`CodeGen.scala`

Generating MIPS code given a AST

MIPS Registers

MIPS instruction uses 5 bits for register addressing, so there can be 2^5 = 32 registers

Number	Name	Use	Preserved across function calls?
0	`$zero`	constant 0	—
1	`$at`	assembler temporary	no
2, 3	`$v0, $v1`	function return values	no
4 - 7	`$a0 – $a3`	function arguments	no
8 - 15	`$t0 – $t7`	temporaries	no
16 - 23	`$s0 – $s7`	temporaries	yes
24 - 25	`$t8 – $t9`	temporaries	no

MIPS Review

```
  # Hello, World! program
.data ## Data declaration section
  ## String to be printed:
  out_string: .asciiz "\nHello, World!\n"
.text ## Assembly language instructions go in text segment
  main: ## Start of code section
  li $v0, 4 # system call code for printing string = 4
  la $a0, out_string # load address of string to be printed into $a0
  syscall # call operating system to perform operation
  # specified in $v0
  # syscall takes its arguments from $a0, $a1, ...
  li $v0, 10 # terminate program
  syscall
```

Concerning `Cint_lit`

In the code generation phase, we need to dump the all the literals in `.data` section

```
  .word -1          # for garbage collector
int_lit13:          # 13 was incrementor, starting from 0
  .word 5           # class tag number
  .word 4           # object size, 4 bytes
  .word Int_dispTab # int's dispatch table label
  .word 20          # 20 was the value of the Int object? not sure here
  .word -1          # next one ..
```

Concerning `Cstring_lit`

The same here, for strings it similar layout, except that for non-printable characters we need to use their ASCII representation. Also, we should not forget 0 to tell assembler that string ended.

```
  .word -1                # for garbage collector
string_lit22:             # 22 was incrementor, starting from 0
  .word 3                 # class tag number
  .word 10                # object size, 10 bytes
  .word String_dispTab    # String dispatch table label
  .word int_lit13         # length label of string
  .ascii "Done with 8 queens!"  # text
  .byte 10                      # non-printable character being
                                # to Int (i.e. '\n'.toInt)
  .byte 0                 # \0
  .align 2                # align data to 2^n bytes (i.e. 2^2)
```

What does `.align` mean?

Concerning `.align`

The "word"s should be in a *word boundary* meaning in MIPS which is a 32 bit we need to have words as 4 bytes, next word as the next 4 bytes and etc.

`ClassameTable`

Table of class names used in the program

```
class_nameTab:
  .word string_lit1
  .word string_lit13
  .word string_lit11
  .word string_lit10
  .word string_lit7
  .word string_lit6
  .word string_lit5
  .word string_lit3
  .word string_lit15
  .word string_lit21
  .word string_lit14
```

Dispatch table for `Symbol`

```
Symbol_dispTab:
  .word Any.Any          # base class constructor
  .word Symbol.toString # toString method that Symbol overrode
  .word Any.equals       # base class equals method
  .word Symbol.Symbol    # Symbol constructor
  .word Symbol.hashCode # hashCode method of Symbol
```

Object prototype for `Symbol` or attribute table

```
  .word -1                # garbage collector tag
Symbol_protObj:
  .word 2                 # class tag
  .word 6                 # object size
  .word Symbol_dispTab    # dispatch table of symbol
  .word 0                 # attribute #1
  .word 0                 # attribute #2
  .word int_lit0          # attribute #2
```

```
class Symbol() {
  var next = native;
  var name: String = "";
  var hash: Int = 0;
}
```

How to use class prototype to create the new object

```
override def visit_alloc(calloc: Calloc, type_name: Symbol) = {
  // get the name of the symbol
  val name: String = output.symbol_name(type_name)
  // load the address of the class name prototype label into $ac
  emitter.opc("la").opn(emitter.s_ACC()).opn(name + emitter.s_PROTOBJ()).endl(calloc)
  // jump and link to Any.Clone method which is a built-in method to cool runtime
  emitter.opc("jal").opn(emitter.s_ANYCLONE).endl(calloc)
};
```

Offsets

We use the combination of offsets and labels to use the static data (i.e. dispatch table, class table and etc.).

- Note that addresses are in bytes in MIPS

```
  .word -1    # offset -4 with respect to "label" (or label's address + (-4))
label:
  .word 2     # offset 0 (or label's address + 0)
  .word 3     # offset 4 (or label's address + 4)
```