Phase 4 : Intermediate-Code Generation

Team 7 : Jae Eun Kim, Eun Do Lee

1. Introduction

In this phase, we converted the abstract syntax tree from the parsing phase into three-address codes. The abstract syntax tree is two-dimensional. Converting it into three-address code form means transforming it into a linear code and making it one step closer to assembly code.

2. How to Translate Abstract Syntax Trees to Three-Address Codes

We implemented the ToTac functions in the AST nodes. The ToTac function returns CTacAddr*, which represents a variable in the three-address code. The function uses both synthesized and inherited attributes, which are the returned values of ToTac of the nodes corresponding to the attributes. With the appropriate operator and attributes, ToTac creates three-address codes that are relevant to each AST node and adds them to the code block in its argument. Three-address codes may be simple arithmetic operations, or branches that calls for a change in the program counter (PC). One such instruction is of operation type opGoto. Instead of attributes, the instruction is created with a label. The instruction tells the program to jump to the location in the code where the label is. Others are conditional branches that compare values of src1 and src2, and branch to a destination label if the comparison result matches that of the operation type. For example, three-address code of type opEqual would jump to its destination label if src1 and src2 are equal. The following are some specific examples of ToTac functions that we implemented.

2-1. Statement Sequence

Most statement nodes' ToTac functions are as simple as explained above. After creating and adding an instruction with their corresponding operator and attributes, they add an instruction with the operator opGoto and the label "next" that tells the program to jump to the next statement in the statement sequence. There were, however, slightly more complicated statements to implement because they required more labels and condition checks.

The ToTac function for the CAstStatIf node creates two labels, ifTrue and ifFalse. The condition is evaluated and the program jumps to ifTrue or ifFalse according to the condition. ToTac must generate code for both cases. It first adds the label ifTrue to the code block. Then it calls ToTac of the if body. In doing so, because it needs to tell the program to come back after executing the body, it makes a new label ifNext and passes it as a parameter to ToTac of the if body. Then it adds ifNext to the code block. After that it adds an instruction telling the program to jump to "next". Likewise, ToTac adds the label ifFalse to the code block and calls ToTac of the else body. It makes a new label elseNext, passes it as a parameter to ToTac of the else body, and adds it to the code block.

The ToTac function for the CAstStatWhile node creates two labels, conditionLabel and bodyLabel. ToTac adds conditionLabel to the code block and evaluates the condition. If the condition is true, the program jumps to bodyLabel, and if the condition is false, the program jumps to "next". The body of the while statement is implemented in a similar way to the if body or the else body of the if statement. A new label bodyNext is created, and passed as a parameter to ToTac of the body. After calling ToTac of the body, the label is added to the code block. After that, the instruction with operator opGoto and label conditionLabel is added to the code block, telling the program to jump back to the condition and evaluate it again.

2-2. Operations

We will only explain binary operations' ToTac functions because unary and special operations' ToTac functions are similar to them and much simpler.

The three-address code is generated differently depending on whether the operator is a relational operator or not. If the operator is not a relational operator, that is, if it is '+', '-', '*', or '/', then the implementation is much easier. We created a temporary variable dst of CIntType in the code block. The attributes are the return values of ToTac of the left and right operands. Then we added to the code block an instruction with arguments : (GetOperation() (this will return the operator), dst, attribute1, attribute2). Thus, a three-address code will be generated that states that dst will have the result of the operation with attribute1 and attribute2 as operands.

Now the case of the relational operators remains. We had to be careful because boolean expressions may be control flows. To implement this efficiently, another ToTac function that accepts the true and false labels as additional arguments is required. We will explain this function, ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse) first. First, consider the case when the operator is not opAnd or opOr. Attributes are the return values of ToTac of the left and right operands. Then we added two instructions to the code block, one for the true condition and the other for the false condition. Next, we implemented for the operator opAnd. This operator is special because if the left operand is false, the entire result is false, so the right operand does not have to be evaluated. An additional label is necessary. We created a new label called localLabel. If the left operand is false, the program jumps to lfalse. Otherwise it jumps to the localLabel. Then the program evaluates the right operand and jumps to ltrue or lfalse accordingly. opOr is just the opposite case of opAnd, so the implementation is quite the same. If the left operand is true, the entire result is true, so the right operand does not have to be evaluated. If the left operand is true, the program jumps to ltrue. Otherwise it jumps to localLabel, then the program evaluates the right operand and jumps to ltrue or lfalse accordingly.

If the boolean expression was not involved in control flow decision, we return back to ToTac(CCodeBlock *cb). ToTac(CCodeBlock *cb) creates the labels ltrue, lfalse and calls ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse) explained above. Then it creates and adds a temporary variable dst to the code block, but this time of type CBoolType. Then it creates a CAstDesignator node with dst's symbol. After that, two boolean CAstConstant nodes are created, one with true value and the other with false value. Two CAstStatAssign nodes are created with the designator node and one with the true constant node, the other with the false constant node. Then each CAstStatAssign node's ToTac function is called. dst is the return value of ToTac(*cb), like the integer operation case.

3. Conclusion

We tested all nine tests in the ir test folder and matrix.mod, and checked that our output three-address codes are the same as those of the reference code except the labels. The reference code's labels were shifted by one or two from ours. However, that means only the naming of the labels is different. The control flow is the same as ours. We concluded that our compiler produces accurate three-address codes.