Report

Correspondence to MIPS code:

Even a 40 line Cool code generates 600 lines of spim code. As we know spim code is of two parts .data which contains varibles and .text which contains implementation details. We can observe that .data takes up nearly 300 lines of code to intialise each class, variable, strings to print etc. And then starts the .text code as a part from heap_start label.

.data : It creates a label for each variable and initalises it. For example, if we want to print a string “Enter number”, it creates a label for it, initalises reqquired fields for it to contain that string and stores that string. When ever we want to print this string, it is loaded using “la” instruction . This is applicable for all strings and identifiers.

It contains labels for datatype constants like string constants, int constants, bool constants and for each object like Main, Int, String, Bool, IO and labels for group of variables or datatype constants that are under particular class.

Suppose we have i <- 2, i is stored under a string constant and 2 is stored a int constant. There will be again datatype labels which store Int.

(label is used to refer to sections in the spim code, they are not essentially same as label that is used in spim code)

.text : Starts with heap_start which contains init functions for all object types. Main_init, Int_init, String_init, Bool_init and Main.main (in case of fib.cl, program for fibonacci numbers). The label names say what they do. We can see that input is taken in Main_init by calling IO_init which calls Object_init. We can also see that all dataype init functions call object_init since this initialises any object.

And then Main.main implements main function of Main class in Cool code. For fib.c program it loads str_const1 first which is the first prompt statement in main and then we can see bne (branch on not equal to) for the corresponding if statement in Cool code. Bne is generally compiler optimization followed for if statements. In this bne statement, we can see that it branches to other labels.

Labels are in the same order as they are in statement order of Cool code. For example, fib function is defined before main in Cool code. We can see that Main.fib is before Main.main in spim code. And so are the lables named in that order.

Continuing above bne, each statement is implemented under a label and in each label, error checking is happening. Implementing each statement as a label makes sense because we need to validify that each statement is implemented properly and there is no error . We can see a bunch of jumping to _dispatch_abort statement. The number of lines that have proper code statements is almost same as number of labels that start with label# which explains why there are so many labels. We can also see many lw statements that load proper addresses from stack to which flow of control shoould jump to. So the flow of control for fib propgram starting from main

```
 main() : SELF_TYPE {                    --Main.main
        {
          out_string("enter a number of fibonacci terms needed\n");  --str_const1
           size <- in_int();          --label6  ->l abel7
           if 0 < size  then{       --label10
              out_string("1  ");   --label10 - str_const2
              fib(0,1);              --label11
              out_string("\n");    --str_const3 - label12
              }
           else out_string("positive number needed\n") -label8
```

label10 branches to label11 or label8 depending on predicate of if statement.
label8 goesto label 14 which ends the program by setting stack to its initial.

Label12 after going to other labels ends the program.
Similar correspondence can be found for fib function in Cool code with Main.fib and label0-5.
Understanding errors that were produced while developing the code:
Dispatch abort error due to statement *"jal _dispatch_abort"* which is checked in every label where there can be a chance of improper implementatio. For example, out_string("something"). Here, there can be chance of the function not being properly invoked, string constant not properly loaded etc.
Increasing heap... Can't expand data segment : We saw that .text is under label heap_start. So an infinite loop can potentially result in such errors.

Incorrect programs - Errors:
Rules I have broken
section 10.1 : Integers, Identifiers, and special notation
Error message : **syntax error at or near TYPEID = Size**
Type identifiers begin with a capital letter; object identifiers begin with a lower case letter. We can make type identifiers to start with lowercase and object identifiers to start with upper case.
Section 10.2 Strings
Error message : **syntax error at or near ERROR = '**
Since strings should be enclosed in double quotes "...", we can initialise string as str <- 'something' . Or str <- '\0' (initialising to null)
Section 10.3 Comments
Error message : **syntax error at or near ERROR = EOF in comment**
1. As comments cannot cross file boundaries, if we omit closing commets *) , it results in a error since it reaches EOF.
Section 10.4 Keywords
Error message : **syntax error at or near TYPEID = False**
1. Since the first letter of true and false must be lowercase, writing True or False results in error.
Section 10.5 Whitespace
It does not describe any rule. It's just a description.

Other rules that can be broken are:
section 10.1 : Integers, Identifiers, and special notation
1. Since identifiers are strings consisting of letters, digits, and the underscore character, we can declare a identifier name containing special character (for example *), which results in error.
Section 10.2 Strings
1. As a string may not contain EOF or null (character \0), we can try to intialise string to either of these.
2. As strings cannot cross file boundaries, if we omit closing double quotes ", it results in a error since it reaches EOF.
Section 10.4 Keywords
1. Using keyowrds for variable names results in error.
(Except for the constants true and false, keywords are case insensitive. So using a keyword as ClAss does not generate an error.)