Work log

1. Decompose statements to match different cases in scheme S(Statements);
2. Decompose Expression to match different cases in scheme SExp(Expression)
   a. Create a scheme to handle the case of the addition between an integer and a pointer and handle the function call case that appears in SExp(Expression),
   b. Create scheme ReadPara(ExpressionList) to take in actual parameters.
3. Handle the while loop case in scheme SWhile(Expression, Statement, Statements) and create Three labels(LTest, LTrue, LFalse) for the Expression, Statement and Statements
   a. Handle the condition test in scheme STest(Expression) for the Expression that appears in SWhile.
   b. Create a helper scheme SStat(Statement) to get rid of the {} turning Statement to {Statements} and lead the process to S(Statements).

Details of some schemes

1. SExp(Expression)

When handling the function call, inside SExp(⟦ f (⟨ExpressionList⟩) ⟧), I first push [R0-R3] values on stack to prepare the registers for the acutal parameters, let the scheme ReadPara(ExpressionList) move the paramters into [R0-R3], use "BL f" to jump to the callee with actual paramters stored in [R0-R3], and then Move R0, which is supposed to contain the return value, to the register that previously is assigned to store the result of the function call and pop the original [R0-R3] values from the stack.

2. ReadPara(ExpressionList)

When handling the actual parameter passing, there is a special case with a string parameter. I first create a label for the string in ReadPara and store the label in the attribute ↓s(label) then pass the attribute ↓s(label) down to StringPara(String) and use the label to the directive "DCS 'This is a string'" and let the address &label stores in the register that is assigned to hold the parameter.

3. ⟦ TA ⟨Expression⟩ ⟨Expression⟩ ⟨Operator⟩ ⟧

This scheme is created specially for the case of addition between an integer and a pointer. So what I did is match the case with different types between two Expression and when these two are the same, then do the usual addiont, but when one of them is a pointer then first use scheme ⟦ SIZEOF ⟨Identifier⟩ ⟧ to get the size value and multiply it with the integer and finally add the new value to that pointer.

4. SWhile(Expression, Statement, Statements)

when matching the case with a while loop inside S(Statements), create three labels ⟦LTest⟧, ⟦LTrue⟧, ⟦LFalse⟧ first and then pass them down by attributes ↓test(⟦Ltest)↓true(⟦LTrue⟧)↓false(⟦LFalse⟧), in this way, SWhile(...) knows how to label its three basic blocks and inside SWhile(...), again pass attributes to STest(), SStat(), therefore, STest() knows where to jump to and SStat() knows how to jump back to condition-test LTest.

5. STest(Expression)

This scheme only accept two cases: id or *id.

with id case, id could be a pointer or an integer, but when a pointer is NULL or an intger is zero, I suppose they both should be represented by 0 in the machine, so I compare the regiter that is holding id value to #0.

with *id case, *id should represent a string, since string is zero-terminated, first I load the string value by LDR ⟨Reg#1⟩, [⟨Reg#2⟩, #0] by assuming that Reg#2 holds the address of the starting character of the string, and again, I still compare the Reg#1 to #0.

Performed Tests

I use samples strlen.MC and strcpy.MC as my tests, and also create a test.MC to show that when there are multiple while loops, my compiler will create associated LTest, LTrue and LFlase for them and you can return from a while loop.
When doing dummpy = puti(strlen(input)) in strlen.MC, I get two consecutive STMPD instructions and two distanced LDMFD instructions following behind, which shows that calling a function inside a function works fine.

Further Work

The current compiler does not handle the if statement case and the condition test of while loop only takes care of id and *id. And the compiler following scheme S(Statements) only does print out instruction along the way with one pass so it does not do any type checking.
Further work should rely on expand more cases for pattern matching and add the type checking with first analyze Statements and boils it down to different sorts to get the types and then synthesize the types back to the higher level of the syntax tree to have more information when dealing with different types computation.