CS3071 – Exercise 6 Report

Edward Bergman
#15335633

Task:

Our task for this exercise was to implement switch statements into the language Tastier.

Specification:

The switch statement must take an identity as the Switch ‘obj’. This obj’s value will be compared to each case ‘Primary’ value in descending order of declaration of the case statements (i.e. compare to first case statement, then second...). You can declare as many case statements as you like. If these values match, we will enter this ‘case’ and execute any statements declared within. Upon finishing this set of statements, we will enter the next ‘case’ and execute any statements declared within and so on until we have reached the end of the switch statement. If the ‘obj’ value does not match any ‘case’ value, we will not execute any code within the switch statement.

Optionally we can declare a ‘break’ statement at the end of each ‘case’. If this is included, once all statements within a ‘case’ have been executed, we will jump to the end of the switch statement and not execute anymore ‘cases’.

Optionally we can decalre a single ‘default case’ after all cases with a specified value. If the ‘obj’ value fails to match with any ‘case’ values, the ‘default case’ will be entered and the code within executed. We can also add a ‘break’ statement at the end of the default case but this is purely aesthetic.

Method:

Firstly, as with the other exercises, we must begin by designing a regular expression to match a Switch statement. This was relativly straight forward.

```
Switch = “switch” Ident ‘{‘
            {
            “case” Primary ‘:’
                  { Stat }
                  [ “break” ‘;’ ]
            }
            [
            “default” ‘:’
                  { Stat }
                  [ “break” ‘;’]
            ]
      ‘}’
```

This allows us to match the switch statement as specified in the specification. We do however limit ourselves to some degree by having an ‘Ident’ after the “switch” keyword. We are not able to to specify a value stored within an array directly i.e. arr[2]. Instead we must first assign it to a variable and then use that variable within the switch statement if we wish to check a specified array index.

Next we must generate the associated code to provide the implementation. We load in the object's value before each case instead of just once as the 'Primary' being evaluated could overwrite all registers. The next issue is where to place labels such that we can move correctly around the generated code.

To help break down this problem we identify where the labels must go and why. For each 'case', we must have a label before the values are loaded in and checked. We also require a label before each set of potential statements. To understand why, we must look at the case where a 'break' is omitted in a 'case'. When the statements in this case have finished executing, we must jump to the next set of statements as the 'break' is not present, this requires a label to jump to of course.

```
      <caseLabel>
'case' Primary ':'
      <statementLabel>
      { Stat }
      ...
      ...
```

To jump from case to case, we can simple use *gen.Branch(caseLabel + 2).* This is because +1 will lead us to statementLabel, +2 will lead us to the next caseLabel. Likewise for jumping from statement to statement.

We must also look at the default case. The key feature to the default case is that it has no comparison. To help unify the default case with a regular case, we have both labels generated before the default statements. This helps the last checked case behave similar to previous cases i.e.

```
if(case.primary.value != obj.value)
      jump to next case
```

It also helps that if the 'break' ';' is omitted from the last case, the program counter can jump to the next set of statements at 'statementLabel'. In the case that the default case is not included, the caseLabel and statementLabel are still there to prevent the last 'case' from jumping to an unintended label.

```
      <caseLabel>
      <statementLabel>
'default' ':'
      { Stat }
      [ 'break' ';' ]
```

The last point to explain is that the keyword 'default' and the optional 'break' in the default case generate no code as the default is essentailly some more statements that should be skipped past if they're not meant to be run. To skip past this or any other cases, we simply include an 'endLabel' at the end of the entire switch statement, at the '}' of the production that can be jumped to if neccessary.

Remarks:

There are not many remarks to make during this exercise as for the most part there wasn't much room for variation. The only two remarks that could be made are in the decision of the functionality of including/omitting break in each 'case' and the default case.

We chose to implement a more familar switch syntax and meaning, such that omitting break will have the code execute the next set of statements, otherwise ‘break’ would have no semantic meaning.

Learnt:

What was learnt from this exercise was how to think about placing labels in the generated code before deciding on how to actually reach those labels. Once the labels are in place, as long as there is a structure to the code, there is a structure to reaching the label required. In this case this simply involved basic addition.