Phase 4: Intermediate-Code Generation

In this fourth phase of the project, you will 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 if you like you may, of course, implement your own version. 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.

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 the statement

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

  return NULL;
}
```

that is, a jump to the next label should be inserted after encoding the statement (of course, you may not need the explicit goto in certain cases, e.g., while loops).

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 is

```
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))

Using the IR: the provided IR is fully implemented and functional. 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.

Submission:

- the deadline for the fourth phase is **November 27, 2014 before midnight**.
- submit a tarball of your SnuPL/0 compiler by email to the TA (compiler-ta@csap.snu.ac.kr). The arrival time of your email counts as the submission time.

As usual, and this time in particular: start early, ask often! We are here to help.

Happy coding!

Appendix: SnuPL/0 Intermediate Representation

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

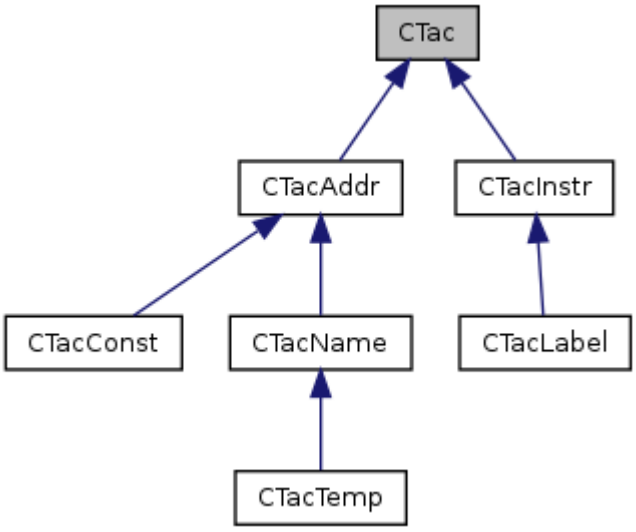

Illustration 1: Three-address code class hierarchy

`CTacAddr` and subclasses represent symbols, temporaries, and constant values. `CTacAddr` and its subclasses are used as operands.

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.

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
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
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/0 intermediate representation