Errata 1

Forward

The following are corrections needed for The Big Baby to pass all targeted test cases for the final project. Changes were made to pass four test cases. Due to the changes, the following lines are needed in codegen.h for these changes to compile:

```
#define LOGIC_XOR “xorl”
#define FUNCTION_RETURN_VALUE "-8(%ebp)"
```

Test Case 3 – Constant Folding

When two constant folding operations occur with the same constant, the first constant is removed from the table and the second one is mutated, thereby breaking usages of it in the TACs. The fix, shown in expression.c (expression_unary and expression_binary) and pascal.y (array_index) is to simple keep the constants in the table, and add a third constant with the result of the folding operation.

Test Case 7 – Correctly Handling Pascal Return statements

This test case caused a large conceptual change in how return statements were addressed in the compiler. It was assumed C-syntax for returning functions – a function exists and returns a variable. As a result, a return statement meant loading EAX, and waiting for the next instruction (assumed to be end_function) to put the assembly instructions to leave a scope.

However, in Pascal, the function *continues* executing after a return statement (<func_name> := <symbol>). There can even been multiple return statements in a row, in which only the last one takes effect.

For the change, a change was made to the stack offset logic – no longer was just the first 8 bytes reserved (old base pointer and static link), but now 12 bytes are reserved, even for procedures (this minor inefficiency was conceded for code understandability) to allow for a static location off the base pointer for the return value. When a return statement is found, the symbol is copied (loading it first into a register if it is in memory) into this static location. Now when end_function is found, the value in this static location is copied into EAX and the cleanup instructions are printed. The fix is made to codegen.c (code_return, code_begin_function, assign_offsets, code_end_function).

Test Case 9 – Incorrect NOT Operator in Code Generator

Since NOTL is a bitwise NOT operation it was producing incorrect results when dealing with Booleans. Therefore, the fix is to now XORL our Boolean with TRUE to produce the complement. This fix is shown in codegen.c, (code_tac).

Test Case 10 – Incorrect Adding of a Node to Expression Linked List

When an expression ($3) is added to an expression_list ($1), the expression was set as $1's next, erasing that current pointer. The fix, shown in pascal.y (expression_list) is to run down the list of next pointers to find the end of the list, and then wire the new element.

```
void code_tac(FILE *file, RegDesc *registers, Tac *current)
{
  switch (current->op)
    {

    <Unchanged code omitted>

    case TAC_NOT:
      //FIX 5-9-21 - INVERT is not what we want - thats a bit wise invert
      //Since we are using booleans, just xor them
      //code_unary(file, registers, INVERT, current->result, current->operand1);
      code_binary(file, registers, LOGIC_XOR, current->result, current->operand1, symbol_one);
      break;

      <Unchanged code omitted>

    }
}

void code_return(FILE *file, RegDesc *registers, Symbol *result)
{
  //FIX: 5-19-2011
  //We can't simply load eax, as function calls can come after this return statement
  //and mess up with what we want to return
  //Therefore, we must put this value in memory

  //Spill EAX
  //code_spill_reg(file, registers, REG_EAX);

  //Load eax
  //code_load_reg(file, registers, REG_EAX, result);

  //Load return symbol into register if it isn't already there
  int source_reg = get_result_register(file, registers, result);

  code_instruction(file, MOVE, registers[source_reg].name, FUNCTION_RETURN_VALUE);
}

void code_begin_function(FILE *file, Symbol *symbol)
{
  debug("Setting Scope to %s from %s", symbol_to_string(symbol), symbol_to_string(current_scope));
  //Set our scope
  current_scope = symbol;

  //print label
  fprintf(file, "%s:\n", symbol->name);

  //print our function header
  code_instruction(file, PUSH, EBP, NULL);
  code_instruction(file, MOVE, ESP, EBP);

```

```c
  if (symbol->symbols->nested == 1)
    {
      //this is main
      code_instruction(file, MOVE, make_integer(0), CURRENT_STATIC_LINK);
    }

  //Get the last offset, and adjust stack pointer
  //FIX 5-19-2011
  //int offset = -4;
  int offset = -8;
  int i;
  SymbolTable *table = symbol->symbols;
  Symbol *current;
  for (i = 0; i < HASHSIZE; i++)
    {
      current = table->entries[i];

      while (current != NULL)
      {
      if (current->offset < offset)
        offset = current->offset;
      current = current->next;
      }
    }

  //Set esp to point to the next location after our variables, make it positive so we subtract
  int esp_fix = -offset;

  code_instruction(file, SUBTRACT, make_integer(esp_fix), ESP);
}

void assign_offsets(Symbol *symbol)
{
//At this point, we need to assign offsets to our symbols in the table
  SymbolTable *table = symbol->symbols;
  int i;
  Symbol *current;
  Type *type;
  //int offset = -4;//Lets leave the first 4 for our static link
  //FIX 5-19-2011
  int offset = -8; //Reserve 4 for static link and 4 for return value

  for (i = 0; i < HASHSIZE; i++)
    {
      current = table->entries[i];

      while (current != NULL)
      {
      if (current->is_parameter != TRUE)
        {
          type = current->type;
          int code = type->code;
          if ((code != TYPE_PARAMETER) && (code != TYPE_PROGRAM) && (code != TYPE_PROCEDURE) && (code != TYPE_FUNCTION) && (code != TYPE_LABEL))
          {
          //we have a variable here that needs to take up space on the stack
          //subtract an offset, and assign it
          offset -= get_size(current->type);
          current->offset = offset;
          //offset -= get_size(current->type);

          //Set array C value
          //TODO: Fix for multidimensional arrays and values of larger than 4 bytes
          if (type->is_array == TRUE)
            {
              type->c = current->offset - type->intervals->start * 4;
            }
          }
        }
      current = current->next;
      }
    }

  //Now assign our parameters
  offset = 4;//first 4 bytes below is return address
  Tac *tac_current = symbol->parameters;
  while ((tac_current != NULL) && (tac_current->op != TAC_BEGINFUNCTION))
    {
      offset += get_size(tac_current->result->type);
      tac_current->result->offset = offset;
      tac_current = tac_current->prev;
    }

  //display
  printf("Displaying Symbol Table for: %s\n", symbol->name);
  symboltable_dump(table);
  printf("\n");
}

void code_end_function(FILE *file, RegDesc *registers)
{
  //FIX 5-19-2011
  if (current_scope->type->code == TYPE_FUNCTION)
    {
      //We need to return this value

      //Spill eax
      code_spill_reg(file, registers, REG_EAX);

      //Load it
      code_instruction(file, MOVE, FUNCTION_RETURN_VALUE, EAX);
    }

  code_flush_all(file, registers);

```

```c
  code_instruction(file, LEAVE, NULL, NULL);
  code_instruction(file, RETURN, NULL, NULL);
}
```

```c
Expression *expression_unary(int op, Expression *only)
{
  Tac *temp;
  Tac *result;
  Symbol *temp_variable;

  debug("Unary: %d Symbol: %s Type: %d", op, only->result->name, only->result->type->code);

  //Constant folding
  if (only->result->type->code == TYPE_NATURAL)
    {
      //Fix 5-19-2011
      //Since each symbol is added to the constant table only once
      //if we delete a symbol that was used previously
      //We wil have problems
      //Instead - make a new symbol, and do the wiring

      //Remove from symbol table
      //symboltable_delete(constantTable, only->result);

      Symbol *fold_result = make_symbol();
      fold_result->type = make_type(TYPE_NATURAL);
      fold_result->name = (char*)safe_malloc(sizeof(char) * 12);

      switch (op)
    {
    case TAC_NEGATIVE:
      //only->result->value.integer = - only->result->value.integer;
      fold_result->value.integer = - only->result->value.integer;

      //Make new name, overwriting old name
      //sprintf(only->result->name, "%d", only->result->value.integer);
      break;
    }

      //rewire expression to use new result
      only->result = fold_result;

      //Fix name
      sprintf(only->result->name, "%d", only->result->value.integer);

      //Reinsert
      symboltable_insert(constantTable, only->result);

      //debug("Folded constant to %d", only->result->value.integer);
      return only;
    }

  temp_variable = make_temp();
  temp_variable->type = make_type(TYPE_VARIABLE);//Generic variable type

  //Since we dont have a constant, we make a new tac with a temporary symbol
  temp = make_tac(TAC_VARIABLE, temp_variable, NULL, NULL);
  temp->prev = only->tac;

  //This is result of "calling" operator

  //FIX 5-19-2011
  //The backend looks for the only operand in operand1, not operand 2
  //result = make_tac(op, temp->result, NULL, only->result);

  result = make_tac(op, temp->result, only->result, NULL);
  result->prev = temp;

  //Rewire the expression struct to point to the true result
  only->result = temp->result;
  only->tac = result;

  return only;
}

Expression *expression_binary(int op, Expression *first, Expression *second)
{
  Tac *temp;
  Tac *result;
  Symbol *temp_variable;

  debug("Binary Op: %d First: %s Type: %d Second: %s Type: %d", op, first->result->name, first->result->type->code, second->result->name, second->result->type->code);

  //Constant folding
  //if ((first->result->type->code == TYPE_NATURAL) && (second->result->type->code == TYPE_NATURAL)
  //    && ((op == TAC_ADD) || (op == TAC_SUBTRACT) || (op == TAC_MULTIPLY) || (op == TAC_DIVIDE)))

  if ((first->result->type->code == TYPE_NATURAL) && (second->result->type->code == TYPE_NATURAL))
  {
    //Fix 5-19-2011
    //For same reasons in expression_unary

    //Remove both symbols from symbol table
    //symboltable_delete(constantTable, first->result);
    //symboltable_delete(constantTable, second->result);

    Symbol* fold_result = make_symbol();
    fold_result->type = make_type(TYPE_NATURAL);
    fold_result->name = (char*)safe_malloc(sizeof(char) * 12);

    switch (op)
      {
      case TAC_ADD:
    fold_result->value.integer = first->result->value.integer + second->result->value.integer;
```

```
        break;

          case TAC_SUBTRACT:
        fold_result->value.integer = first->result->value.integer - second->result->value.integer;
        break;

          case TAC_MULTIPLY:
        fold_result->value.integer = first->result->value.integer * second->result->value.integer;
        break;

          case TAC_DIVIDE:
        fold_result->value.integer = first->result->value.integer / second->result->value.integer;
        break;

          case TAC_GT:
        fold_result->value.integer = first->result->value.integer > second->result->value.integer;
        break;

          case TAC_LT:
        fold_result->value.integer = first->result->value.integer < second->result->value.integer;
        break;

          case TAC_GTE:
        fold_result->value.integer = first->result->value.integer >= second->result->value.integer;
        break;

          case TAC_LTE:
        fold_result->value.integer = first->result->value.integer <= second->result->value.integer;
        break;

          case TAC_EQUAL:
        fold_result->value.integer = first->result->value.integer == second->result->value.integer;
        break;

          case TAC_NOTEQUAL:
        fold_result->value.integer = first->result->value.integer != second->result->value.integer;
        break;

          default:
        die("Unrecognized Binary Operation: %s", op);
        break;

          }

```

```c
    //Rewire expression
    first->result = fold_result;

    //Now fix the symbol name
    sprintf(first->result->name, "%d", first->result->value.integer);

    //Insert the symbol back
    symboltable_insert(constantTable, first->result);

    //Give back some resources
    //free(second->result);
    free(second);

    debug("Folded constant %d", first->result->value.integer);

    return first;
  }

  //Create temp
  temp_variable = make_temp();
  temp_variable->type = make_type(TYPE_VARIABLE);//Generic variable type

  temp = make_tac(TAC_VARIABLE, temp_variable, NULL, NULL);
  temp->prev = join_tac(first->tac, second->tac);

  //Call operator
  result = make_tac(op, temp->result, first->result, second->result);
  result->prev = temp;

  //Rewire the expression
  first->result = temp->result;
  first->tac = result;

  //Cleanup
  free(second);
  return first;
}
```

```
array_index: NATURAL
{
 $$ = $1;
}
  | '-' NATURAL
{
 //Fix: 5-19-2011 See Expression_unary
 //Remove from constant table
 //symboltable_delete(constantTable, $2);

 Symbol *fold_result = make_symbol();
 fold_result->type = make_type(TYPE_NATURAL);
 fold_result->name = (char*)safe_malloc(sizeof(char) * 12);

 //Fix integer and name
 //$2->value.integer = $2->value.integer * -1;
 //sprintf($2->name, "%d", $2->value.integer);

 fold_result->value.integer = $2->value.integer * -1;
 sprintf(fold_result->name, "%d", fold_result->value.integer);

 //Put back in constant table
 symboltable_insert(constantTable, fold_result);

 //Return
 $$ = fold_result;
}
  ;

expression_list:
  expression
{
 $$ = $1;
}
  | expression_list ',' expression
{
 //FIX 5-19-2011 - To correctly add a node to the list, we must run down the next pointers
 Expression *prev = $1;
 Expression *current = prev->next;
 while (current != NULL)
  {
   prev = current;
   current = current->next;
  }

 prev->next = $3;

 $$ = $1;
}
  ;
```