Scribble memory allocation

Currently Scribble can store values in three locations. The registers, the stack or on the heap. When a function is executing the value of its local variables are stored in registers, this was useful when I was developing language as it allowed me to prototype more easily and check references very easily it would however be more efficient in terms of speed to use the register area for temporary values and store local variables on the stack and I will hopefully implement it this way in the future.

The stack is currently used to store the values of registers during certain operations such as a function call or a statement which requires more registers than it can use and restore them afterwards. It is however given a base pointer and a current pointer as opposed to just a current pointer so that I can fairly easily extend it to store local variables later on.

Both the registers and the stack are only capable of storing primitives and don't care about the primitives type when processing. Because these are all primitives values larger than 8 bytes wide are not supported.

The heap is a much stricter environment, it was designed from my knowledge of the Java heap however it is likely. Each element on the heap has a type which will either have to be registered with the heap before execution (like a new structure) or in the case of an array the VM will generate the type at runtime (So as long as int is a registered type array(int) will be valid, and so will array(array(int))). Each entry in the heap has a size and in the case of arrays will have an element size (So the length of an array can be computed). Access to the heap is provided through OpCodes in the virtual machine and is very controlled.

Within the language all structures and arrays or strings will be allocated on the heap. Any other type will be allocated on either the registers or the stack. Primitives inside structures on the heap will not be references and will instead be allocated within the memory of that structure so

```
type A := struct {
        a : int,
        b : int,
        c : A
};
```

would have be allocated enough space for 2 integers plus a reference. The location of each element will be computed at compile time (So in the case of var j := A->a; the offset of a would be calculated at compile time) however information on the contents of a structure will also be written into its type so that reflection could potentially be added later.