COL226: Programming Languages
II semester 2021-22

Assignmant: The Basic Decimal Integer Machine

The following table is the instruction set of a very Basic Decimal Integer Machine (BDIM) which may be used to do simple integer computations. There are no other data types and no structured data types. You are supposed to design an interpreter which reads in the instructions (one per line) from a program file `<filename>.bdim`.

- We use positive integer `op`-codes. The special opcode `0` denotes a **halt** instruction. Programs halt on seeing this opcode.
- Assume that `code` is a vector (read-only) of quadruples called *three-address code*. For each value `c` in the range of indices of the vector `code[c]` denotes the quadruple at index `c`. Each quadruple is an instruction of the BDIM machine.
- Assume that memory is an array (with a fixed maximum size pre-defined by a value *maxMemSize*) `mem[0..maxMemSize-1]` of integers. The entire memory consists of registers. For each `i` in the range `0` to `maxMemSize-1`, `mem[i]` denotes the value stored at index `i`
- **Constant literals.** The operand `v` denotes a constant integer value (e.g. for reading input from the terminal). Inputs are read only one value at a time. So for each value a separate instruction needs to be written.

  Constant internal. Unlike input constant literals these are constant values required to be stored as part of the code of the program (e.g. initializations).

  Memory access. The operands `i`, `j` and `k` of any operation are indexes into `mem`.

  Code access. The operand `c` denotes an index into the code-segment `code`. Clearly if vector-bounds are not respected the program halts.
- Each instruction is a quadruple `(op, opd1, opd2, tgt)` of 4 non-negative integers representing respectively the operation `op` to be performed on operands `opd1` and `opd2` whose result is stored in `tgt`.
- Boolean values are represented by `0` for **false** and `1` for **true** respectively.
- An underscore "_" (in the table below) denotes an **inessential** operand is a wild-card and could be any integer and is ignored for the purposes of that instruction. But in any actual program there needs to be an actual integer value, since all instructions are of type `int*int*int*int`.
- Every operand that is not "_" is an **essential** operand and should have only non-negative values.
- Assume that the maximum integer that can be read, stored or printed are limited by whatever your machne allows for the integer datatype (e.g. `valOf(Int.maxInt)`).

op	opd1	opd2	tgt	Meaning
0	_	_	_	**halt**
1	v	_	k	**input:** `mem[k] := v`
2	i	_	k	`mem[k] := mem[i]`
3	i	_	k	`mem[k] :=` **not** `mem[i]`
4	i	j	k	`mem[k] := mem[i]` **or** `mem[j]`
5	i	j	k	`mem[k] := mem[i]` **and** `mem[j]`
6	i	j	k	`mem[k] := mem[i] + mem[j]`
7	i	j	k	`mem[k] := mem[i] - mem[j]`
8	i	j	k	`mem[k] := mem[i] * mem[j]`
9	i	j	k	`mem[k] := mem[i]` **div** `mem[j]`
10	i	j	k	`mem[k] := mem[i]` **mod** `mem[j]`
11	i	j	k	`mem[k] := (mem[i] = mem[j])`
12	i	j	k	`mem[k] := (mem[i] > mem[j])`
13	i	_	c	`if mem[i] goto code[c]`
14	_	_	c	`goto code[c]`
15	i	_	_	**output:** `print mem[i]`
16	v	_	k	`mem[k] := v`

What you need to do

1. Create program files in BDIM for the following problems. Make sure
   - each program has an output instruction to print the result of executing the program.
   - each line of the program file contains exactly one quadruple.
   - there are no blank lines (especially at the end of the file).
   - Also the last line of the file ends with a *newline* character.

   (a) `abs.bdim`. Finding the absolute value of an integer.

   (b) `ap.bdim`. Given inputs a, d, n, to compute the sum of the arithmetic progression of n integers starting from a with a common difference of d.

   (c) `fact.bdim`. Finding the factorial of a given non-negative integer.

   (d) `fib.bdim`. Finding the value of the n fibonacci number given that $fib(0) = 0$ and $fib(1) = 1$

   (e) `gcd.bdim`. BDIM program to compute the gcd of two integers.

   (f) `reverse.bdim`. BDIM program to compute the integer value obtained by *reversing the digits* of a given input non-negative integer.

   (g) `russian.bdim`. The russion multiplication algorithm for multiplying two numbers defined by the following SML code.

```
fun russianMult (x, y) =
    if ((x = 0) orelse (y = 0)) then 0
    else
        let val twox = 2*x
            val halfy = y div 2
        in if (y mod 2 = 0)
           then russianMult (twox, halfy)
           else x + russianMult (twox, halfy)
        end
```

   (h) Any other interesting functions you may have thought of, each in a separate `_.bdim` file.

2. Write an interpreter (filename: `bdim.sml`) in SML-NJ for programs written in BDIM.

- The interpreter reads in the `_.bdim` file into the vector `code` which contains exactly one instruction (quadruple) per line with no blank lines anywhere.
- The interpreter executes a function `interpret` which evaluates the code starting from the instruction `code[0]` as per the explanations given in the instruction set.
- Since there may be multiple **input** instructions, you may need to prompt the user with a "`input:` " message.
- For operations such as `div` and `mod` which are not defined for "division by zero" you have to do a check before performing such operations, so as to "trap" the error and abort the program gracefully. Similar remarks also apply to certain other operations whose essential operands have negative indices, or jumps to code indices that are negative.

3. Zip up all your files (`bdim.sml` and other `_.bdim` files test files into a file called `<your-entry-number>.zip` and submit this single file on moodle.

Note

1. You are *not* allowed to change any of the names or types given in the specification/signature. You are not even allowed to change upper-case letters to lower-case letters or vice-versa.
2. The evaluator may use automatic scripts to evaluate the assignments, especially when the number of submissions is large.
3. You may define any new auxiliary functions you like in your code besides those mentioned in the specification.
4. Your program should implement the given specifications/signature.
5. You need to think of the *most efficient way* of implementing the various functions given in the specification/signature so that the function results satisfy their definitions and properties.
6. The evaluator may look at your source code before evaluating it, you must explain your algorithms in the form of comments, so that the evaluator can understand what you have implemented.
7. Do *not* add any more decorations or functions or user-interfaces in order to impress the evaluator of the program. Nobody is going to be impressed by it.
8. There is a serious penalty for code similarity (similarity goes much deeper than variable names, indentation and line numbering). If it is felt that there is too much similarity in the code between any two persons, then both are going to be penalized equally. So please set permissions on your directories, so that others have no access to your programs.