Design internals

MIPS Function calling convention implementation

We follow mips convention:
First arguments try to go into the designated registers.

```
private final static String[] INT_ARGS = {"$a3", "$a2", "$a1", "$a0"};
private final static String[] FLOAT_ARGS = {"$f14", "$f12"};
```

others go on the stack.
After the function is called all the arguments are copied into its stackframe/registers.

Class design and general code structure

When the compiler frontend finishes we are left with a single variable of type ProgramIR(you can see checkout this interface in our code).
Each variable is now stored as a BackendVariable which is responsible for knowing where that variable is stored during the whole function scope. Since a variable can be stored on stack some instructions will need to load them temporarily inside registers. For this we made a beautiful abstraction `LoadedVariable:`
Interface is quite simple:

```
public interface ILoadedVariable {
   public String loadAssembly();
   public String getRegister();
   public String flushAssembly();
}
```

We basically feed the backend variable into it with the base type that we want it loaded as and it abstracts away all the instructions needed for casting, loading, storing them inside temporary variables and flushing them back to their original location. It even works with constants!
One tradeoff we made was that modifying `loadedvariable.getRegister()` in runtime should not change actual value inside the original variable, unless `.flushAssembly()` is used. This makes it easier for us to reason about code but results in too many temporary variables and copying(when variables are already stored inside a register). This could be fixed by an alternative `LoadedVariable` which does not give this guarantee.

Otherwise most of the instruction translation is done inside `translateInstructions.` Our code is written in a way that instruction translation code does not care about the allocation algorithm.
In runtime stack frame is controlled using $sp and $fp registers. We use $fp for referring to variables since during function calls $sp might change.

Design of data structures used to implement each register allocation scheme: naïve, block, Briggs

- For naive algorithm `BackendVariable` and `LoadedVariable` do most of the heavy lifting. Each `BackendVariable` is marked as spilled and we load/store them using `LoadedVariable`.
- In intrablock findBlocks function gives us back List<IRBlock> where each IRBlock stores its instructions and neighbors. `LoadedVariable` is still useful, duh.
    - Before and after each block only variables which appear in IRInstruction.reads() and IRInstruction.writes() are loaded and stored.
- Briggs:
    - Block finding is identical to intrablock
    - Liveness analysis gives each BackendVariable a boolean for each instruction where it is “alive”
        - Using this we can tell which variables intersect
        - IRInstruction.reads() and IRInstruction.writes() come in handy here too
    - We pass down loopDepth from compiler frontend which is used by spill calculator
        - spill costs are written in BackendVariable
    - Graph coloring is done on the BackendVariables and assigns registers to them
    - `LoadedVariable` knows how to load variables during instruction translation.

Bugs or features that are not operational

No bugs, just possible improvements.

Possible Improvements

- Save only used registers in functions
- Alternative LoadedVariable which does not use temporary registers unnecessarily
- Use more registers for intrablock/briggs