Function

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

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

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

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

Index

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

The index is constant, in which case we treat the operator as if it was the arrow operator.

The index is not constant and the array or pointer type size is one, in which case we generate that add the index to the pointer or array.

The index is not constant and the array or pointer type size is greater than, in which case we also need to generate code that multiply the index with the type size.

Dot

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

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

Increment

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

Default

The default statement have to comply with the following demands:

Main.m\_defaultStack must not be empty. If it is empty, the default statements misses a surrounding switch statements.

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

While

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

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

Header

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

Switch

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

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

Do

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

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

For

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

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

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

Label goto

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

Case

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.

Default

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

The GenerateReturnStatement method is surprisly 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 surrounding 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.

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 an 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));

}

public static Expression LogicalOrExpression(Expression leftExpression,

Expression rightExpression) {

We 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 expressions to the 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. If the left expression is evaluated to true, the right expression (including its side effects) shall not be 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));

}

There is a set of optimizations available:

We clear goto-next-statements: jumps to the next line.

We also modify goto-next-double-statements: an conditional jump instruction that jumps two steps ahead and is followed by an unconditional jump instruction.

We trace goto-chains: jump instructions that jump to other jump instructions.

We clear unreachable code: code that is not reachable from the first function instruction.

We remove empty code: code that has been cleared by the optimization above.

If the deep search does not found such a solution, we have to do something about the graph. In this book we remove one edge and try the deep search again. In this way we continue to remove edges until we have found a solution. The benefit of removing edges is that the likeliness of founding a solution increases for each removed edge. The drawback is that each removed edge means that two tracks have to share we one register and that we have to add store and load instruction for the code to work. When we need to remove an edge we sort the edges by counting the number of load and store instructions necessary to add as compensation for the removal of the edge. There is no guaranty that this method generates the optimal solution but since shortage of registers only occur on rare occasions it shall be good enough.

In this way, every call to DeepFirstSearch, including the first call in RegisterAllocator.

If a track holds the pointer property, it can be allocated to a reduced set or registers only.

The Generate method is called after the tracks have been assigned registers. We iterate through the entries and assign them the given register. Note that the entries may have different sizes, why we have to convert the register to the size of each entry.