Code Generation Plan

11/22/2015

Basic Approach:

The code generation process is split into two stages. In the first stage, it scans the whole program and collects those members defined in the interface, in order to define there address offset in the interface body as well as the size of the interface. For functions, it will use the function name as the label and extracts its body structure for ARM code. So the first stage basically deals with the global declarations in the whole program. The above extract information will form an initial environment which will be passed down to each node of the program. In the second stage, it generates the ARM code for expressions and statements it meets, using the passed down environment when necessary.

Calling Conventions:

R0-R3 are for parameters, and R0 are usually stores the return result if any exists. LR are for return address. The caller manages R0-R3 and save a space for the return value. The callee manages the R4-R12. Before execution, the callee saves the states of the register it control and set R12 as the frame pointer, where the stack pointer starts growing towards lower address.

First Stage:

A subscript program is made up of Units, which is a sequence of Unit. So the ARM32 presentation will be a list of Unit according to their appearance in the

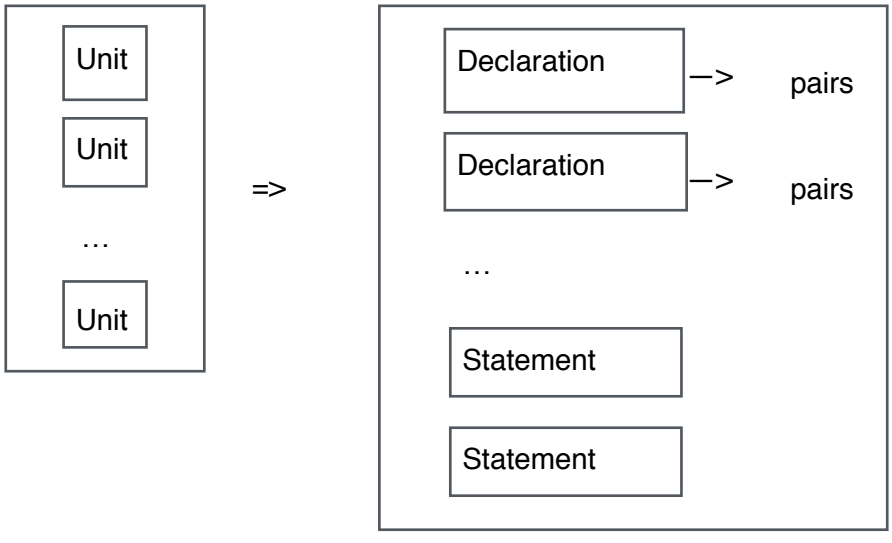


Figure 1

program. And each Unit is consist of a sequence of declarations and statement, show in Figure 1.

In each declaration, if it consists of an interface, it outputs a top level environment containing the members within the declarations. In the meantime, their addresses will be calculated and stored for later use in the second stage as show in Figure 2 (according to their appearance in order). If the declaration consists of a function, the name will be used as the label and ARM generates the CallSignature and its body structured in Figure 3. Here, the top level environment is represented as a pair which the first is a member name and the second is a member type.

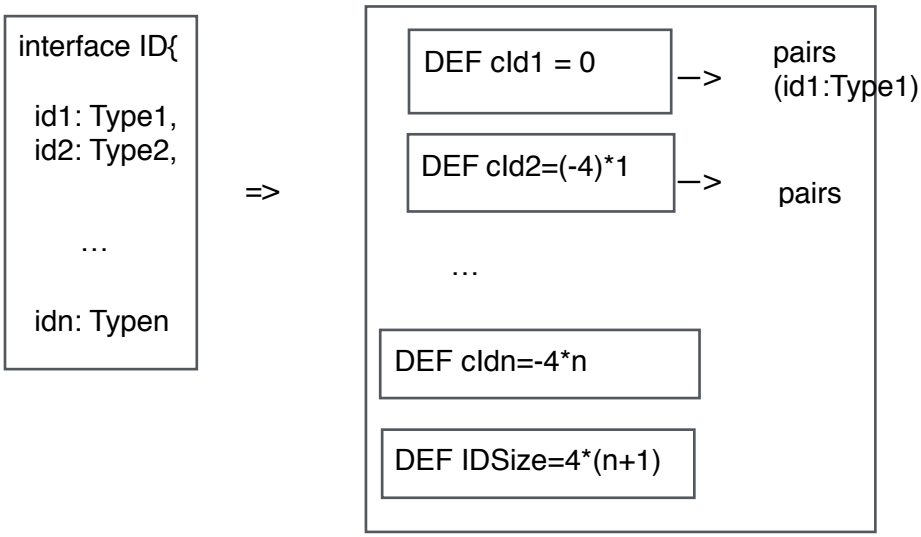


Figure 2

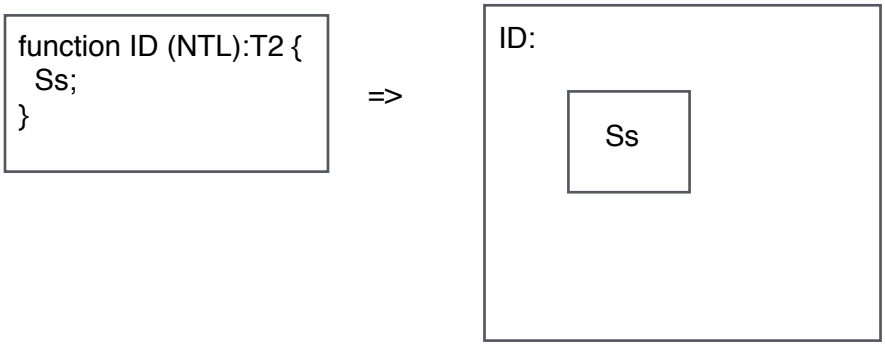


Figure 3

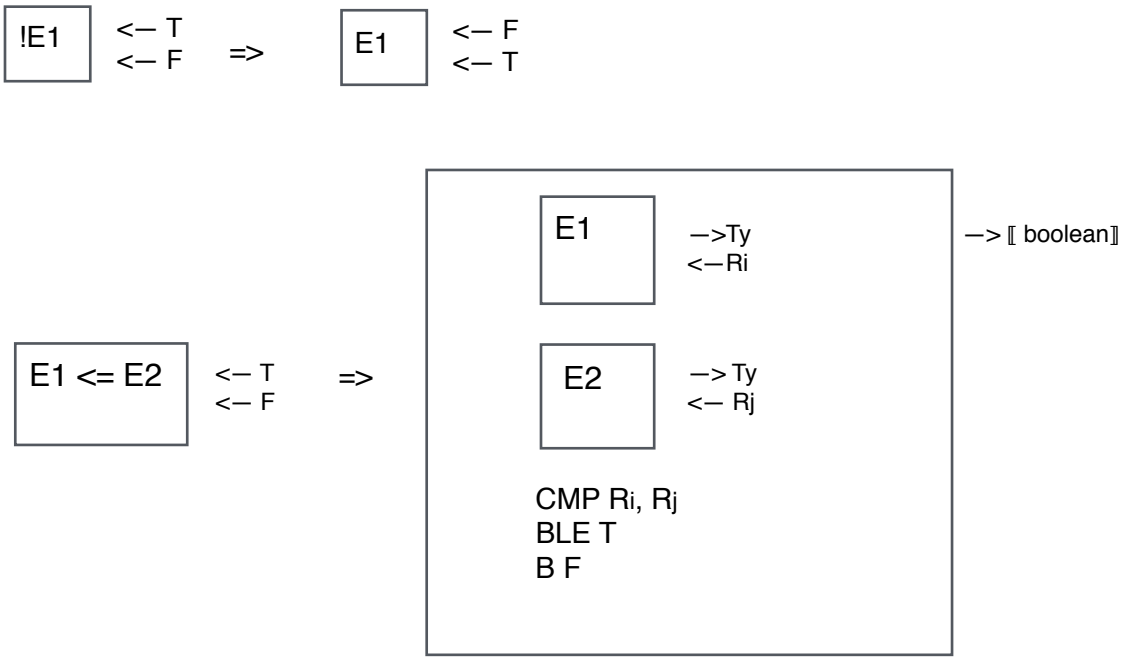
Second Stage:

With the pairs information we have in the first stage, we can generate ARM code for statement, expressions and other subscript structs.

1) Expression

Expressions are divided into two parts, where one is boolean expression containing conditions that involves jumps and the other is value expression that outputs the type(consider the + operator) of the expression, representing by "Ty" in the box. ARM code for boolean expressions and value expressions are in Figure 4 and Figure 5 respectively.

For boolean expressions, there are two labels T(represents true) and F(represents false) inherited into the expression. The R means the register where the expression value has been loaded.



Here, the case is the same for >=, <, >, == and !=. The differences is the BLE label will be changed to BGE, BLT, BGT, BEQ and BNE, respectively.

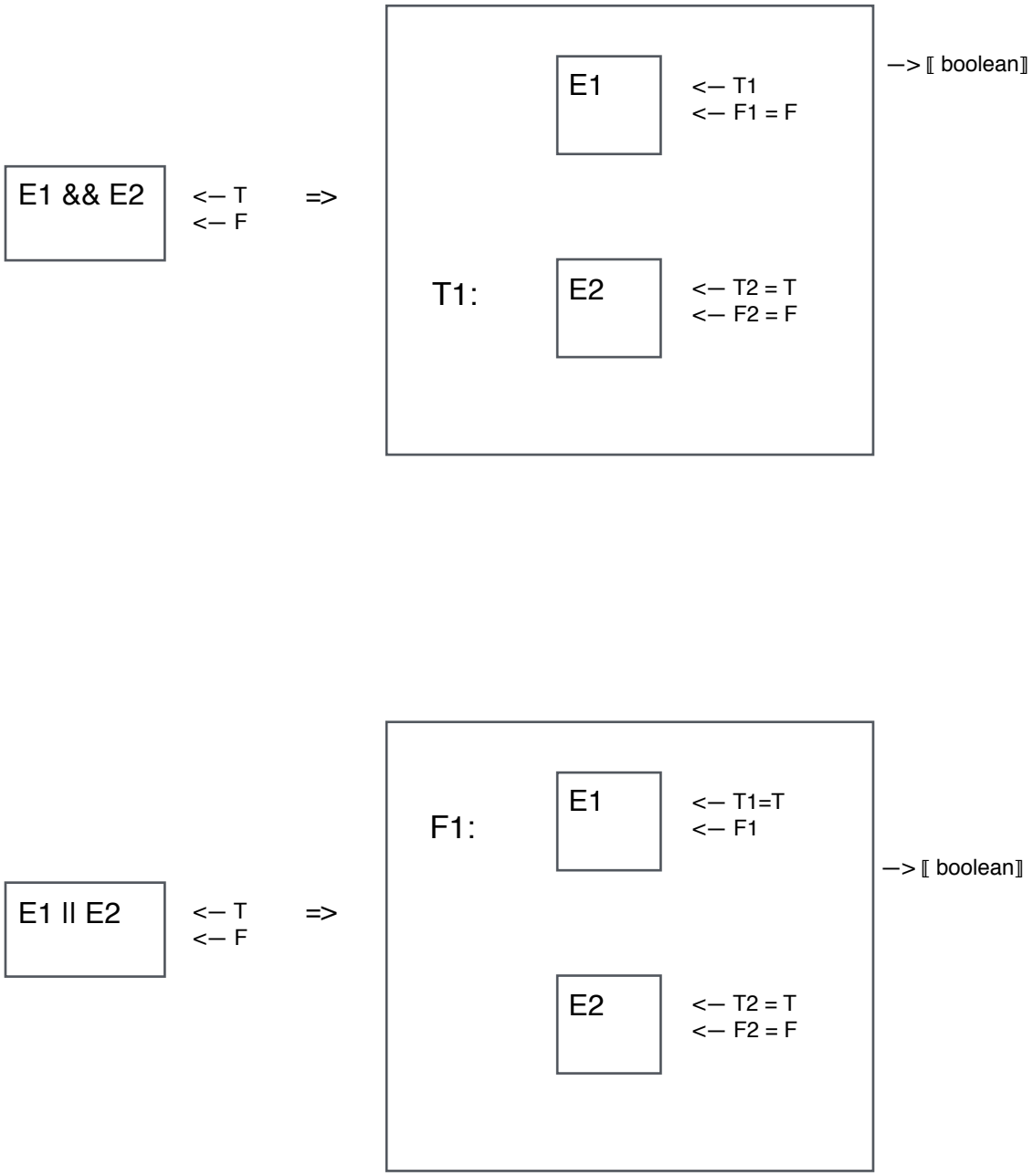
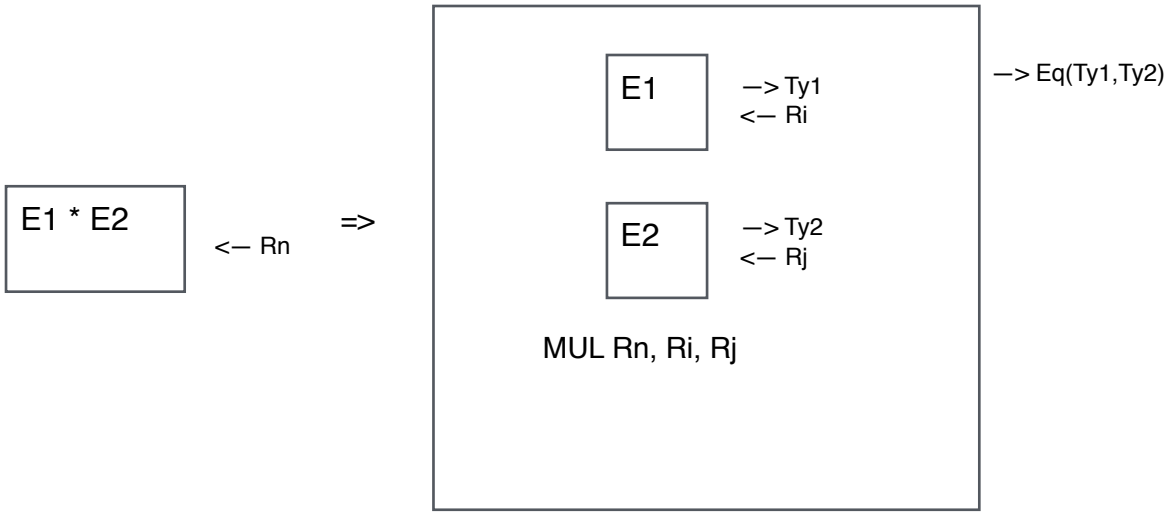
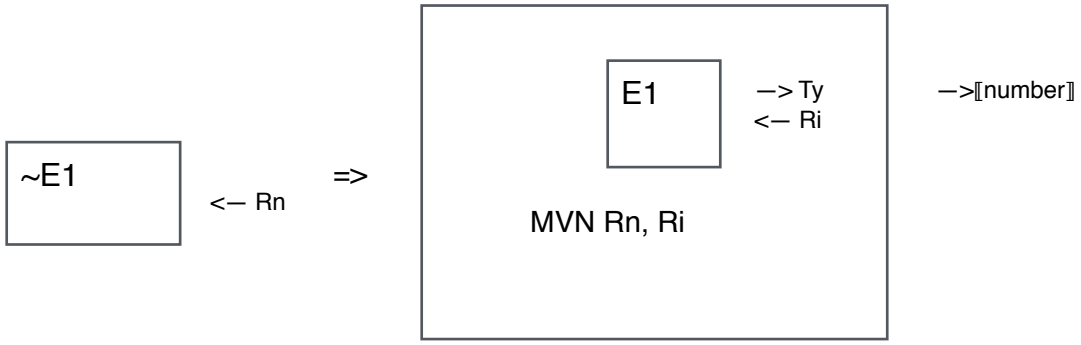
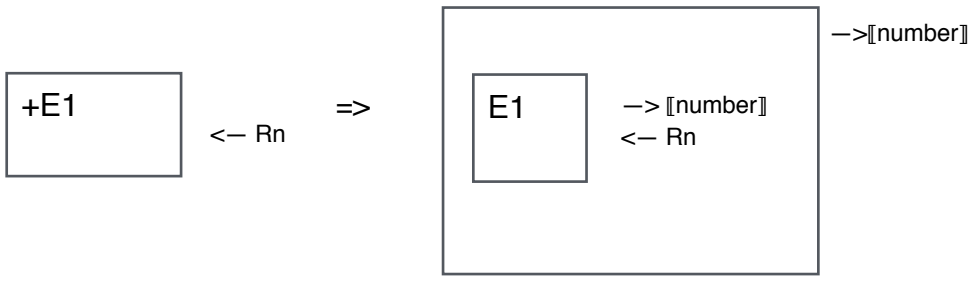

E1 && E2
<— T
<— F
=>
E1
<— T1
<— F1 = F
T1:
E2
<— T2 = T
<— F2 = F
—> ⟦ boolean⟧
E1 ll E2
<— T
<— F
=>
F1:
E1
<— T1=T
<— F1
E2
<— T2 = T
<— F2 = F
—> ⟦ boolean⟧

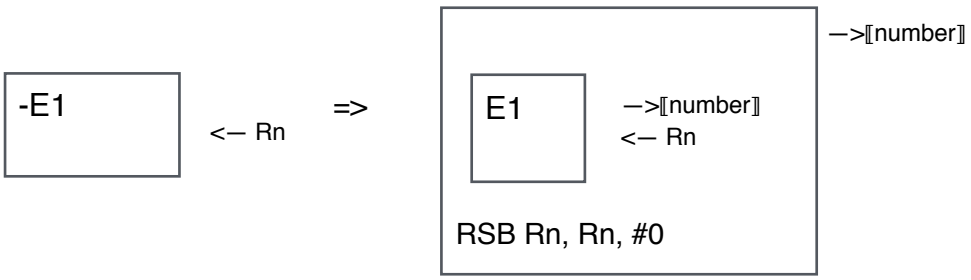
Figure 4

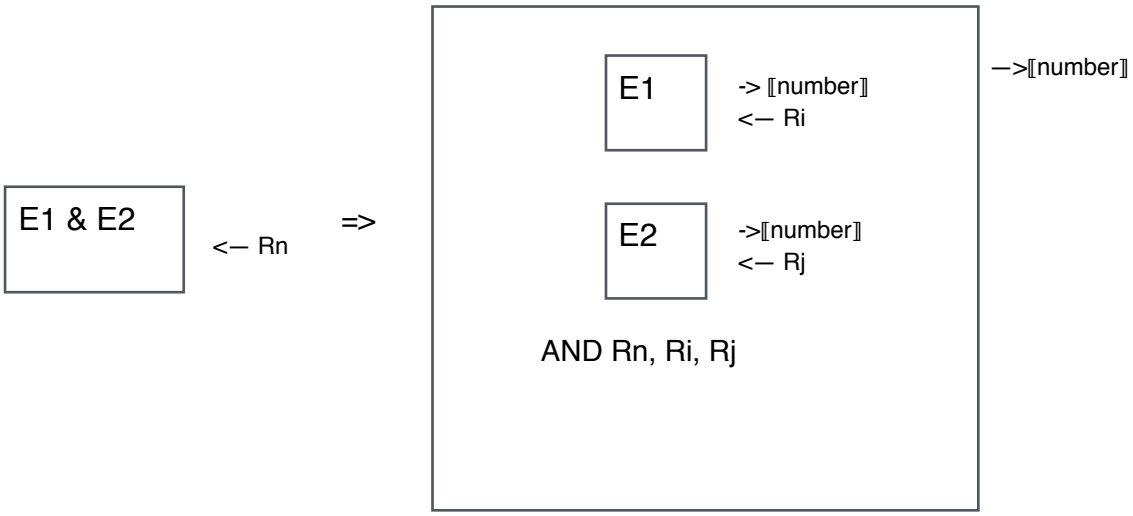


Here, the ARM code is similar for /, +, - and %, except that the MUL will be replaced by divide(library function), ADD, SUB and remainder(library function). Another difference is that besides multiplication, if the second operand is an immediate constant, there's no need to load it into register. Eg, ADD Rn, Ri, #a.

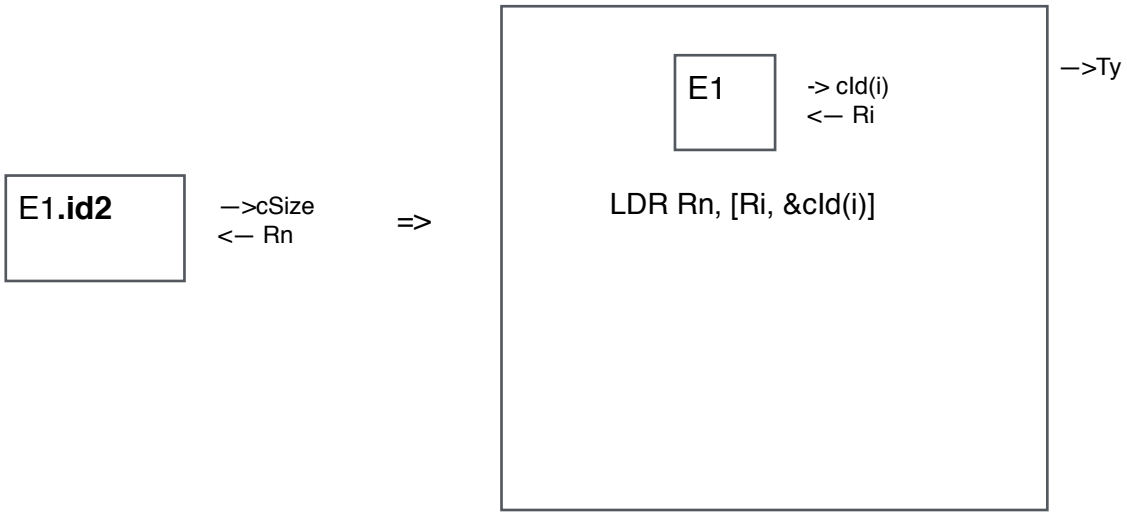


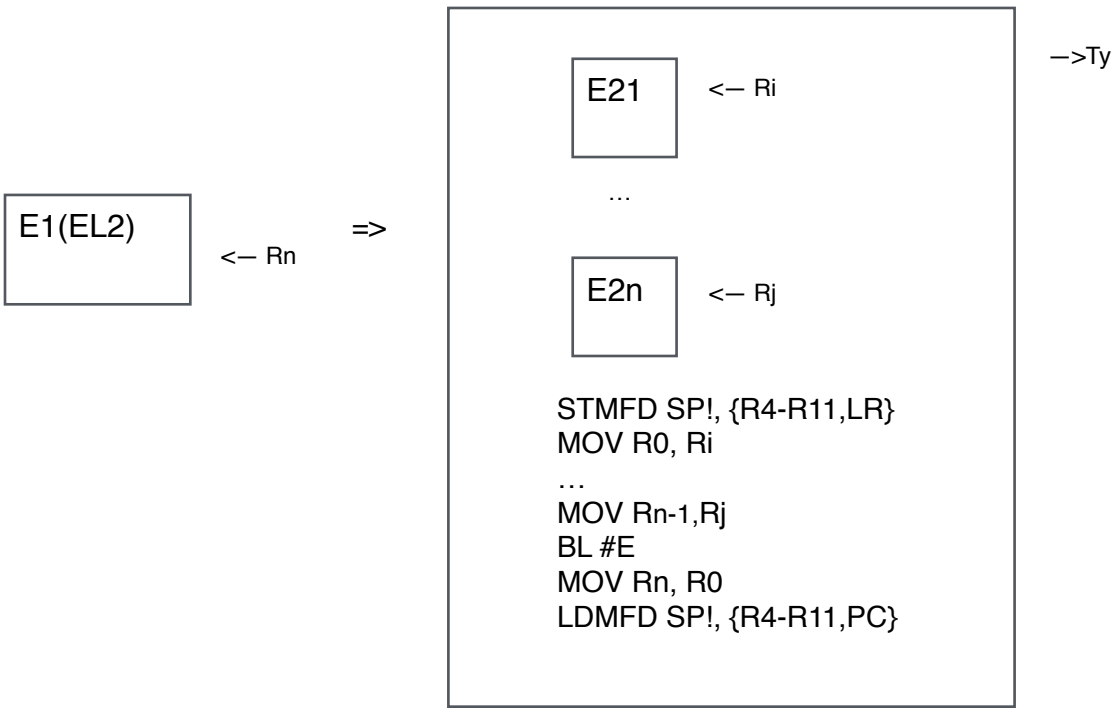




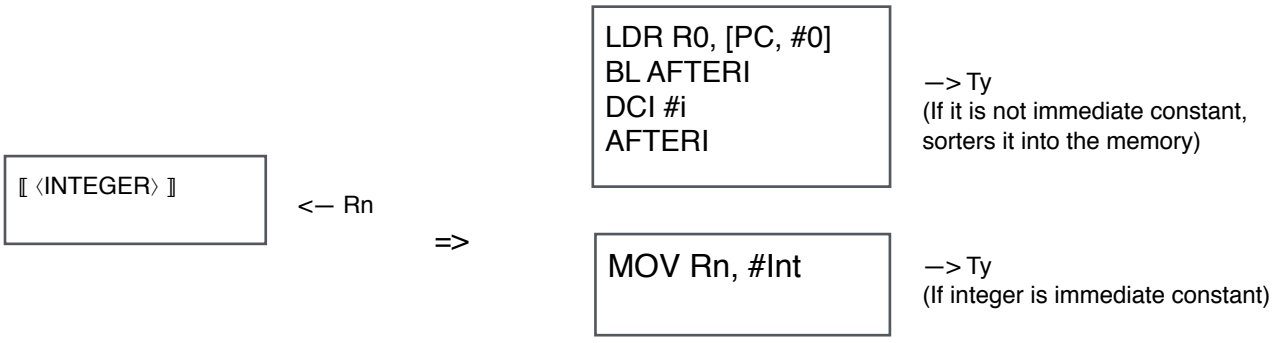


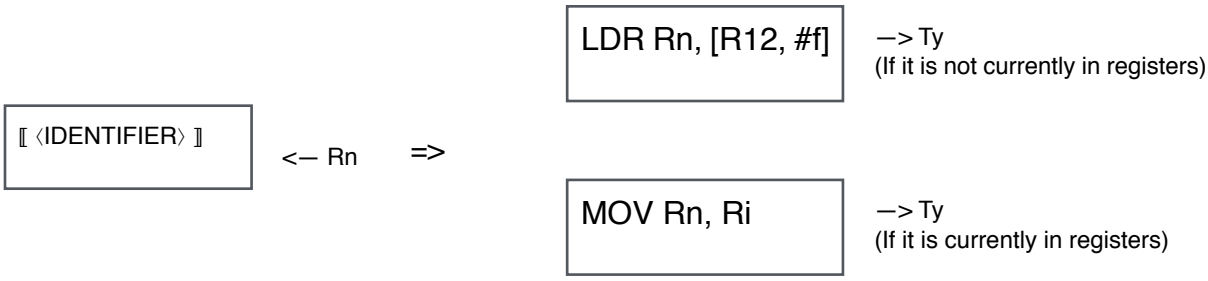
Here, the ARM code for E1|E2 is similar, only replaced the AND with OR.

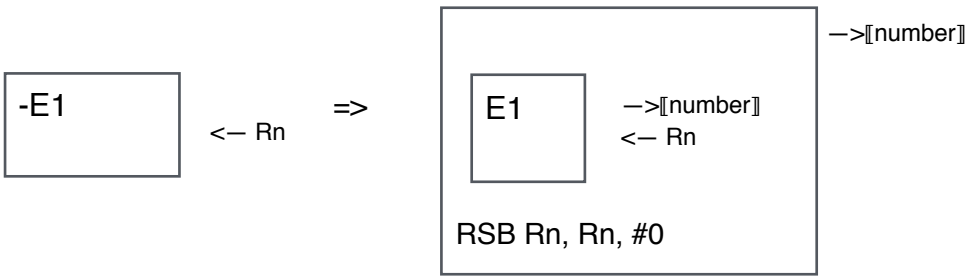




Here, E1 is usually a function identifier, represented as a label #E in the box, where EL2 = (EL21, EL22, …, EL2n). Reflects of the assumption made, here n is no greater than 4.





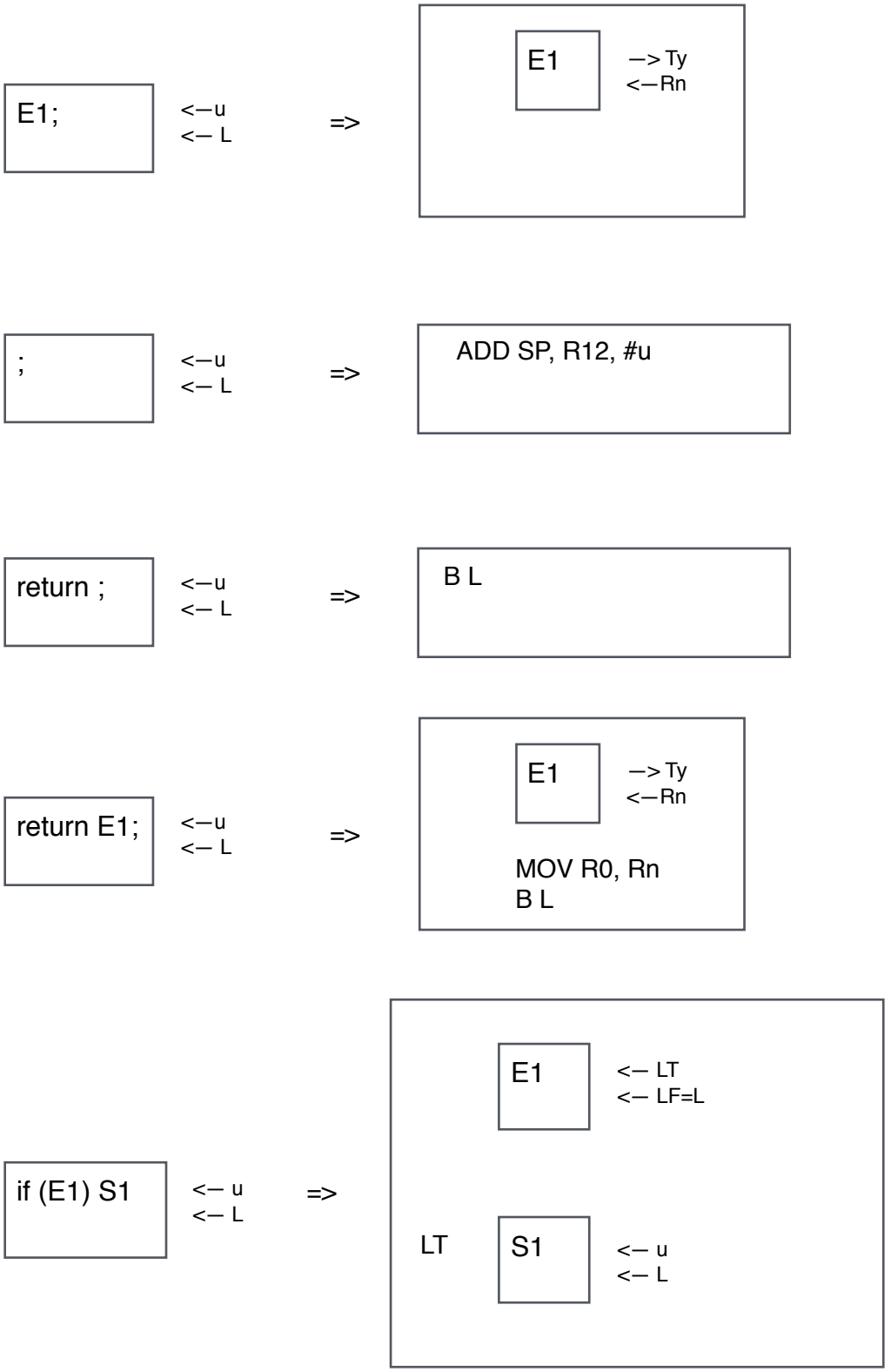


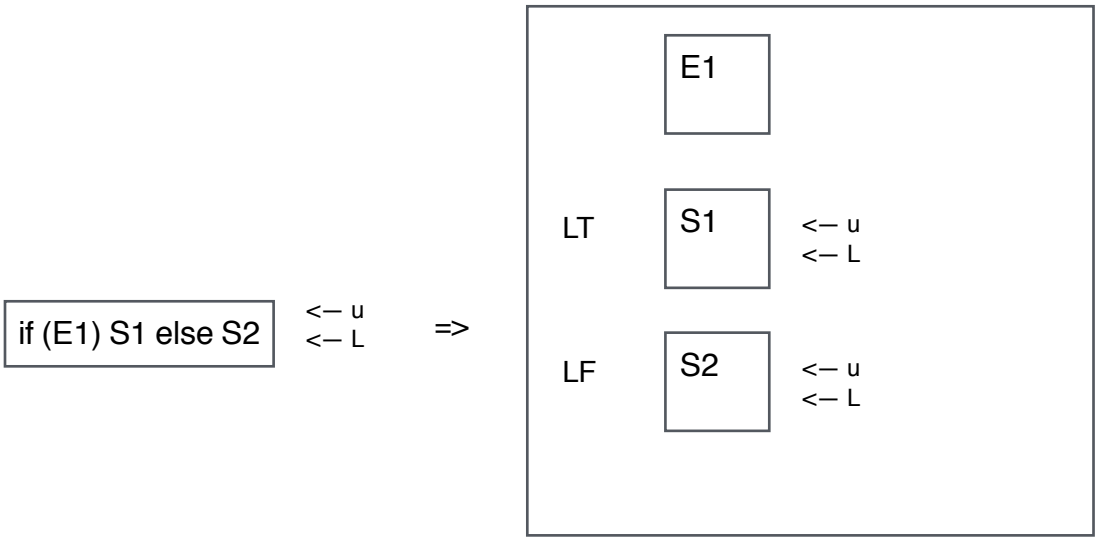
String invokes heap allocation, which will call the library function and it is used DCB to load it into the register.

2) Statement

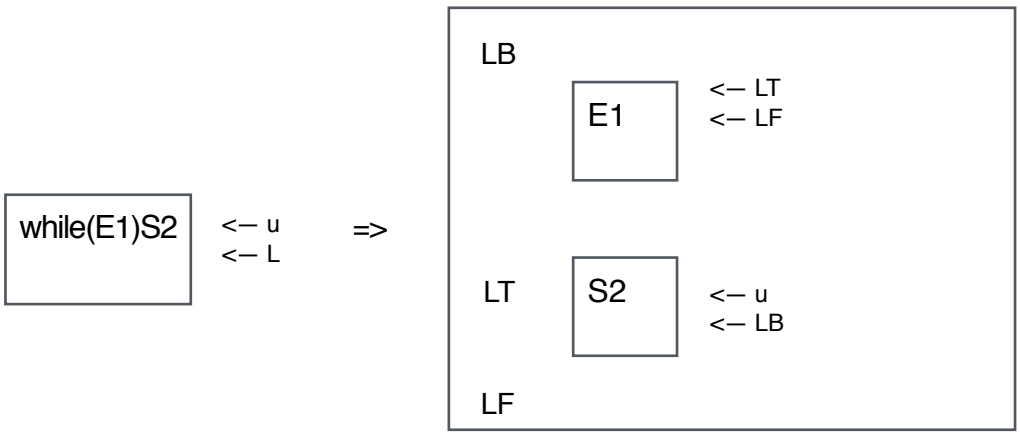
The "scale" of the Statement affects the stack and the stack pointer pointing position.

So it needs some kind of a "label" from the outer context to tell Statement that the next unused location in the current activation record. We name the label u. And there is also another label inherited from the outer context to tell the Statement where is goes after it finishes. We name it L.


E1;
<—u
<— L
=>
E1
—> Ty
<—Rn
;
<—u
<— L
=>
ADD SP, R12, #u
return ;
<—u
<— L
=>
B L
return E1;
<—u
<— L
=>
E1
—> Ty
<—Rn
MOV R0, Rn
B L
if (E1) S1
<— u
<— L
=>
E1
<— LT
<— LF=L
LT
S1
<— u
<— L

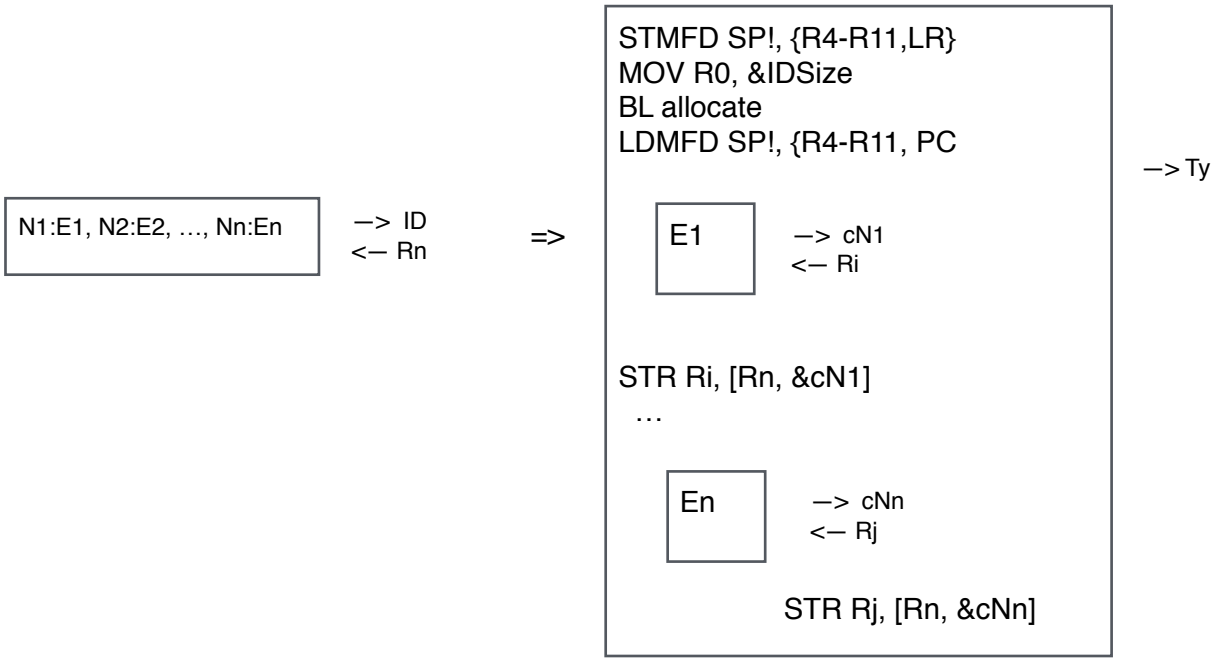


Here, after statement S2 executed, it will jump to LB and evaluates the boolean



expression E1 again.

3) KeyValueList



Here, the Rn means the address in register Rn. ID means the interface name, where we can use the environment in the first stage to determine its size in the heap.

Assumptions:

1. There is at most four arguments in a functions.
2. All the type correctness are assured here for simplicity.
3. The generated code will free heap allocations "automatically", which is not show in the explicit code.