Compilers Project 4 'Intermediate-Code Generation' Report

2013-11431 Hyunjin Jeong

The fourth term project was to create Intermediate-Representation(IR) codes. Especially, we had to implement the Three-Address Code(TAC) in ast.cpp. the TAC is assembly-like code which will be used in the next project, Code Generation. In this report, I will write how I implemented ToTac() functions of the classes in ast.cpp.

0. Changes from the base implementation.

Because of changes of the updated reference implementation, I modified some functions to check duplication of string constants/temporary variables and sub-array expressions.

In src/ir.cpp, there's a function, "CScope::CreateTemp()". It makes temporary variables named 't0', 't1', 't2', and so on. However, it doesn't check existence about identifiers defined by user which have same name with temporary variables. Therefore, I changed this function to check local/global symbol table to not make duplicated name.

CAstStringConstant class in src/ast.cpp had same problem. It makes global character arrays for strings with name '_str_1', '_str_2', and so on, but it doesn't check name duplication. So I checked global symbol table to not make duplicated name.

The updated reference implementation (on 5/26) doesn't support sub-array expressions (i.e. A[0] of A[1][2]). So I added dimension check conditions to CAstArrayDesignator::TypeCheck(). Now SnuPL/1 allows full-dimensional accesses only for arrays.

But these changes are not applied to phase 4 source code (they are commented out currently). I will apply them to phase 5 source code because I want to make the result as same as possible with the result of reference implementation of phase 4, '4_test_ir'.

1. CAstScope

1.1. ToTac(CCodeBlock *cb)

I copied the reference implementation in project pdf. For all statements, this function calls totac() of statements and makes labels to be executed in next instruction. Finally, it calls ‘CleanupControlFlow’ which cleans up useless labels, instructions.

2. CAstStatAssign

2.1. ToTac(CCodeBlock *cb, CTacLabel *next)

First it calls totac() of Right-Hand Side(RHS) and then Left-Hand Side(LHS). After that, it adds instruction to the code block, cb. The instruction assigns right to left. Finally, it adds new instruction to jump to next label.

3. CAstStatCall

3.1. ToTac(CCodeBlock *cb, CTacLabel *next)

All IR codes generation is implemented in CAstFunctionCall::ToTac(). Therefore, this function just calls totac() of _call, then set jump instruction to next label.

4. CAstStatReturn

4.1. ToTac(CCodeBlock *cb, CTacLabel *next)

If scope’s type is NULL (it means the scope is module or procedure), it creates new return instruction with nothing. Otherwise, it calls totac() of the expression first and creates new return instruction with return value of totac() of the expression. Finally, it sets jump instruction to next label.

5. CAstStatIf

5.1. ToTac(CCodeBlock *cb, CTacLabel *next)

First it creates two labels, if_true and if_false. Then it calls totac() of the condition expression with if_true and if_false. Then it calls totac() for all true-body statements. After that, it creates opGoto instruction to ‘next’ label. It makes codes jump to ‘next’ instruction in true case. Then this function repeats above sequence for false case. Finally, it returns NULL.

6. CAstStatWhile

6.1. ToTac(CCodeBlock *cb, CTacLabel *next)

First it creates two labels, while_cond and while_body. It sets 'while_cond' label, then calls totac() of the condition expression. Then it sets 'while_body' label, and calls totac() for all statements. Then it adds jump instruction to 'while_cond' label. This enables repeat for while statement. After that, it adds jump instruction to 'next' label, then returns NULL.

7. CAstBinaryOp

7.1. ToTac(CCodeBlock *cb)

In operator case '<', '<=', '>', '>=', '#', '=': First it creates labels 'ltrue', 'lfalse', 'end', and 'dummy'. Dummy label is useless but it is used to synchronize label number with the reference implementation. Then it calls totac() of LHS and RHS and their results are saved into 'left' and 'right'. After that, it adds an instruction with the operator, 'ltrue', 'left', and 'right'. Then it adds jump instruction to 'false' label, and it sets label 'ltrue'. Then it creates a temporary variable and assigns '1' to 'temp' in true case, or assigns '0' to 'temp' in false case. Finally, it returns 'temp'.

In operator case '&&': It creates lables 'ltrue', 'lfalse', 'end', and 'andtrue'. Then it calls totac() of LHS, and sets label 'andtrue'. Then it calls totac() of RHS and sets label 'ltrue'. Then it creates a temporary variable and assigns '1' to 'temp' if true or assigns '0' to temp if false. Finally it returns 'temp'. In operator case '||', the label 'andtrue' is changed to 'orfalse' but the process is similar to '&&' case.

In operator case '+', '-', '/', '*': This case is simple. First it calls totac() of LHS and RHS, then creates temporary variable 'temp'. Then it adds new instruction with the operator, 'temp', and the results of totac(). Finally it returns 'temp'.

7.2. ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

If the operator is '&&', it creates label 'andtrue' first. Then it calls totac() of LHS and set labels 'andtrue'. After that, it calls totac() of RHS. If the operator is '||', the label 'andtrue' is changed to 'orfalse' and the process is similar.

If the operator is '<' or '<=' or '>' or '>=' or '=' or '#', First it creates a dummy label. Then it calls totac() of LHS and RHS. Then it adds an operation with the operator, 'ltrue', the results of totac(). Finally it creates a jump instruction to 'lfalse'.

Other operator case is not called in this prototype. Therefore I set assert().

8. CAstUnaryOp

8.1. ToTac(CCodeBlock *cb)

If the operator is '+' or '-', it calls totac() of the operand. Then it creates 'temp', and adds new instruction with the operator, 'temp', and the result of totac(). It returns 'temp'.

If the operator is '!', first it creates labels 'ltrue', 'lfalse', and 'end'. Then it calls totac() of the operand and sets label 'ltrue'. Then it creates 'temp' and assigns '1' to 'temp' if true, or assigns '0' to 'temp' if false. It returns 'temp'.

8.2. ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

In this function, the only allowed operator is '!'. Therefore I set assert() first. It just calls totac() of the operand and returns NULL.

9. CAstSpecialOp

9.1. ToTac(CCodeBlock *cb)

First it calls totac() of the operand and saves the result to 'src'. Then it sets return type to the pointer of 'src' type. The sub-array types are converted into their base type. I don't know why but the reference implementation did it same way. This sub-array type conversion is not needed in future because the sub-array accessing is not allowed now. Finally, it creates 'temp' and adds an address instruction of 'src' to 'temp'. It returns 'temp'.

10. CAstFunctionCall

10.1. ToTac(CCodeBlock *cb)

For all arguments, this function calls totac() of the argument and add them to parameter in reverse order. After that, if the type is NULL (it means procedure/module), it adds a call instruction with no destination and returns NULL. Otherwise, it adds a call instruction with 'temp' and returns 'temp'.

10.2. ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

It just calls totac() of itself and adds new instruction to compare the value of totac(). Then it sets a label 'false'.

11. CAstDesignator

11.1. ToTac(CCodeBlock *cb)

It is really simple. It just returns CTacTemp with symbol.

11.2. ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

First it calls totac() of itself, and saves the result to 'res'. Then it adds a comparing instruction with 'res' and '1'. Then it sets label 'lfalse', and returns NULL.

12. CAstArrayDesignator

12.1. ToTac(CCodeBlock *cb)

There are two cases: pointer to array, array.

In array case, it first creates temp that address of array is saved in it. Then it gets first array index expression by calling its totac(). Then it gets the dimension number. Then it loops acoording to dimension number. In for loop, it first sets parameters for DIM(A: array, i: i-th dimension). Then it calls DIM and saves the result to temporary variable. Then it multiplies by previous array index expression which saved in 'res_tmp'. Then if an index exists, it calls totac() of an index expression to get new index expression and it adds new array index expression and saves the result to 'res_tmp'. Otherwise, it just adds '0' (This part is not need in future, because we don't support the sub-array access). After the for loop, it multiplies the result to the array element size (4 if integer, 1 if Boolean/character). Then it takes parameter for DOFS(A: array), and calls DOFS(A). Then it adds element offset to the data offset, and the address of the array. Finally, it returns 'CTacReference' of the result value.

The pointer to array case has same procedure with array case, but it doesn't create temporary variables to use the address of the array.

12.2. ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

It calls totac() of itself and saves the result to 'res'. Then it adds a comparing instruction to 'res' and '1'. Then it adds a jumping instruction to 'lfalse', and returns NULL.

13. CAstConstant

13.1. ToTac(CCodeBlock *cb)

It just returns the 'CTacConst' with symbol value.

13.2. ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

If the symbol value is 0, then it adds a jump instruction to 'lfalse'. Otherwise, it adds a jump instruction to 'ltrue'. It returns NULL.

14. CAstStringConstant

14.1. ToTac(CCodeBlock *cb)

It just returns the 'CTacName' with symbol.

14.2. ToTac(CCodeBlock *cb, CTacLabel *ltrue, CTacLabel *lfalse)

String cannot be Boolean. Therefore, I set assert here.