Phase 4: Intermediate-Code Generation

In this fourth phase of the project your task is to convert the abstract syntax tree into our intermediate representation in three-address code form.

You are advised to use the intermediate representation and its implementation provided with the project sources (provided in the tarball of the assignment), but of course you are free to implement your own version if you like. The following text assumes you are using the project IR (details of the IR can be found in the Appendix of this assignment).

To translate the AST into IR, you have to implement the (still empty) conversion routines in the AST (CAst*::ToTac). The translation uses both inherited and synthesized attributes as discussed in the lecture.

The translation follows the scheme outlined in the Dragon book chapter 6 (in particular, 6.2, 6.4, 6.6, and 6.7). Study the textbook carefully, then apply the described concepts to our AST. The provided IR contains classes for TAC representation: CTacInstr representing instructions, CTacAddr and subclasses representing different kinds of variables, and CTacLabel for labels. Blocks of code are managed by the CCodeBlock class. The CScope class and its subclasses CModule and CProcedure are containers representing the main code block and the code of subclasses. Methods to create labels and temporary values are also managed by the CScope class.

Examine the HTML documentation of the classes to get an idea about their roles, methods, and properties. As always, run

```
 $ make doc
```

in the snuplc/ directory to generate the documentation from the Doxygen comments in the code. In the following we provide information about the SnuPL/1 AST's specific translations.

Statements: statements are translated with an inherited attribute *next* that denotes where the control flow will continue after the current statement. The translation template for statements is thus

```
CTacAddr* CAstStatement::ToTac(CCodeBlock *cb, CTacLabel *next)
{
  // generate code for statement (assignment, if-else, etc.)

  // jump to next
  cb->AddInstr(new CTacInstr(opGoto, next));

  return NULL;
}
```

that is, a jump to the next label should be inserted after encoding the statement.

Expressions: translating expressions is a bit trickier because we have to distinguish between boolean and non-boolean expression evaluation. Non-boolean expressions can be translated in a straight-forward manner by simply emitting the operation with the correct operation and operands. Boolean expressions, however, have to be evaluated lazily, i.e., once the result of a boolean expression is known, the remaining expression must not be evaluated anymore. This lazy evaluation allows us to write statements such as

```
if ((divisor # 0) && (dividend / divisor > 5)) then …
```

because the second operand of the `&&` operator, `(dividend / divisor > 5)`, is only evaluated if the first operand, `(divisor # 0)`, is true.

As shown in the lecture, lazy evaluation of boolean expressions is implemented by translating the expression into a series of tests and GOTOs, so called "short-circuit code". The expression

```
a && b
```

can be translated as

```
  if a then goto test_b
  goto lbl_false
test_b:
  if b then goto lbl_true
  goto lbl_false
```

This example suggests that boolean expression evaluation needs two inherited attributes, `lbl_true` and `lbl_false`, denoting the targets to jump to when the condition evaluates to true or false, respectively. Indeed, the translation template for boolean expressions follows the template

```
CTacAddr* CAstExpression::ToTac(CCodeBlock *cb,
                            CTacLabel *ltrue, CTacLabel *lfalse)
{
  // generate jumping code for boolean expression

  return NULL;
}
```

Since in our AST we use the same class to represent scalar and boolean expressions, generating the jumping labels may be a bit tricky. Also, be aware that boolean expressions may contain subtrees with scalar expressions as demonstrated by the following expression

```
a && (w * 5 + 3 < foo(x, y, z))
```

Translation of Array Accesses: until now, array element accesses have been represented by the expressions for the individual dimensions. Now it is time to translate this abstract representation into an actual memory address.

SnuPL/1's implementation of arrays states

1. elements are ordered row-major
2. the first element of an array A is located at &A + DOFS(A)
3. the size of the *i*-th dimension can be queried by DIM(A, *i*). DIM(A, 0) returns the number of dimensions.

The address computation is then a simple modification of the array element address calculation formula (6.6) from the textbook.

The address computation code should be implemented in `CAstArrayDesignator::ToTac(CCodeBlock *cb)`. While one could generate the sequence of TAC instructions directly, we advise you to generate a higher-level representation in AST form and then call ToTac() on the AST computing the array element's address. This has the advantage that code for function calls and operations does not need to be duplicated.

As an example, consider the definition

```
var i: integer;
    A: integer[10][5];
```

and the source code

```
A[1][3] := I
```

The AST generated by the reference implementation for phase 3 is

```
:= <int>
  [ @A        <array 10  of <array 5  of <int>>> ] <int>
    1 <int>
    3 <int>
  [ @i        <int> ] <int>
```

and the TAC produced by the reference implementation for this phase produces

```
      0:     &()    t0 <- A                 # address of A (used in line 12)
      1:     param  1 <- 2                  # lines 1-4: get size of 2nd dimension
      2:     &()    t1 <- A                 # parameter 0: &A, parameter 1: 2
      3:     param  0 <- t1
      4:     call   t2 <- DIM
      5:     mul    t3 <- 1, t2             # multiply by first array index expression (1)
      6:     add    t4 <- t3, 3             # add second array index expression
      7:     mul    t5 <- t4, 4             # multiply by array element size
      8:     &()    t6 <- A                 # lines 8-10: get offset of data
      9:     param  0 <- t6
     10:     call   t7 <- DOFS              # call DOFS(A)
     11:     add    t8 <- t5, t7            # add element offset to data offset
     12:     add    t9 <- t0, t8            # add address of A
     13:     assign @t9 <- i                # assign value of i to *t9
```

You can find the code for this example in `test/tac/test5.mod`.

Using the IR: the provided IR is fully implemented and functional. You need to implement the `CAst*::ToTac()` methods in `ast.cpp`. To get you started, here is the implementation of `CAstScope::ToTac`

```
CTacAddr* CAstScope::ToTac(CCodeBlock *cb)
{
  assert(cb != NULL);

  CAstStatement *s = GetStatementSequence();
  while (s != NULL) {
    CTacLabel *next = cb->CreateLabel();
    s->ToTac(cb, next);
    cb->AddInstr(next);
    s = s->GetNext();
  }

  cb->CleanupControlFlow();

  return NULL;
}
```

Use the `CCodeBlock *cb` parameter's `AddInstr` method to insert instructions. Labels are created by calling `CTacLabel* CCodeBlock::CreateLabel`(<optional descriptive character string>), temporary values can be created with `CTacLabel* CCodeBlock::CreateTemp(const CType *type)`.

The call `cb->CleanupControlFlow()` above removes unnecessary GOTO's and labels from the IR. You may want to comment it out first to see what IR exactly your code is creating, and only enable it when you are sure that the generated IR is correct.

Compile your code using the test_ir target as follows

```
$ make test_ir
```

The file reference/4_test_ir is a binary of our reference implementation for this phase (using the relaxed constant folding scheme for integer constants). You can use it to compare your IR against the reference implementation. If you discover a bug in the reference implementation , please let us know.

Materials to submit:

- source code of your compiler (use Doxygen-style comments)
- brief report describing your implementation of the three-address code generation (PDF)

Submission:

- the deadline for the third phase is **Wednesday, May 25, 2016 before midnight**.
- email your submission to the TA (compiler-ta@csap.snu.ac.kr). The arrival time of your email counts as the submission time.

As usual: start early, ask often! We are here to help.

Happy coding!

Appendix: SnuPL/1 Intermediate Representation

The SnuPL/0 IR is implemented in `ir.cpp/h` and largely follows the textbook. The class hierarchy is illustrated below:

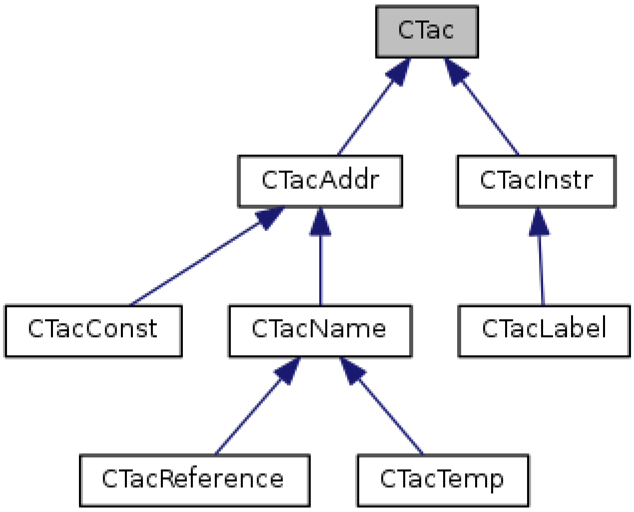

Illustration 1: Three-address code class hierarchy

`CTacAddr` and subclasses represent symbols, temporaries, and constant values. `CTacAddr` and its subclasses form the operands of `CTacInstr` instructions.

Operations are implemented using `CTacInstr`. `CTacLabel` is a special instruction that simply serves as a label and does not actually execute any code. `CTacLabel` can be used as an operand for branching operations (goto, if relop goto..., see below). Different operations require different operands, both in type and number; refer to Table 1 below.

The `CCodeBlock` class manages the list of instructions, and is also responsible to generate (unique) temporary values and labels. The relevant methods are:

```
CTacTemp* CCodeBlock::CreateTemp(const CType *type);
CTacLabel* CCodeBlock::CreateLabel(const char *hint=NULL);
CTacInstr* CCodeBlock::AddInstr(CTacInstr *instr);
```

`CScope` and its subclasses, finally, represent the module and procedures/functions of the program.

SnuPL/1 IR

Opcode	Dst	Src1	Src2	Description
opAdd	result	$operand_1$	$operand_2$	result := $operand_1$ + $operand_2$
opSub	result	$operand_1$	$operand_2$	result := $operand_1$ - $operand_2$
opMul	result	$operand_1$	$operand_2$	result := $operand_1$ * $operand_2$
opDiv	result	$operand_1$	$operand_2$	result := $operand_1$ / $operand_2$
opAnd	result	$operand_1$	$operand_2$	result := $operand_1$ && $operand_2$
opOr	result	$operand_1$	$operand_2$	result := $operand_1$ || $operand_2$
opNeg	result	operand		result := -operand
opPos	result	operand		result := +operand
opNot	result	operand		result := ~operand
opEqual	target	$operand_1$	$operand_2$	if $operand_1$ = $operand_2$ goto target
opNotEqual	target	$operand_1$	$operand_2$	if $operand_1$ # $operand_2$ goto target
opLessThan	target	$operand_1$	$operand_2$	if $operand_1$ < $operand_2$ goto target
opLessEqual	target	$operand_1$	$operand_2$	if $operand_1$ <= $operand_2$ goto target
opBiggerThan	target	$operand_1$	$operand_2$	if $operand_1$ > $operand_2$ goto target
opBiggerEqual	target	$operand_1$	$operand_2$	if $operand_1$ >= $operand_2$ goto target
opAssign	LHS	RHS		LHS := RHS
opAddress	result	operand		result := &operand
opDeref	result	operand		result := *operand
opCast	result	operand		result := (type)operand
opGoto	target			goto target
opCall	result	target		result := call target
opReturn		operand		return operand
opParam	index	operand		index-th parameter := operand
opLabel				jump target
opNop				no operation

Table 1: SnuPL/1 intermediate representation