COMP520 Milestone 3

CHEN Hongji 260610116 Xiru Zhu 260498154 Tianzi Yang 260599365

The target language our group chose is the native assembly code, and it will be generated in Intel Syntax, and can be assembled by NASM(@http://www.nasm.us).

Register Allocation

In order to link with C library, our group will use the registers following the System V calling convention (@http://www.x86-64.org/documentation/abi.pdf). %rax register will always be reserved as a expression evaluation temporary register for reference type, int, rune and bool, and %xmm0 register will be reserved as a temporary register for float expression evaluation. The first six parameters will be passed within registers, and others will be pushed onto stack.

Memory Frame and Allocation

Every memory frame is started by push %rbp register onto the stack, and we reserved $16 * \lceil \#pram/2 \rceil$ bytes immediate follow that for the pram shadow space, and every evaluation of the expression adjust the %rsp register respectively (Note : stack should be 16-byte aligned).

The primitive types are stored directly in the stack, while reference types are stored as a pointer to a object within the memory pool which is managed by the GoLite library. The logic expression is generated as pattern shown in class slides (use labels and control jumps), and for each expression evaluation we reserve a 16-byte memory stack space, and free then immediately after the evaluation. Unlike the implementation in class, after each evaluation, we do not have stack size increase by one for expression (as we store the evaluated value in the return registers).

We decide to start working from the reference type. For example the for the following GoLite code:

```
var arr [10][1]string
var arrString [1]string
{
        var str string = "demo String :D"
        arrString[0] = str
}
arr[0] = str
arr[0] = str
```

We will translate into assembly code as,

```
;; [rbp - 8] arr, [rbp - 16] arrString, [rbp - 24] str
;; constString: db       "demo String :D"
mov     rdi, 10
mov     rsi, GoLite_Type_ReferenceArray
call    _GoLite_Array_Alloc
mov     qword [rbp - 8], rax
call    _GoLite_Array_Scope
mov     rdi, 1
mov     rsi, GoLite_Type_ReferenceString
call    _GoLite_Array_Alloc
mov     qword [rbp - 16], rax
call    _GoLite_Array_Scope
```

```
mov     rdi, [rel constString]
call    _GoLite_String_ConstDump
mov     qword [rbp - 24], rax
call    _GoLite_String_Scope
mov     rdi, qword [rbp - 16]
mov     rsi, 0x0
mov     rdx, qword [rbp - 24]
xor     rax, rax
call    _GoLite_Array_Store
mov     rdi, qword [rbo - 24]
call    _GoLite_String_UnScope
mov     rdi, qword [rbp - 8]
mov     rsi, 0x0
mov     rdx, qword [rbp - 16]
xor     rax, rax
call    _GoLite_Array_Store
mov     rdi, qword [rbp - 8]
mov     rsi, 0x0
mov     rdx, qword [rbp - 16]
xor     rax, rax
call    _GoLite_Array_Store
mov     rdi, qword [rbp - 16]
call    _GoLite_Array_UnScope
mov     rdi, qword [rbp - 8]
call    _GoLite_Array_UnScope
```

Our current concern is the efficiency of such implementation. Although such implementation will help the garbage collector free the memory efficiently and allow the program to detect the array overflow, but it will have a long overhead on array assignment. For now, we have perform a 10^8 array access benchmark test on our GoLite program, and a equivalent C program. The C program will take around 0.5 seconds while our GoLite program will need around 1 seconds to complete the task. (Still have large space for improvement)