and short integer are cast into (signed or unsigned) integer, and float is cast into double. These type casts apply to the elliptic function call rules of ANSI C.

Type type = expression.Symbol.Type;

if (type.IsChar() || type.IsShort()) {

if (type.IsSigned()) {

expression =

TypeCast.ImplicitCast(expression, Type.SignedIntegerType);

}

else {

expression =

TypeCast.ImplicitCast(expression, Type.UnsignedIntegerType);

}

}

else if (type.IsFloat()) {

expression = TypeCast.ImplicitCast(expression, Type.DoubleType);

}

}

We need to update current offset to allocate space in case of nested function calls.

int offset = ParameterOffsetStack.Pop();

ParameterOffsetStack.Push(offset +

ParameterType(expression.Symbol).Size());

return (new Expression(expression.Symbol, expression.LongList,

expression.LongList));

}

The ParameterType method returns the type of the parameter. In case of array, string, or function, it returns pointer types. Otherwise, it returns the regular type.

private static Type ParameterType(Symbol symbol) {

switch (symbol.Type.Sort) {

case Sort.Array:

return (new Type(symbol.Type.ArrayType));

case Sort.Function:

return (new Type(symbol.Type));

case Sort.String:

return (new Type(new Type(Sort.SignedChar)));

default:

return symbol.Type;

}

}

### Primary Expressions

The value holds floating type, we push it at the floating value stack.

MiddleCodeGenerator.cs

public static Expression ValueExpression(Symbol symbol) {

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

if (symbol.Type.IsFloating()) {

AddMiddleCode(longList, MiddleOperator.PushFloat, symbol);

}

return (new Expression(symbol, new List<MiddleCode>(), longList));

}

Given a name, we look it up in the current symbol table. If there is no symbol, we add a function without a parameter list, that returns a signed integer, to the symbol table.

MiddleCodeGenerator.cs

public static Expression NameExpression(string name) {

Symbol symbol = SymbolTable.CurrentTable.LookupSymbol(name);

Assert.Error(symbol != null, name, Message.Unknown\_name);

List<MiddleCode> shortList = new List<MiddleCode>(),

longList = new List<MiddleCode>();

if (symbol.Type.IsFloating()) {

AddMiddleCode(shortList, MiddleOperator.PushFloat, symbol);

AddMiddleCode(longList, MiddleOperator.PushFloat, symbol);

}

return (new Expression(symbol, shortList, longList));

}

The registers are only used internally, in conjunction with system calls. On some occasion the interrupt call returns information stored in a register.

The closed statements do also include a set of internal statements; that is, statements not included in standard C, but necessary for the standard library functionality. The first statement is the load register statement, which stores a value in a register. The operands are the name of a register and an expression of integral or pointer type with same size as the register.

MiddleCodeGenerator.cs

public static Expression RegisterExpression(Register register) {

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

int size = AssemblyCode.SizeOfRegister(register);

Type type = TypeSize.SizeToUnsignedType(size);

Symbol symbol = new Symbol(type);

AddMiddleCode(longList, MiddleOperator.InspectRegister,

symbol, register);

return (new Expression(symbol, new List<MiddleCode>(),

longList, register));

}

MiddleCodeGenerator.cs

public static Expression CarryFlagExpression() {

ISet<MiddleCode> trueSet = new HashSet<MiddleCode>(),

falseSet = new HashSet<MiddleCode>();

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

trueSet.Add(AddMiddleCode(longList, MiddleOperator.Carry));

falseSet.Add(AddMiddleCode(longList, MiddleOperator.Goto));

Symbol symbol = new Symbol(trueSet, falseSet);

return (new Expression(symbol, new List<MiddleCode>(), longList));

}

MiddleCodeGenerator.cs

public static Expression StackTopExpression() {

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

Symbol symbol = new Symbol(new Type(Type.UnsignedCharType));

AddMiddleCode(longList, MiddleOperator.StackTop, symbol);

return (new Expression(symbol, new List<MiddleCode>(), longList));

}