Compiler Milestone 3

Abir Rajbongshi - 210273, Aditya Bangar - 210069, Pratham Sahu - 210755

Course Instructor : Dr Swarnendu Biswas

Abstract—This is the report for the final milestone of CS335-Compiler Design course.

1. Compiling the Compiler

The directory structure is as follows:

```
/
├── Makefile
├── run.sh
├── run_x86.sh
├── src/
│   ├── allocmem.s
│   ├── lexer.l
│   ├── node.cpp
│   ├── parser.y
│   ├── print.s
│   ├── symbol_table.cpp
│   └── x86.cpp
└── testcases/
    ├── testcase1.py
    ├── testcase2.py
    ├── testcase3.py
    ├── testcase4.py
    └── testcase5.py
```

commands to run

1. `make`
2. `./run.sh testcasees/testcase1.py`
3. `gcc -o a.out x86_code.s -no-pie`
4. `./a.out`

The above commands will generate the `x86_code.s` file which contains the assembly code for the given python file along with the other files from the previous milestones. The `a.out` file will be the executable file for the given python file.

2. Language Features Implemented

2.1. Minimum Features to be implemented :

- **Arithmetic Operations** : Addition(+), Subtraction(-), Multiplication(*), Division(/, //), Modulus(%)
- **Relational Operations** : Less than(<), Less than or equal to(<=), Greater than(>=), Greater than or equal to(>=), Equal to(=), Not equal to(!=)
- **Logical Operations** : And(`and`), Or(`or`), Not(`not`)
- **(**Bitwise Operations) : And(&), Or(|), Xor(ˆ), Left shift(«), Right shift(»)
- **Assignment Operations** : Simple assignment(=), Add and assign(+=), Subtract and assign(-=), Multiply and assign(*=), Divide and assign(/=), Modulus and assign(%=), Bitwise and assign(&=), Bitwise or assign(|=), Bitwise xor assign(^=), Left shift and assign(«=), Right shift and assign(»=)
- **Control Structures** : `if-else`, `if-elif-else`, `while`, `for`, `break`, `continue`
- **Data Types** : Integers, Booleans, Strings, Arrays
- **Functions** : Function definition, Function call, Function return, Function arguments, Function recursion, library functions (`len`, `print`, `range`)
- **Classes** : Class definition, Class instantiation, Class methods (static and non-static), Class inheritance, Class attributes, Objects, Self
- **Multilevel Inheritance** : Implemented multilevel inheritance in the language.
- **String Comparison** : Implemented string comparison in the language.

2.2. Additional Features Implemented :

- **Functions inside Functions** : Cases like

```
print(func(a, b))
print(func1(a, b) + func2(func1(a, b) + c))
range(len(arr))
```

  are handled.
- **Array Handling and Extensions** : Cases like

```
arr = [1, 2, 3, 4, 5]
# passing arrays as argument to function.
func(arr)
# returning arrays from functions.
arr2 = function_which_returns_array()
a : A = A()
b : B = B()
c : C = C()
arr3 = [a, b, c] # array of objects.
x : A = arr3[0]
x.func() # function call on object taken from
```

  are also handled.
- **Passing Objects as Function Arguments** : Objects can be passed as arguments to functions.
- **List of objects**: We can have a list of non primitive data types(ex objects of a ckass)

3. Some Assumptions and Important Points

Some of the assumptions which we make while implementing the language are as follows:

- You cannot call len for an array that has been passed as a parameter. If you wish to use len, you can pass it as another parameter. len must be caled only from the scope of the function where it is initialised
- We do not take care of arrays that have just one element. It does not make sense to support it, thus for ease of implementation of our compiler, we do not support it.

4. Examples

Example 1

```
1  class Dog:
2      def __init__(self, name: str) -> None:
3          self.name: str = name
4
5      def speak(self) -> None:
6          print("woof")
7
8  class Pet(Dog):
9      def __init__(self, name: str, breed: str) -> None:
10         self.name: str = name
```

```

class Labrador(Pet):
    def __init__(self, name: str, breed: str) -> None:
        self.name: str = name
        self.breed: str = breed


def main() -> None:
    labrador:Labrador = Labrador("Max", "Labrador
     Retriever")
    labrador.speak()          # Output: Max says Woof!


if __name__ == "__main__":
    main()
```

Code 1. Multilevel Inheritance

```
.data
  integer_format: .asciz "%d\n"
  string_format: .asciz "%s\n"
  .str0:
    .string "woof"
  .str1:
    .string "Max"
  .str2:
    .string "Labrador Retriever"
  .str3:
    .string "__main__"
  .global main
  .text
Dog.__init__:
  pushq %rbp
  movq %rsp, %rbp
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  subq $56, %rsp
  movq 16(%rbp), %rdx
  add $0, %rdx
  movq %rdx, -40(%rbp)
  movq -40(%rbp), %rax
  movq 24(%rbp), %rdx
  movq %rdx, (%rax)
  addq $56, %rsp
  popq %r15
  popq %r14
  popq %r13
  popq %r12
  popq %rbx
  popq %rbp
  ret
Dog.speak:
  pushq %rbp
  movq %rsp, %rbp
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  subq $64, %rsp
  lea .str0(%rip), %rax
  movq %rax, -48(%rbp)
  movq $0, %rdi
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq -48(%rbp)
  call global.print
  addq $8, %rsp
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq $1, %rdi
  addq $64, %rsp
  popq %r15
  popq %r14
  popq %r13
  popq %r12
  popq %rbx
  popq %rbp
  ret
Pet.__init__:
  pushq %rbp
  movq %rsp, %rbp
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  subq $56, %rsp
  movq 16(%rbp), %rdx
  add $0, %rdx
  movq %rdx, -40(%rbp)
  movq -40(%rbp), %rax
  movq 24(%rbp), %rdx
  movq %rdx, (%rax)
  addq $56, %rsp
  popq %r15
  popq %r14
  popq %r13
  popq %r12
  popq %rbx
  popq %rbp
  ret
Labrador.__init__:
  pushq %rbp
  movq %rsp, %rbp
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  subq $72, %rsp
  movq 16(%rbp), %rdx
  add $0, %rdx
  movq %rdx, -40(%rbp)
  movq -40(%rbp), %rax
  movq 24(%rbp), %rdx
  movq %rdx, (%rax)
  movq 16(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -56(%rbp)
  movq -56(%rbp), %rax
  movq 32(%rbp), %rdx
  movq %rdx, (%rax)
  addq $72, %rsp
  popq %r15
  popq %r14
  popq %r13
  popq %r12
  popq %rbx
  popq %rbp
  ret
main:
  pushq %rbp
  movq %rsp, %rbp
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  subq $120, %rsp
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq $32
  call memalloc
  add $8, %rsp
  movq %rax, -48(%rbp)
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
```

```
  movq $1, %rdi
  lea .str2(%rip), %rax
  movq %rax, -64(%rbp)
  lea .str1(%rip), %rax
  movq %rax, -80(%rbp)
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq -64(%rbp)
  pushq -80(%rbp)
  pushq -48(%rbp)
  call Labrador.__init__
  addq $24, %rsp
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq -48(%rbp), %rdx
  movq %rdx, -96(%rbp)
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq -96(%rbp)
  call Dog.speak
  addq $8, %rsp
  movq %rax, -112(%rbp)
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq $60, %rax
  xorq %rdi, %rdi
  syscall
global.print:
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %rbx
    pushq   %r12
    pushq   %r13
    pushq   %r14
    pushq   %r15

    # Ensure stack alignment for printf
    movq    %rsp, %rax          # Store current rsp
    andq    $15, %rax           # Check lower 4 bits,
     rsp should be 16-byte aligned before call
    jz      align_ok            # If zero, alignment is
      okay
    subq    $8, %rsp            # Adjust stack to be
     16-byte aligned
    movq    $1, -192(%rbp)        # Mark that we
     adjusted the stack

align_ok:
    # Check type in %rdi (1=string, otherwise integer)
    cmpq    $0, %rdi
    je      print_string

    # Setup for printing integer

    leaq    integer_format(%rip), %rdi # Load format
     string for integer
    movq    16(%rbp), %rsi              # Load integer
     value from stack
    xor     %rax, %rax                  # Float register
      count
    call    printf                      # Call printf
    jmp     cleanup                     # Jump to
     cleanup

print_string:
    # Setup for printing string
    leaq    string_format(%rip), %rdi  # Load format
     string for string
    movq    16(%rbp), %rsi              # Load string
     pointer from stack
    xor     %rax, %rax                  # Float register
      count
    call    printf                      # Call printf

cleanup:
    # Restore the stack if it was misaligned
    movb    -192(%rbp), %al               # Check if we
     marked the stack as adjusted
    testb   $1, %al
    jz      stack_ok
    addq    $8, %rsp                    # Restore
     original stack alignment

stack_ok:
    popq    %r15
    popq    %r14
    popq    %r13
    popq    %r12
    popq    %rbx
    popq    %rbp
    ret
memalloc:
    pushq %rbp
    mov %rsp, %rbp
    pushq %rbx
    pushq %rdi
    pushq %rsi
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15

    testq $15, %rsp
    jz is_mem_aligned

    pushq $0                  # align to 16 bytes

    movq 16(%rbp), %rdi
    call malloc

    add $8, %rsp              # remove padding

    jmp mem_done

is_mem_aligned:

    movq 16(%rbp), %rdi
    call malloc

mem_done:

    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %rsi
    popq %rdi
    popq %rbx
    popq %rbp

    ret
```

Code 2. The corresponding x86 Assembly of Example 1.

Output of Example 1

```
woof
```

Example 2

```
def main() -> None:
    arr: list[int] = [10, 7, 8, 9, 1, 5]
    arr[2]=3
    arr[4]=arr[0]
    i:int=0
    for i in range(len(arr)):
        print(arr[i])

```

```
if __name__ == "__main__":
    main()
```

Code 3. Arrays along with their referencing and dereferecing

```
.data
  integer_format: .asciz "%d\n"
  string_format: .asciz "%s\n"
  .str0:
    .string "__main__"
  .global main
  .text
main:
  pushq %rbp
  movq %rsp, %rbp
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  subq $224, %rsp
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq $48
  call memalloc
  add $8, %rsp
  movq %rax, -48(%rbp)
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq -48(%rbp), %rax
  movq $10, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rax
  movq $7, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rax
  movq $8, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rax
  movq $9, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rax
  movq $1, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rax
  movq $5, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rdx
  sub $48, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rdx
  movq %rdx, -64(%rbp)
  movq $2, %rdx
  imul $8, %rdx
  movq %rdx, -72(%rbp)
  movq -64(%rbp), %rdx
  add -72(%rbp), %rdx
  movq %rdx, -72(%rbp)
  movq -72(%rbp), %rax
  movq $3, %rdx
  movq %rdx, (%rax)
  movq $4, %rdx
  imul $8, %rdx
  movq %rdx, -88(%rbp)
  movq -64(%rbp), %rdx
  add -88(%rbp), %rdx
  movq %rdx, -88(%rbp)
  movq $0, %rdx
  imul $8, %rdx
  movq %rdx, -96(%rbp)
  movq -64(%rbp), %rdx
  add -96(%rbp), %rdx
  movq %rdx, -96(%rbp)
  movq -96(%rbp), %rax
  movq (%rax), %rdx
  movq %rdx, -112(%rbp)
  movq -88(%rbp), %rax
  movq -112(%rbp), %rdx
  movq %rdx, (%rax)
  movq $0, -128(%rbp)
  movq $6, -136(%rbp)
  movq $0, -144(%rbp)
L2:
  movq -144(%rbp), %rdx
  movq %rdx, -128(%rbp)
  movq -144(%rbp), %rbx
  movq -136(%rbp), %rcx
  cmp %rcx, %rbx
  jl L4
  jmp L1
L4:
  movq -128(%rbp), %rdx
  imul $8, %rdx
  movq %rdx, -184(%rbp)
  movq -64(%rbp), %rdx
  add -184(%rbp), %rdx
  movq %rdx, -184(%rbp)
  movq -184(%rbp), %rax
  movq (%rax), %rdx
  movq %rdx, -200(%rbp)
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq -200(%rbp)
  call global.print
  addq $8, %rsp
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
L3:
  movq -144(%rbp), %rdx
  add $1, %rdx
  movq %rdx, -144(%rbp)
  jmp L2
L1:
  movq $60, %rax
  xorq %rdi, %rdi
  syscall
global.print:
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %rbx
    pushq   %r12
    pushq   %r13
    pushq   %r14
    pushq   %r15

    # Ensure stack alignment for printf
    movq    %rsp, %rax          # Store current rsp
    andq    $15, %rax           # Check lower 4 bits, rsp should be 16-byte aligned before call
    jz      align_ok            # If zero, alignment is okay
    subq    $8, %rsp            # Adjust stack to be 16-byte aligned
```

```
    movq    $1, -192(%rbp)        # Mark that we adjusted the stack

align_ok:
    # Check type in %rdi (1=string, otherwise integer)
    cmpq    $0, %rdi
    je      print_string

    # Setup for printing integer

    leaq    integer_format(%rip), %rdi # Load format string for integer
    movq    16(%rbp), %rsi             # Load integer value from stack
    xor     %rax, %rax                 # Float register count
    call    printf                     # Call printf
    jmp     cleanup                    # Jump to cleanup

print_string:
    # Setup for printing string
    leaq    string_format(%rip), %rdi  # Load format string for string
    movq    16(%rbp), %rsi             # Load string pointer from stack
    xor     %rax, %rax                 # Float register count
    call    printf                     # Call printf

cleanup:
    # Restore the stack if it was misaligned
    movb    -192(%rbp), %al              # Check if we marked the stack as adjusted
    testb   $1, %al
    jz      stack_ok
    addq    $8, %rsp                   # Restore original stack alignment

stack_ok:
    popq    %r15
    popq    %r14
    popq    %r13
    popq    %r12
    popq    %rbx
    popq    %rbp
    ret
memalloc:
    pushq %rbp
    mov %rsp, %rbp
    pushq %rbx
    pushq %rdi
    pushq %rsi
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15

    testq $15, %rsp
    jz is_mem_aligned

    pushq $0                  # align to 16 bytes

    movq 16(%rbp), %rdi
    call malloc

    add $8, %rsp              # remove padding

    jmp mem_done

is_mem_aligned:

    movq 16(%rbp), %rdi
    call malloc

mem_done:

    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %rsi
    popq %rdi
    popq %rbx
    popq %rbp

    ret
```

Code 4. The corresponding x86 Assembly of Example 2.

Output of Example 2

```
10
7
3
9
10
5
```

Example 3

```
def string_comparison(string1: str, string2: str,n1:int, n2:int) -> sNone:
    if string1 == string2:
        print("The strings are equal.")
    else:
        print("String are not equal.")

def main():
    string_comparison("apple", "banana",5,6)
    string_comparison("python", "python",6,6)

if __name__ == "__main__":
    main()
```

Code 5. Strings and their comparison

```
.data
  integer_format: .asciz "%d\n"
  string_format: .asciz "%s\n"
  .str0:
    .string "The strings are equal."
  .str1:
    .string "String are not equal."
  .str2:
    .string "apple"
  .str3:
    .string "banana"
  .str4:
    .string "python"
  .str5:
    .string "python"
  .str6:
    .string "__main__"
  .global main
.text
global.string_comparison:
  pushq %rbp
  movq %rsp, %rbp
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  subq $120, %rsp
  movq 16(%rbp), %rbx
  movq 24(%rbp), %rcx
  movq %rbx, %rdi
  movq %rcx, %rsi
  call strcmp
  movsx %eax, %rax
  cmp $0, %rax
  je L2
  jmp L3
L2:
  lea .str0(%rip), %rax
  movq %rax, -80(%rbp)
  movq $0, %rdi
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq -80(%rbp)
  call global.print
  addq $8, %rsp
  popq %r11
  popq %r10
  popq %r9
```

```
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq $1, %rdi
  jmp L1
L3:
  lea .str1(%rip), %rax
  movq %rax, -112(%rbp)
  movq $0, %rdi
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq -112(%rbp)
  call global.print
  addq $8, %rsp
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq $1, %rdi
L1:
  addq $120, %rsp
  popq %r15
  popq %r14
  popq %r13
  popq %r12
  popq %rbx
  popq %rbp
  ret
main:
  pushq %rbp
  movq %rsp, %rbp
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  subq $120, %rsp
  lea .str3(%rip), %rax
  movq %rax, -48(%rbp)
  lea .str2(%rip), %rax
  movq %rax, -64(%rbp)
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq $6
  pushq $5
  pushq -48(%rbp)
  pushq -64(%rbp)
  call global.string_comparison
  addq $32, %rsp
  movq %rax, -80(%rbp)
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq $1, %rdi
  lea .str5(%rip), %rax
  movq %rax, -96(%rbp)
  lea .str5(%rip), %rax
  movq %rax, -104(%rbp)
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq $6
  pushq $6
  pushq -96(%rbp)
  pushq -104(%rbp)
  call global.string_comparison
  addq $32, %rsp
  movq %rax, -112(%rbp)
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq $1, %rdi
  movq $60, %rax
  xorq %rdi, %rdi
  syscall
global.print:
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %rbx
    pushq   %r12
    pushq   %r13
    pushq   %r14
    pushq   %r15

    # Ensure stack alignment for printf
    movq    %rsp, %rax          # Store current rsp
    andq    $15, %rax           # Check lower 4 bits, rsp should be 16-byte aligned before call
    jz      align_ok            # If zero, alignment is okay
    subq    $8, %rsp            # Adjust stack to be 16-byte aligned
    movq    $1, -192(%rbp)        # Mark that we adjusted the stack

align_ok:
    # Check type in %rdi (1=string, otherwise integer)
    cmpq    $0, %rdi
    je      print_string

    # Setup for printing integer

    leaq    integer_format(%rip), %rdi # Load format string for integer
    movq    16(%rbp), %rsi              # Load integer value from stack
    xor     %rax, %rax                  # Float register count
    call    printf                      # Call printf
    jmp     cleanup                     # Jump to cleanup

print_string:
    # Setup for printing string
    leaq    string_format(%rip), %rdi  # Load format string for string
    movq    16(%rbp), %rsi              # Load string pointer from stack
    xor     %rax, %rax                  # Float register count
    call    printf                      # Call printf

cleanup:
    # Restore the stack if it was misaligned
    movb    -192(%rbp), %al               # Check if we marked the stack as adjusted
    testb   $1, %al
    jz      stack_ok
    addq    $8, %rsp                    # Restore original stack alignment

stack_ok:
    popq    %r15
    popq    %r14
    popq    %r13
    popq    %r12
    popq    %rbx
    popq    %rbp
    ret
memalloc:
    pushq %rbp
    mov %rsp, %rbp
```

```
    pushq %rbx
    pushq %rdi
    pushq %rsi
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15

    testq $15, %rsp
    jz is_mem_aligned

    pushq $0                    # align to 16 bytes

    movq 16(%rbp), %rdi
    call malloc

    add $8, %rsp                # remove padding

    jmp mem_done

is_mem_aligned:

    movq 16(%rbp), %rdi
    call malloc

mem_done:

    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %rsi
    popq %rdi
    popq %rbx
    popq %rbp

    ret
```

Code 6. The corresponding x86 Assembly of Example 3

Output of Example 3

```
String are not equal.
The strings are equal.
```

Example 4

```
def main() -> None:
    x: int = 5
    y: int = 3

    print(x+y)
    print(x & y)
    print(x-(y*(x//y)))
    print(x ^ y)
    print(x | (x&y))

if __name__ == "__main__":
    main()
```

Code 7. Evaluation of expressions inside strings similarly function calls can also be done inside print.

```
.data
  integer_format: .asciz "%d\n"
  string_format: .asciz "%s\n"
  .str0:
    .string "__main__"
  .global main
  .text
main:
  pushq %rbp
  movq %rsp, %rbp
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  subq $128, %rsp
  movq $5, -40(%rbp)
  movq $3, -48(%rbp)
  movq -40(%rbp), %rdx
  add -48(%rbp), %rdx
  movq %rdx, -56(%rbp)
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq -56(%rbp)
  call global.print
  addq $8, %rsp
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq -40(%rbp), %rax
  movq -48(%rbp), %rcx
  and %rcx, %rax
  movq %rax, -72(%rbp)
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq -72(%rbp)
  call global.print
  addq $8, %rsp
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq -40(%rbp), %rax
  cqto
  movq -48(%rbp), %rcx
  idiv %rcx
  movq %rax, -80(%rbp)
  movq -48(%rbp), %rdx
  imul -80(%rbp), %rdx
  movq %rdx, -88(%rbp)
  movq -40(%rbp), %rdx
  sub -88(%rbp), %rdx
  movq %rdx, -96(%rbp)
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq -96(%rbp)
  call global.print
  addq $8, %rsp
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq -40(%rbp), %rax
  movq -48(%rbp), %rcx
  xor %rcx, %rax
  movq %rax, -104(%rbp)
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
```

```
    pushq %r11
    pushq -104(%rbp)
    call global.print
    addq $8, %rsp
    popq %r11
    popq %r10
    popq %r9
    popq %r8
    popq %rcx
    popq %rdx
    popq %rsi
    popq %rdi
    popq %rax
    movq -40(%rbp), %rax
    movq -48(%rbp), %rcx
    and %rcx, %rax
    movq %rax, -112(%rbp)
    movq -40(%rbp), %rax
    movq -112(%rbp), %rcx
    or %rcx, %rax
    movq %rax, -120(%rbp)
    pushq %rax
    pushq %rdi
    pushq %rsi
    pushq %rdx
    pushq %rcx
    pushq %r8
    pushq %r9
    pushq %r10
    pushq %r11
    pushq -120(%rbp)
    call global.print
    addq $8, %rsp
    popq %r11
    popq %r10
    popq %r9
    popq %r8
    popq %rcx
    popq %rdx
    popq %rsi
    popq %rdi
    popq %rax
    movq $60, %rax
    xorq %rdi, %rdi
    syscall
global.print:
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %rbx
    pushq   %r12
    pushq   %r13
    pushq   %r14
    pushq   %r15

    # Ensure stack alignment for printf
    movq    %rsp, %rax          # Store current rsp
    andq    $15, %rax           # Check lower 4 bits,
     rsp should be 16-byte aligned before call
    jz      align_ok            # If zero, alignment is
      okay
    subq    $8, %rsp            # Adjust stack to be
     16-byte aligned
    movq    $1, -192(%rbp)        # Mark that we
     adjusted the stack

align_ok:
    # Check type in %rdi (1=string, otherwise integer)
    cmpq    $0, %rdi
    je      print_string

    # Setup for printing integer

    leaq    integer_format(%rip), %rdi # Load format
     string for integer
    movq    16(%rbp), %rsi              # Load integer
     value from stack
    xor     %rax, %rax                  # Float register
      count
    call    printf                      # Call printf
    jmp     cleanup                     # Jump to
     cleanup

print_string:
    # Setup for printing string
    leaq    string_format(%rip), %rdi  # Load format
     string for string
    movq    16(%rbp), %rsi              # Load string
     pointer from stack
    xor     %rax, %rax                  # Float register
      count
    call    printf                      # Call printf

cleanup:
    # Restore the stack if it was misaligned
    movb    -192(%rbp), %al              # Check if we
     marked the stack as adjusted
    testb   $1, %al
    jz      stack_ok
    addq    $8, %rsp                    # Restore
     original stack alignment

stack_ok:
    popq    %r15
    popq    %r14
    popq    %r13
    popq    %r12
    popq    %rbx
    popq    %rbp
    ret
memalloc:
    pushq %rbp
    mov %rsp, %rbp
    pushq %rbx
    pushq %rdi
    pushq %rsi
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15

    testq $15, %rsp
    jz is_mem_aligned

    pushq $0                    # align to 16 bytes

    movq 16(%rbp), %rdi
    call malloc

    add $8, %rsp                # remove padding

    jmp mem_done

is_mem_aligned:

    movq 16(%rbp), %rdi
    call malloc

mem_done:

    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %rsi
    popq %rdi
    popq %rbx
    popq %rbp

    ret
```

Code 8. The corresponding x86 Assembly of Example 4

Output of Example 4

```
8
1
2
6
5
```

Example 5

```
def main() -> None:
    # no of identical pairs
    arr: list[int]=[1,2,3,1,2]
    x: int = 0
    y: int = 0
    sol:int =0
    while x < len(arr):
        for y in range(len(arr)-x-1):
            if arr[x]==arr[y+1+x]:
                sol=sol+1
        x=x+1
    print(sol)

```

```
if __name__ == "__main__":
    main()
```

Code 9. Strings and their comparison

```
.data
  integer_format: .asciz "%d\n"
  string_format: .asciz "%s\n"
  .str0:
    .string "__main__"
  .global main
  .text
main:
  pushq %rbp
  movq %rsp, %rbp
  pushq %rbx
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  subq $320, %rsp
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq $40
  call memalloc
  add $8, %rsp
  movq %rax, -48(%rbp)
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq -48(%rbp), %rax
  movq $1, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rax
  movq $2, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rax
  movq $3, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rax
  movq $1, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rax
  movq $2, %rdx
  movq %rdx, (%rax)
  movq -48(%rbp), %rdx
  add $8, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rdx
  sub $40, %rdx
  movq %rdx, -48(%rbp)
  movq -48(%rbp), %rdx
  movq %rdx, -64(%rbp)
  movq $0, -72(%rbp)
  movq $0, -80(%rbp)
  movq $0, -88(%rbp)
L2:
  movq $5, -96(%rbp)
  movq -72(%rbp), %rbx
  movq -96(%rbp), %rcx
  cmp %rcx, %rbx
  jl L9
  jmp L1
L9:
  movq $5, -136(%rbp)
  movq -136(%rbp), %rdx
  sub -72(%rbp), %rdx
  movq %rdx, -144(%rbp)
  movq -144(%rbp), %rdx
  sub $1, %rdx
  movq %rdx, -152(%rbp)
  movq $0, -160(%rbp)
L4:
  movq -160(%rbp), %rdx
  movq %rdx, -80(%rbp)
  movq -160(%rbp), %rbx
  movq -152(%rbp), %rcx
  cmp %rcx, %rbx
  jl L8
  jmp L3
L8:
  movq -72(%rbp), %rdx
  imul $8, %rdx
  movq %rdx, -192(%rbp)
  movq -64(%rbp), %rdx
  add -192(%rbp), %rdx
  movq %rdx, -192(%rbp)
  movq -192(%rbp), %rax
  movq (%rax), %rdx
  movq %rdx, -208(%rbp)
  movq -80(%rbp), %rdx
  add $1, %rdx
  movq %rdx, -216(%rbp)
  movq -216(%rbp), %rdx
  add -72(%rbp), %rdx
  movq %rdx, -224(%rbp)
  movq -224(%rbp), %rdx
  imul $8, %rdx
  movq %rdx, -232(%rbp)
  movq -64(%rbp), %rdx
  add -232(%rbp), %rdx
  movq %rdx, -232(%rbp)
  movq -232(%rbp), %rax
  movq (%rax), %rdx
  movq %rdx, -248(%rbp)
  movq -208(%rbp), %rbx
  movq -248(%rbp), %rcx
  cmp %rcx, %rbx
  je L7
  jmp L6
L7:
  movq -88(%rbp), %rdx
  add $1, %rdx
  movq %rdx, -280(%rbp)
  movq -280(%rbp), %rdx
  movq %rdx, -88(%rbp)
  jmp L6
L6:
L5:
  movq -160(%rbp), %rdx
  add $1, %rdx
  movq %rdx, -160(%rbp)
  jmp L4
L3:
  movq -72(%rbp), %rdx
  add $1, %rdx
  movq %rdx, -296(%rbp)
  movq -296(%rbp), %rdx
  movq %rdx, -72(%rbp)
  jmp L2
L1:
  pushq %rax
  pushq %rdi
  pushq %rsi
  pushq %rdx
  pushq %rcx
  pushq %r8
  pushq %r9
  pushq %r10
  pushq %r11
  pushq -88(%rbp)
  call global.print
  addq $8, %rsp
  popq %r11
  popq %r10
  popq %r9
  popq %r8
  popq %rcx
  popq %rdx
  popq %rsi
  popq %rdi
  popq %rax
  movq $60, %rax
  xorq %rdi, %rdi
```

```
  syscall
global.print:
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %rbx
    pushq   %r12
    pushq   %r13
    pushq   %r14
    pushq   %r15

    # Ensure stack alignment for printf
    movq    %rsp, %rax          # Store current rsp
    andq    $15, %rax           # Check lower 4 bits, rsp should be 16-byte aligned before call
    jz      align_ok            # If zero, alignment is okay
    subq    $8, %rsp            # Adjust stack to be 16-byte aligned
    movq    $1, -192(%rbp)        # Mark that we adjusted the stack

align_ok:
    # Check type in %rdi (1=string, otherwise integer)
    cmpq    $0, %rdi
    je      print_string

    # Setup for printing integer

    leaq    integer_format(%rip), %rdi # Load format string for integer
    movq    16(%rbp), %rsi              # Load integer value from stack
    xor     %rax, %rax                  # Float register count
    call    printf                      # Call printf
    jmp     cleanup                     # Jump to cleanup

print_string:
    # Setup for printing string
    leaq    string_format(%rip), %rdi  # Load format string for string
    movq    16(%rbp), %rsi              # Load string pointer from stack
    xor     %rax, %rax                  # Float register count
    call    printf                      # Call printf

cleanup:
    # Restore the stack if it was misaligned
    movb    -192(%rbp), %al             # Check if we marked the stack as adjusted
    testb   $1, %al
    jz      stack_ok
    addq    $8, %rsp                    # Restore original stack alignment

stack_ok:
    popq    %r15
    popq    %r14
    popq    %r13
    popq    %r12
    popq    %rbx
    popq    %rbp
    ret
memalloc:
    pushq %rbp
    mov %rsp, %rbp
    pushq %rbx
    pushq %rdi
    pushq %rsi
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15

    testq $15, %rsp
    jz is_mem_aligned

    pushq $0                # align to 16 bytes

    movq 16(%rbp), %rdi
    call malloc

    add $8, %rsp            # remove padding

    jmp mem_done

is_mem_aligned:

    movq 16(%rbp), %rdi
    call malloc

mem_done:

    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %rsi
    popq %rdi
    popq %rbx
    popq %rbp

    ret
```

Code 10. The usage of `len` inside `range` for usage in a loop. Along with some array handling

Output of Example 5

```
2
```

Effort Log

All the team members have contributed equally to the project.