COMP3109 Assignment 2

Compiler Front-End (10 marks)

This third assignment is a **group assignment**, and is due in Week 12 **on Friday, October 25, at 5pm**. Your assignment will not be assessed unless all of the following criteria are met:

1. Hand in a signed academic honesty form in the tutorial before
2. Submit a tarball of your source code to eLearning. The documentation should be done in form of comments. Please provide plenty of them.

In this assignment you will be implementing an optimizer for the intermediate language from the previous assignment. Your optimizer should read intermediate code from a file, construct a control-flow graph (CFG) for each function, then apply optimizations on the individual CFGs, and finally write the optimized intermediate code to another file.

The CFG of your optimizer should consist of control-flow edges and basic blocks, which in turn contain intermediate code instructions. Be careful to take the other tasks of this assignment into consideration when designing the corresponding data structures. A flexible design will be of help during the implementation of the subsequent optimizations.

Once the CFG is available, several optimizations should be performed. The first optimization targets unreachable code, i.e., code that cannot be reached by any execution of the program. The analysis for this problem is a simple depth-first traversal of the CFG, marking all visited basic blocks as reachable. The subsequent transformation then removes all unmarked blocks.

The second optimization is responsible to cleanup instructions that do not compute anything useful, e.g., instructions left-over by some other optimization. Dead code elimination uses data-flow analysis to determines which computations are actually used by a program and which computations can be eliminated without any visible side-effect. The optimization should consists of two phases. First, a data-flow analysis keeps track of registers holding useful values. You have to implement an iterative solver that computes a fixed-point solution for this data-flow analysis problem. The second phase, performs the actual transformation, i.e., removes all instructions that do not compute anything useful.

The final optimization targets the elimination of redundant load instructions (`ld`). You should define a data-flow analysis problem that determines whether the value read by such a load instruction is already available in some register, i.e., the value of the corresponding variable has been loaded previously into that register and the variable as well as the register have not been modified since. Finally, implement a transformation that replaces registers accordingly, without removing the actual load instructions (this would render the optimization considerably more complicated).

Your optimizer should apply these optimizations repeatedly until no further improvements can be achieved. The optimizer should additionally work with any syntactically correct intermediate code. You should not limit your tests to files generated by the front-end developed for Assignment 2.

Intermediate Code

The grammar of the intermediate language is defined below (it is unchanged with regard to Assignment 2).

Program Structure

A program is a list of functions, where a function is a list consisting of the function's name followed by a lists of basic blocks. Each basic block in turn is a list consisting of a block number and a list of instructions.

Instructions

The intermediate code supports 13 different instructions:

- Load constant (`lc`)
  Copy a constant into a register.

  Example:
  `(lc r5 5)` ... copy the value 5 into register 5
- Load instructions (`ld`)
  Read the value of a variable and copy its value into a register.

  Example:
  `(ld r5 a)` ... read the value of `a` and copy it into register 5
- Store instructions (`st`)
  Write the value of a register to a variable.

  Example:
  `(st b r5)` ... read the value of register 5 and assign it to variable `b`
- Arithmetic instructions (`add`, `sub`, `mul`, `div`)
  Perform the respective arithmetic operations on registers.

  Example:
  `(add r3 r4 r5)` ... read register 4 and 5 and write the sum into register 3
- Comparison instructions (`lt`, `gt`, `feq`)
  Perform a comparison and write the value 0 (false) or 1 (true) into a register.

  Example:
  `(eq r3 r4 r5)` ... compare register 4 and 5 and write the result into register 3
- Branch instructions (`br`)
  Perform a conditional branch depending on a register value. If the value of the register is non-zero the execution continues at the basic block whose number is specified by the second operand. The execution, otherwise, resumes at the basic block specified by the third operand.

  Example:
  `(br r3 1 2)` ... depending on register 3 branch to basic block 1 or 2
- Return instructions (`ret`)
  Return the value of a register from a function.

  Example:
  `(ret r3)` ... exit the current function and return the value of register 3

- Call instructions (`call`)
  Perform a function call. The second argument specifies the function to be invoked, followed by a list of registers whose values should be passed as function arguments. The return value of the called function is written into the register specified as the first operand.

  Example:
  `(call r1 factorial r3)` ... invoke function `factorial` and pass the value of register 3 as its first (and only) argument; write the return value into register 1

Intermediate Code Grammar

```
<program> ::= ( <functions> )

<functions> ::= ε
<functions> ::= <function> <functions>

<function> ::= ( <ID> <arguments> <blocks> )

<arguments> ::= ( <id_list> )

<id_list> ::= ε
<id_list> ::= <ID> <id_list>

<blocks> ::= ( <NUM> instructions )
<blocks> ::= ( <NUM> instructions ) blocks

<instructions> ::= <instruction>
<instructions> ::= <instruction> <instructions>

<instruction> ::= ( lc <REG> <NUM> )
<instruction> ::= ( ld <REG> <ID> )
<instruction> ::= ( st <ID> <REG> )
<instruction> ::= ( add <REG> <REG> <REG> )
<instruction> ::= ( sub <REG> <REG> <REG> )
<instruction> ::= ( mul <REG> <REG> <REG> )
<instruction> ::= ( div <REG> <REG> <REG> )
<instruction> ::= ( lt <REG> <REG> <REG> )
<instruction> ::= ( gt <REG> <REG> <REG> )
<instruction> ::= ( eq <REG> <REG> <REG> )
<instruction> ::= ( br <REG> <NUM> <NUM> )
<instruction> ::= ( ret <REG> )
<instruction> ::= ( call <REG> <ID> <reg_list> )

<reg_list> ::= ε
<reg_list> ::= <REG> <reg_list>

<NUM> ::= -?[0-9]+
<ID> ::= [a-zA-Z][a-zA-Z0-9]*
<REG> ::= r[1-9][0-9]*
```

Example

The unoptimized intermediate code of an example factorial program might initially look like this:

```
( (factorial (n)
    (0  (ld r1 n)
        (lc r2 0)
        (eq r3 r1 r2)
        (st cond r3)
        (ld r4 cond)
        (br r4 1 2) )
    (1  (lc r5 1)
        (st tmp r5)
        (ld r6 tmp)
        (ret r6) )
    (2  (ld r7 n)
        (lc r8 1)
        (sub r9 r7 r8)
        (st tmp r9)
        (ld r10 tmp)
        (call r11 factorial r10)
        (ld r12 n)
        (mul r13 r11 r12)
        (st tmp r13)
        (ld r14 tmp)
        (ret r14) ) )
  (main (n)
    (0  (ld r1 n)
        (call r2 factorial r1)
        (st tmp r2)
        (ld r3 tmp)
        (ret r3) ) ) )
```

The optimizer should generate an optimized program from this intermediate code as illustrated below:

```
( (factorial (n)
    (0  (ld r1 n)
        (lc r2 0)
        (eq r3 r1 r2)
        (st cond r3)
        (ld r4 cond)
        (br r4 1 2) )
    (1  (lc r5 1)
        (st tmp r5)
        (ld r6 tmp)
        (ret r6) )
    (2  (lc r8 1)
        (sub r9 r1 r8)
```

```
        (st tmp r9)
        (ld r10 tmp)
        (call r11 factorial r10)
        (mul r13 r11 r1)
        (st tmp r13)
        (ld r14 tmp)
        (ret r14) ) )
  (main (n)
    (0  (ld r1 n)
        (call r2 factorial r1)
        (st tmp r2)
        (ld r3 tmp)
        (ret r3) ) ) )
```

Task 1 Control-Flow Graph Constructions (2 marks)

The design of the internal data structures can significantly impact the extensibility and maintainability of a compiler. Your task is to design a suitable data structure for your optimizer, based on the usual definition of a control-flow graph:

- Nodes in the graph are basic blocks, each consisting of a sequence of instructions
- Edges in the graph represent the potential flow of execution.

Take care to take the other tasks of this assignment into consideration when designing the data structure. Document, explain, and justify your design decisions in the relevant code parts.

Task 2 Unreachable Code (2 marks)

The fist optimization targets unreachable code, i.e., code that can never be reached by any execution. Your task is to develop a simple analysis based on depth-fist search that marks basic blocks as reachable. The transformation phase of this optimization then simply removes all basic blocks that have not been marked.

Task 3 Dead Code Elimination (3 marks)

Dead code eliminate is responsible of cleaning up useless code – that is often left-over from optimization performed before (see below). The optimization consists of a data-flow analysis and a transformation phase.

The data-flow analysis tracks for each register whether it will be *used* subsequently. This is done by performing a backward data-flow analysis, i.e., information is propagated backwards with respect to the flow of execution. A register is considered to be *used* if its value (1) is returned from the current function (`ret`), (2) is stored to a variable (`st`), (3) is used as a condition for a branch (`br`). or (4) is used by an instructions whose result is used. Your task is to define a data-flow analysis, including lattice, transfer functions, and meet operator. In addition, you should develop an iterative solver that computes a fixed-point solution. Document each component of your analysis thoroughly.

The transformation phase then simply removes all instructions from the program whose result is not used according to the data-flow analysis.

Example:

```
(0   (ld r1 x)
     (br r1 1 2) )
(1   (lc r2 1)
     (add r1 r1 r2)
     (br r1 3 3) )
(2   (ld r2 x)
     (lc r3 2)
     (add r1 r1 r3)
     (br r1 3 3) )
(3   (ld r5 x)
     (ret r5) )
```

Given the code from above the analysis determines that the value returned from the program (`r5`) along with the corresponding load instruction has to be retained. In basic blocks 1 and 2 the addition and load constant instructions have to be retained since the result of the addition is used as a condition for the respective branches. The same applies for the conditional branch and the load instruction of basic block 0. The load instruction in basic block 2, however, is not used anywhere (neither by a return, branch, store, or any other instruction whose result has to be retained). The load thus can be removed from the program as shown below.

```
(0   (ld r1 x)
     (br r1 1 2) )
(1   (lc r2 1)
     (add r1 r1 r2)
     (br r1 3 3) )
(2   (lc r3 2)
     (add r1 r1 r3)
     (br r1 3 3) )
(3   (ld r5 x)
     (ret r5) )
```

Task 4 — Elimination of Redundant Loads (3 marks)

Avoiding redundant computations is a frequent way of optimizing programs. For this exercise you have to develop an optimization that eliminates redundant load instructions (`ld`). Your solution should consist of a data-flow analysis (that determines which registers hold the value of a variable) and a corresponding transformation.

The data-flow analysis has to track which registers hold the value of a variable. Clearly, after a load instruction `(ld r, v)`, the destination register `r` holds the value of variable `v`. This remains unchanged until either `r` is redefined or the value of `v` is overwritten by a store (`st`). Your task is to define a corresponding data-flow analysis problem, consisting of a lattice, transfer functions, and a meet operator. Provide detailed documentation and explanations for each of the analysis's components. You should extended the iterative solver of Task 3 to solve the respective data-flow equations.

The goal of this optimization is to eliminate redundant load instructions from the program. However, for the sake of simplicity the transformation phase of the optimization does not remove any instructions (as this would invalidate the analysis information). Instead, only register uses are rewritten according to the result of the data-flow analysis. Once all instructions referring to the register defined by a load instruction are rewritten, the load becomes dead code and can be removed using dead code elimination (Task 3).

Hint: The data-flow analysis problem required to solve this task has the structure of a GEN/KILL problem, a simple and frequent form of analysis problems. The transformation phase has a large impact on how successful the optimization can eliminate load instructions. You thus need to device a good strategy to decide how the rewrite the program.

Example:

```
(0   (ld r1 x)
     (br r1 1 2) )
(1   (lc r2 1)
     (add r1 r1 r2)
     (br r1 3 3) )
(2   (ld r2 x)
     (lc r3 2)
     (add r1 r2 r3)
     (br r1 3 3) )
(3   (ld r5 x)
     (ret r5) )
```

Given the basic blocks from above, the analysis determines that at the end of basic block 0, `r1` holds the value of `x`. Basic block 1 does not contain any load, however, register `r1` is redefined. The analysis thus determines that `x` is not available in any register at the end of basic block 1. Basic block 2 similarly contains a redefinition of register `r1`. In addition the block contains a load instruction, which allows the analysis to determine that `r2` holds the value of `x` at the end of this block. Note that both, `r1` and `r2`, hold the value of `x` up to the point of the redefinition of `r1` within basic block 2. The analysis then determines that `x` is not available in any register at the beginning of basic block 3 by combining the information of basic block 1 and 2.

Using this information, the registers in the program can be rewritten (as shown by the code below). In this example only the use of register `r2` of the addition instruction within basic block 2 is updated to refer to `r1`. This renders the preceding load to `r2` in the same basic block dead code (which will be removed later on).

```
(0   (ld r1 x)
     (br r1 1 2) )
(1   (lc r2 1)
```

```
    (add r1 r1 r2)
    (br r1 3 3) )
(2  (ld r2 x)             -- becomes dead code
    (lc r3 2)
    (add r1 r1 r3)        -- use of r2 rewritten to r1
    (br r1 3 3) )
(3  (ld r5 x)
    (ret r5) )
```

Implementation

Your assignment needs to run on the `ucpu[01]` machines. You need to provide all code required to build and execute your compiler. You need to provide at a minimum a script, `optimize.sh` and a `README` file. The `README` should explain where the relevant code corresponding to each of the tasks can be found. The `optimize.sh` script should take two arguments: the first argument is the name of a file containing unoptimized intermediate code, while the second argument is the name of the intermediate code file that should be produced by your optimizer.

Further Reading

If you need more background on data-flow analysis, please consult any of the many text books on compiler construction. In particular, the book "Compilers" by Aho, Lam, Sethi, and Ullman provides a very good introduction to the topic.