CS-1319: PLDI - Monsoon 23	**Team Name:** julius-stabs-back
Assignment #5	
Instructor: PPD	**Name:** Gautam Ahuja, Nistha Singh

Note: All of the codes are compiled on `Windows Subsystem for Linux version: 1.2.5.0 using Ubuntu 22.04.2 LTS` over gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0, flex 2.6.4 and bison (GNU Bison) 3.8.2. We used the following sources for reference:

1. SUNSOFT - x86-64 Assembly Language Reference Manual

2. Stack Frame Layout on x86-64

3. Fronsto (GitHub)

4. RKJenamani (GitHub)

5. Input-Output System Calls in C

We have written all of our code in pure `C` and took help from above repositories to understand the direction of working.

The test cases are present in folder `A5_Tests`. The resulting `TAC` are in folder `A5_Tests/tac` and the resulting `asm` files are in folder `A5_Tests/x86`. On compilation, the resulting programs will be present in `A5_Tests/exe`.

Note: To get the `.out` and `.s` files for a `.nc` file do the following:

- do a `make`.

- `./compiler 1 <file>.nc` for generating only Quads.

- `./compiler 2 <file>.nc` for generating both Quads and `x86` assembly.

Note: To use the`IOLibrary`, do a `make build` and generate an assembly file as per above steps. Then generate the `.o` file from assembly and link it to `myl.c` by using the following commands:

- ... Assume the `.asm` file exists

- `as <file>.asm -o <file>.o`

- `gcc <file>.o -o <file> -L. -lmyl -no-pie`

Makefile

Our main `c` file looks as follows:

```makefile
.SUFFIXES:
GROUP = 3_A5
TEST_DIR = A5_Tests
EXTRA_DIR = $(TEST_DIR)/extra_cred

compiler: 3_A5.y 3_A5_translator.c 3_A5.l
	bison -d 3_A5.y
	flex -o lex.yy.c 3_A5.l
	gcc -c myl.c
	ar -rcs libmyl.a myl.o
	gcc 3_A5_translator.c 3_A5.tab.c lex.yy.c -lfl -Werror -o
	  compiler

build: compiler

test-nc: compiler
	./compiler 2 3_A5.nc
	mv 3_A5_quads5.out 3_A5_quads_nc.out && mv 3_A5_quads5.asm
	  3_A5_quads_nc.asm
	as 3_A5_quads_nc.asm -o 3_A5_quads_nc.o
	gcc 3_A5_quads_nc.o -o 3_A5_quads_nc -L. -lmyl -no-pie

clean:
	rm -f compiler lex.yy.c 3_A5.tab.c 3_A5_translator.tab.h
	  test.tac 3_A5.tab.o 3_A5.tab.h libmyl.a myl.o 3_A5
	  3_A5_quads_nc.o 3_A5_quads_nc.asm 3_A5_quads_nc.out
	  3_A5_quads_nc
test-all: compiler
	python3 test-cases.py

test: compiler
	rm -rf ${TEST_DIR}/x86 && rm -rf ${TEST_DIR}/tac && rm -rf
	  ${TEST_DIR}/exe
	mkdir ${TEST_DIR}/x86 && mkdir ${TEST_DIR}/tac && mkdir
	  ${TEST_DIR}/exe

	./compiler 2 ${TEST_DIR}/test2.nc
	mv 3_A5_quads2.out ${TEST_DIR}/tac && mv 3_A5_quads2.asm
	  ${TEST_DIR}/x86
	as ${TEST_DIR}/x86/3_A5_quads2.asm -o
	  ${TEST_DIR}/x86/3_A5_quads2.o
	gcc ${TEST_DIR}/x86/3_A5_quads2.o -o ${TEST_DIR}/exe/3_A5_quads2
	  -L. -lmyl -no-pie

	./compiler 2 ${TEST_DIR}/test3.nc
	mv 3_A5_quads3.out ${TEST_DIR}/tac && mv 3_A5_quads3.asm
	  ${TEST_DIR}/x86
```

```
37    as ${TEST_DIR}/x86/3_A5_quads3.asm -o
      ↪  ${TEST_DIR}/x86/3_A5_quads3.o
38    gcc ${TEST_DIR}/x86/3_A5_quads3.o -o ${TEST_DIR}/exe/3_A5_quads3
      ↪  -L. -lmyl -no-pie
39
40    ./compiler 2 ${TEST_DIR}/test4.nc
41    mv 3_A5_quads4.out ${TEST_DIR}/tac && mv 3_A5_quads4.asm
      ↪  ${TEST_DIR}/x86
42    as ${TEST_DIR}/x86/3_A5_quads4.asm -o
      ↪  ${TEST_DIR}/x86/3_A5_quads4.o
43    gcc ${TEST_DIR}/x86/3_A5_quads4.o -o ${TEST_DIR}/exe/3_A5_quads4
      ↪  -L. -lmyl -no-pie
44
45    ./compiler 2 ${TEST_DIR}/test5.nc
46    mv 3_A5_quads5.out ${TEST_DIR}/tac && mv 3_A5_quads5.asm
      ↪  ${TEST_DIR}/x86
47    as ${TEST_DIR}/x86/3_A5_quads5.asm -o
      ↪  ${TEST_DIR}/x86/3_A5_quads5.o
48    gcc ${TEST_DIR}/x86/3_A5_quads5.o -o ${TEST_DIR}/exe/3_A5_quads5
      ↪  -L. -lmyl -no-pie
49
50    ./compiler 2 ${TEST_DIR}/test7.nc
51    mv 3_A5_quads7.out ${TEST_DIR}/tac && mv 3_A5_quads7.asm
      ↪  ${TEST_DIR}/x86
52    as ${TEST_DIR}/x86/3_A5_quads7.asm -o
      ↪  ${TEST_DIR}/x86/3_A5_quads7.o
53    gcc ${TEST_DIR}/x86/3_A5_quads7.o -o ${TEST_DIR}/exe/3_A5_quads7
      ↪  -L. -lmyl -no-pie
54
55    ./compiler 2 ${TEST_DIR}/test1.nc
56    mv 3_A5_quads1.out ${TEST_DIR}/tac && mv 3_A5_quads1.asm
      ↪  ${TEST_DIR}/x86
57    as ${TEST_DIR}/x86/3_A5_quads1.asm -o
      ↪  ${TEST_DIR}/x86/3_A5_quads1.o
58
59    ./compiler 2 ${TEST_DIR}/test6.nc
60    mv 3_A5_quads6.out ${TEST_DIR}/tac && mv 3_A5_quads6.asm
      ↪  ${TEST_DIR}/x86
61    as ${TEST_DIR}/x86/3_A5_quads6.asm -o
      ↪  ${TEST_DIR}/x86/3_A5_quads6.o
62    gcc ${TEST_DIR}/x86/3_A5_quads6.o -o ${TEST_DIR}/exe/3_A5_quads6
      ↪  -L. -lmyl -no-pie
63
64    gcc ${TEST_DIR}/x86/3_A5_quads1.o -o ${TEST_DIR}/exe/3_A5_quads1
      ↪  -L. -lmyl -no-pie
65
66 test-extra: compiler
```

```
67    rm -rf ${EXTRA_DIR}/x86 && rm -rf ${EXTRA_DIR}/tac && rm -rf
      ↪    ${EXTRA_DIR}/exe
68    mkdir ${EXTRA_DIR}/x86 && mkdir ${EXTRA_DIR}/tac && mkdir
      ↪    ${EXTRA_DIR}/exe
69
70    ./compiler 2 ${EXTRA_DIR}/ec_1.nc
71    mv 3_A5_quads1.out ${EXTRA_DIR}/tac && mv 3_A5_quads1.asm
      ↪    ${EXTRA_DIR}/x86
72    as ${EXTRA_DIR}/x86/3_A5_quads1.asm -o
      ↪    ${EXTRA_DIR}/x86/3_A5_quads1.o
73    gcc ${EXTRA_DIR}/x86/3_A5_quads1.o -o
      ↪    ${EXTRA_DIR}/exe/3_A5_quads1 -L. -lmyl -no-pie
74
75    ./compiler 2 ${EXTRA_DIR}/ec_3.nc
76    mv 3_A5_quads3.out ${EXTRA_DIR}/tac && mv 3_A5_quads3.asm
      ↪    ${EXTRA_DIR}/x86
77    as ${EXTRA_DIR}/x86/3_A5_quads3.asm -o
      ↪    ${EXTRA_DIR}/x86/3_A5_quads3.o
78    gcc ${EXTRA_DIR}/x86/3_A5_quads3.o -o
      ↪    ${EXTRA_DIR}/exe/3_A5_quads3 -L. -lmyl -no-pie
79
80    ./compiler 2 ${EXTRA_DIR}/ec_4.nc
81    mv 3_A5_quads4.out ${EXTRA_DIR}/tac && mv 3_A5_quads4.asm
      ↪    ${EXTRA_DIR}/x86
82    as ${EXTRA_DIR}/x86/3_A5_quads4.asm -o
      ↪    ${EXTRA_DIR}/x86/3_A5_quads4.o
83    gcc ${EXTRA_DIR}/x86/3_A5_quads4.o -o
      ↪    ${EXTRA_DIR}/exe/3_A5_quads4 -L. -lmyl -no-pie
84
85    ./compiler 2 ${EXTRA_DIR}/ec_5.nc
86    mv 3_A5_quads5.out ${EXTRA_DIR}/tac && mv 3_A5_quads5.asm
      ↪    ${EXTRA_DIR}/x86
87    as ${EXTRA_DIR}/x86/3_A5_quads5.asm -o
      ↪    ${EXTRA_DIR}/x86/3_A5_quads5.o
88    gcc ${EXTRA_DIR}/x86/3_A5_quads5.o -o
      ↪    ${EXTRA_DIR}/exe/3_A5_quads5 -L. -lmyl -no-pie
89
90    ./compiler 2 ${EXTRA_DIR}/ec_6.nc
91    mv 3_A5_quads6.out ${EXTRA_DIR}/tac && mv 3_A5_quads6.asm
      ↪    ${EXTRA_DIR}/x86
92    as ${EXTRA_DIR}/x86/3_A5_quads6.asm -o
      ↪    ${EXTRA_DIR}/x86/3_A5_quads6.o
93    gcc ${EXTRA_DIR}/x86/3_A5_quads6.o -o
      ↪    ${EXTRA_DIR}/exe/3_A5_quads6 -L. -lmyl -no-pie
94
95    ./compiler 2 ${EXTRA_DIR}/ec_7.nc
96    mv 3_A5_quads7.out ${EXTRA_DIR}/tac && mv 3_A5_quads7.asm
      ↪    ${EXTRA_DIR}/x86
```

```
 97          as ${EXTRA_DIR}/x86/3_A5_quads7.asm -o
      ↪  ${EXTRA_DIR}/x86/3_A5_quads7.o
 98          gcc ${EXTRA_DIR}/x86/3_A5_quads7.o -o
      ↪  ${EXTRA_DIR}/exe/3_A5_quads7 -L. -lmyl -no-pie
 99
100          ./compiler 2 ${EXTRA_DIR}/ec_8.nc
101          mv 3_A5_quads8.out ${EXTRA_DIR}/tac && mv 3_A5_quads8.asm
      ↪  ${EXTRA_DIR}/x86
102          as ${EXTRA_DIR}/x86/3_A5_quads8.asm -o
      ↪  ${EXTRA_DIR}/x86/3_A5_quads8.o
103          gcc ${EXTRA_DIR}/x86/3_A5_quads8.o -o
      ↪  ${EXTRA_DIR}/exe/3_A5_quads8 -L. -lmyl -no-pie
```

The `Makefile` here is compiling all the files we wrote for making our compiler, Flex, Bison, and C. The Bison file (3_A5.y), Flex file (3_A5.l), and C file (3_A5_translator.c) are compiled into a single executable named `compiler`.

The `build` target depends on the compiler target, meaning it will build the compiler if it hasn't been built already. There are testing targets (test-nc, test-all, test) that run the compiler on various test files (like 3_A5.nc, TEST_DIR/test1.nc, etc.). The output of these tests are moved to specific directories. The `clean` target removes all the generated files from the previous build, allowing for a clean build the next time `make` is run. The assembly code generated by the compiler is assembled into object files using the `as` command. These object files are then linked with a `IO-Library-Link` (libmyl.a) to create executables with `IO` support.

The `IOLibrary` is copmpiled into an object file which on which the `ar` is used to create a static library (`libmyl.a`) from the object file (`myl.o`). The flag `-rcs` is creating the archive `libmyl.a` with the object file `myl.o` and adding an index to the archive for faster linking.

The gcc command is being used to link the object file $TEST_DIR/x86/3_A5_quads1.o with the myl library to create an executable named $TEST_DIR/exe/3_A5_quads1. The `-L.` is telling the linker to look in the current directory for the `myl` library. Then, the `-no-pie` is being used to create a regular, non-position independent executable because when working with assembly code and generating binaries, we need to run at a specific address.

Symbol Table – Update Offsets

From the last assignment, we successfully generated Three Address Codes for `nanoC` programs. Here we write the program to convert the three address code to x86 `Assembly`.

For that we first started by updating offsets in out symbol table.

```
1  // Set Offsets after table is generated
2  void set_offset(symboltable* currST){
3      int offset = 0;
4      for(int i=0; i< currST->count; i++){
5          symboltableentry* entry = (currST->table_entries[i]);
```

```
6        if(entry->category == TYPE_FUNCTION){
7            set_offset(entry->next);
8        }
9        entry->offset = offset;
10       offset += entry->size;
11    }
12    return;
13 }
```

In the above code, The function set_offset(symboltable* currST) is used to set the offsets after the symbol table is generated. It takes the pointer to the current symbol table as an argument and in the function, we initialize an offset variable to 0. Then, we loop over each entry in the symbol table. For each entry, we are checking if its category is a function. If it is, then we are recursively call set_offset on the next entry in the symbol table.

After that, we set the offset of the current entry to the value of the offset variable, and then increment the offset by the size of the current entry. We continue doing this until all entries of symbol table are processes.
The process is helping us in the process of generating the activation record.

Activation Record

The Hash Map stores the name of the entry in the symbol table and its associated offset in the activation record. We are now implementing a Hash Table here for the Activation Record Structure for each function call.

```
1  #define MAX_HASH_AR 500
2
3  struct HashAR{
4      char* key;
5      int value;
6      struct HashAR* next;
7  };
8  typedef struct HashAR HashAR;
9
10 unsigned int hash_ar(char *key);
11 void insert_ar(char *key, int value, HashAR* hashmap[]);
12 int search_ar(char *key, HashAR *hashmap[]);
```

In the above code, we are using the HashAR structure to create a linked list for handling collisions in the hash table.

The hash_ar(char *key) function is used to generate a hash value for a given key. We are explaining the function code later. Then, the insert_ar(char *key, int value, HashAR* hashmap[]) function is used to insert a key-value pair into the hash table. It takes the key, the value, and the hash table as arguments. After that, the search_ar(char *key, HashAR *hashmap[])function is used to search for a key in the hash table and

return its associated value.

We then added an instance of this activation record in each of the symbol table structure as follows:

```c
struct symboltable{
    char* name;                         // Name of symbol table
    ...
    HashAR* _aRecord[MAX_HASH_AR];      // Activation Record Hashmap
};
```

The associated functions which are used to implement the hash table above are as follows:

```c
// HASH for Activation Record
unsigned int hash_ar(char *key){
    unsigned int hashValue = 0;
    while(*key !='\0'){
        hashValue += *key++;
    }
    return hashValue % MAX_HASH_AR;
}

void insert_ar(char* key, int value, HashAR* hashmap[]){
    if(key == NULL){
        // printf("Error insert_ar: Key cannot be NULL.\n");
        return;
    }
    int hashIndex = hash_ar(key);
    HashAR* temp = hashmap[hashIndex];
    while(temp != NULL){
        if(strcmp(temp->key, key) == 0){
            temp->value = value;
            return;
        }
        temp = temp->next;
    }
    HashAR* newNode = (HashAR*)malloc(sizeof(HashAR));
    newNode->key = key;
    newNode->value = value;
    newNode->next = hashmap[hashIndex];
    hashmap[hashIndex] = newNode;
}

// search_ar() -- return the offset
int search_ar(char *key, HashAR *hashmap[]){
    if(key == NULL){
        // printf("Error search_ar: Key cannot be NULL.\n");
        return 0;
    }
```

```
37    int hashIndex = hash_ar(key);
38    HashAR* temp = hashmap[hashIndex];
39    while(temp != NULL){
40        if(strcmp(temp->key, key) == 0){
41            return temp->value;
42        }
43        temp = temp->next;
44    }
45    return 0;
46 }
```

The function `hash_ar(char *key)` is a hash function generating a hash value for a given key upon call. It initializes a variable hashValue to 0 and then iterates over each character in the key, adding its `ASCII` value to hashValue. It then returns the remainder of hashValue divided by `MAX_HASH_AR`, ensuring that the hash value falls within the range of the hash table.

Next, the `insert_ar(char* key, int value, HashAR* hashmap[])` function is used to insert a key-value pair into the hash table. It first checks if the key is NULL, and if it is then it is simply exiting and returning. Then, it calculates the hash index for the key and checks if a node with the same key already exists in the hash table. If it does, it updates the value of that node. If it doesn't, it creates a new node, sets its key and value, and inserts it at the beginning of the linked list at the calculated hash index.

Then we are using the `search_ar(char *key, HashAR *hashmap[])` function to search for a key in the hash table and return its associated value. It first checks if the key is NULL, and if it is then it returns -1. Then, it calculates the hash index for the key and iterates over the linked list at that index, looking for a node with the same key. If it finds such a node, it returns its value. If it doesn't, it returns 0.

Then we created the function to generate activation record from a given symbol table. The function `gen_activation_record(symboltable* currST)` is doing the same in the below code (explained in detail below. It is recursive, which allows it to generate an activation record for all symbol tables.
The activation Record on an `x86-64` looks as follows:
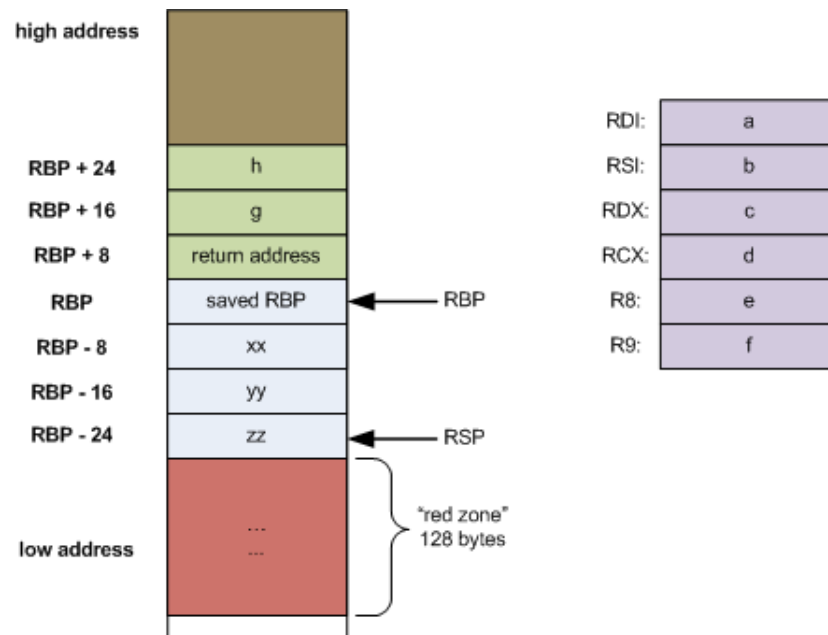
```
1   void gen_activation_record(symboltable* currST){
2       int local = -20;
3       int param = -24;
4
5       // iterate over the symbol table entries
6       for(int i=0; i < currST->count; i++){
7           symboltableentry* entry = currST->table_entries[i];
8           // printf("Entry: %s\n", entry->name);
9           if(entry->category == TYPE_PARAM){
10              // printf("Param: %s\n", entry->name);
11              param += entry->size;
12              insert_ar(entry->name, param, currST->_aRecord);
13          }
14          else if(entry->category == TYPE_RETURN){
15              // printf("Local: %s\n", entry->name);
16              continue;
17          }
18          else if(entry->category == TYPE_FUNCTION){
19              // printf("Function: %s\n", entry->name);
20              gen_activation_record(entry->next);
21          }
22          else{
23              // printf("Local: %s\n", entry->name);
24              local -= entry->size;
25              insert_ar(entry->name, local, currST->_aRecord);
26          }
27      }
28      return;
29  }
```

In the above code, The function gen_activation_record(symboltable* currST) be-

gins by setting two variables, local and param, to -20 and -24 respectively. In `x86-64` assembly language, the offsets -20 and -24 are used to reference memory locations relative to a base address, often the base of the stack. These offsets are negative because they are used to access memory locations that are below the current stack pointer. `rdi, rsi, rdx, rcx`.

The variables local and `param`, we will use them to track the offsets of local variables and parameters in the activation record. The function then iteraates over each entry in the symbol table. For each entry, it checks the category of the entry. If the entry is a parameter, it increments `param` variable by the size of the entry and inserts a new key-value pair into the activation record, with the key being the name of the entry and the value being `param`.

If the entry is a return type, then we continue to the next iteration and If the entry is a function then it recursively calls gen_activation_record on the next symbol table.

For all other entries, which are assumed to be local variables, it decrements varibale, local, by the size of the entry and inserts a new key-value pair into the activation record, with the key being the name of the entry and the value being local.

We do the above steps until all entries in the symbol table are processed. Once its done, then `fxn` returns, having generated a complete activation record for the given symbol table.

An activation record will look something like this:

```
===================================
ACTIVATION RECORD: main
-----------------------------------
Name            Category        Offset
-----------------------------------
a               local           -24
b               local           -32
c               local           -40
t1              temp            -28
t2              temp            -36
t3              temp            -44
===================================
```

Auxiliary Data Structures

We also created a few auxiliary data structures (particularly hash tables and linked lists) to help us out during the code translation process. These were:

Parameter Linked List

```
struct param_list{
    char* param; // The parameter stored in this node
    struct param_list* next; // Pointer to the next node in the list
};
typedef struct param_list param_list;
// Param List Functions
```

```
7   param_list* param_list_initialize(); // Function to initialize a new
    ↪   parameter list
8   param_list* param_list_delete(param_list* head); // Function to delete
    ↪   the parameter list
9   void param_list_insert(param_list* head, char* str); // Function to
    ↪   insert a new parameter into the parameter list
10  param_list* param_list_delete(param_list* head); // Function to delete
    ↪   the parameter list
```

The above code defines a linked list (param_list) and several functions for manipulating the LL. The param_list_initialize function is used to create a new list, the param_list_insert function is used to add a new item to an existing list, and the param_list_delete function is used to free the memory associated with a list when it is no longer needed. The exact implementation of these functions is shown and explain in detail later. The param field in the param_list structure is used to store the parameter associated with a particular node in the list, and the next field is used to link to the next node in the list. This LL is used to manage the parameters necessary for the function call. The idea is to have all parameters ready in order before calling a function.

The definitions of the above called functions are:

```
1   param_list* param_list_initialize(){
2       param_list* head = (param_list*)malloc(sizeof(param_list));
3       head->param = NULL;
4       head->next = NULL;
5       return head;
6   }
7
8   // insert a new string at the end of the linked list
9   void param_list_insert(param_list* head, char* str){
10      // check if it is the first entry
11      if(head->param == NULL){
12          head->param = strdup(str);
13          return;
14      }
15      // insert new entry at the end
16      param_list* temp = (param_list*)malloc(sizeof(param_list));
17      temp->param = strdup(str);
18      temp->next = NULL;
19      param_list* curr = head;
20      while(curr->next != NULL){
21          curr = curr->next;
22      }
23      curr->next = temp;
24  }
25
26  // delete the linked list end entry
```

```
27  param_list* param_list_delete(param_list* head){
28      param_list* curr = head;
29      while(curr != NULL){
30          param_list* temp = curr;
31          curr = curr->next;
32          free(temp);
33      }
34      head = param_list_initialize();
35      return head;
36  }
```

1. The function `param_list_initialize()` is creating a new linked list. It allocates memory for a new `param_list` structure, sets its `param` and `next` fields to NULL, and returns a pointer to the new structure which will serve as the head of the linked list.

2. The function `param_list_insert(param_list* head, char* str)` inserts a new parameter into the linked list. If the list is empty (i.e., `head->param` is NULL), it sets `head->param` to the new parameter. Otherwise, it creates a new `param_list` structure, sets its `param` field to the new parameter, and inserts it at the end of the list.

3. The function `param_list_delete(param_list* head)` deletes the linked list. It iterates over each `param_list` structure in the list, freeing the memory allocated to it. It then reinitializes the head of the list and returns it.

Global Variable Hash Map

```
1   globalVars* _globalVars[MAX_HASH_GLOBAL];
2   struct globalVars{
3       char* key; // The key stored in this node (name of the global
        ↪  variable)
4       bool value; // scope of the global variable
5       struct globalVars* next; / Pointer to the next node in the list
6   };
7   typedef struct globalVars globalVars;
8
9   unsigned int hash_global(char *key); //Generating a hash value for a
    ↪  given key
10  void insert_global(char *key, bool value, globalVars *hashmap[]);
    ↪  //inserting a new key-value pair into the hash table
11  bool search_global(char *key, globalVars *hashmap[]); //searching for a
    ↪  key in the hash table and return its value
```

This will be helpful when translating TAC to `x86` as we will be needing some information regarding the scope of global variable (and if any local variable arises with same name). This will help us manage variables of them in a hash table with name as key and a `bool` value regarding the global scope.

The definitions of these functions are as follows:

Global Variable Hash Map

```c
// HASH for Global Variables
unsigned int hash_global(char *key){
    // used the djb2 hash function
    unsigned int hashValue = 5381;
    int c;
    while ((c = *key++)) {
        hashValue = ((hashValue << 5) + hashValue) + c; // hashValue *
        ↪ 33 + c
    }
    return hashValue % MAX_HASH_GLOBAL;
}

void insert_global(char* key, bool value, globalVars* hashmap[]){
    if(key == NULL){
        // printf("Error insert_global: Key cannot be NULL.\n");
        return;
    }
    int hashIndex = hash_global(key);
    globalVars* temp = hashmap[hashIndex];
    while(temp != NULL){
        if(strcmp(temp->key, key) == 0){
            temp->value = value;
            return;
        }
        temp = temp->next;
    }
    globalVars* newNode = (globalVars*)malloc(sizeof(globalVars));
    newNode->key = key;
    newNode->value = value;
    newNode->next = hashmap[hashIndex];
    hashmap[hashIndex] = newNode;
}

bool search_global(char *key, globalVars *hashmap[]){
    if(key == NULL){
        // printf("Error search_global: Key cannot be NULL.\n");
        return false;
    }
    int hashIndex = hash_global(key);
    // printf("\nSearching for %s on %d\n", key, hashIndex);
    globalVars* temp = hashmap[hashIndex];
    while(temp != NULL){
        if(strcmp(temp->key, key) == 0){
            return true;
        }
        temp = temp->next;
```

```
46          }
47          return false;
48      }
49
```

The `hash_global` function uses the djb2 hashing algorithm to generate a unique hash value for each key, which is then used to determine the index at which the key-value pair should be stored in the hash table. The `insert_global` function inserts a new key-value pair into the hash table. If a collision occurs (i.e., two keys have the same hash value), the function handles it by creating a linked list at the corresponding hash table index. The `search_global` function searches for a key in the hash table and returns its associated value. If the key is not found, then it returns false.

Labels Hash Map

The following piece of code outlines implementation of a hash table for managing labels in a program. Here, The `HashLabel` structure, which includes a key, a value, and a pointer to the next `HashLabel` structure, is the base of this hash table. The `hash_label` function generates a unique hash value for each key, while the `insert_label` function inserts a $key - value$ pair into the hash table. The `label_count` function counts the occurrences of a key in the hash table, and the `label_at` function retrieves a label at a specific key.

```
1   HashLabel* _lablesRecord[MAX_HASH_LABEL];
2   struct HashLabel{
3       int key;
4       int value;
5       struct HashLabel* next;
6   };
7   unsigned int hash_label(int key);
8   void insert_label(int key, int value, HashLabel *hashmap[]);
9   bool label_count(int key, HashLabel *hashmap[]);
10  HashLabel* label_at(int key, HashLabel *hashmap[]);
```

The function definitions are as follows:

```
1   // HASH for Label (return label)
2   unsigned int hash_label(int key){
3       return key % MAX_HASH_LABEL;
4   }
5   void insert_label(int key, int value, HashLabel *hashmap[]){
6       if(key < 0){
7           // printf("Error: Key cannot be negative.\n");
8           return;
9       }
10      int hashIndex = hash_label(key);
11      HashLabel* temp = hashmap[hashIndex];
12      while(temp != NULL){
```

```
13      if(temp->key == key){
14          temp->value = value;
15          return;
16      }
17      temp = temp->next;
18  }
19  HashLabel* newNode = (HashLabel*)malloc(sizeof(HashLabel));
20  newNode->key = key;
21  newNode->value = value;
22  newNode->next = hashmap[hashIndex];
23  hashmap[hashIndex] = newNode;
24 }
25 bool label_count(int key, HashLabel *hashmap[]){
26     if(key < 0){
27         // printf("Error: Key cannot be negative.\n");
28         return false;
29     }
30     int hashIndex = hash_label(key);
31     HashLabel* temp = hashmap[hashIndex];
32     while(temp != NULL){
33         if(temp->key == key){
34             return true;
35         }
36         temp = temp->next;
37     }
38     return false;
39 }
40 // lable_at() -- return the reference to the label
41 HashLabel* label_at(int key, HashLabel *hashmap[]){
42     if(key < 0){
43         // printf("Error: Key cannot be negative.\n");
44         return NULL;
45     }
46     int hashIndex = hash_label(key);
47     HashLabel* temp = hashmap[hashIndex];
48     while(temp != NULL){
49         if(temp->key == key){
50             return temp;
51         }
52         temp = temp->next;
53     }
54     return NULL;
55 }
```

The `hash_label` function generates a hash value for a given key, which is then used to determine the index at which the key-value pair should be stored in the hash table. The `insert_label` function inserts a new key-value pair into the hash table. If a collision occurs, the function handles it by creating a linked list at the corresponding hash table

index.

The label_count function checks if a label exists in the hash table. It returns true if the label is found and false otherwise. The label_at function retrieves a label at a specific key in the hash table. If the label is not found, the function returns NULL.

TAC to x86-64

The TAC is converted to x86-64 assembly by the tac2x86() function.

```c
void tac2x86(){
    // first loop
    qArray* currentQArray = quadArray;
    while(currentQArray !=NULL && currentQArray->arr != NULL &&
        currentQArray->count != 0){
        quad* currQ = currentQArray->arr;
        if (currQ->op == OP_GOTO || currQ->op == OP_LT || currQ->op ==
            OP_GT || currQ->op == OP_LT_EQUALS || currQ->op ==
            OP_GT_EQUALS || currQ->op == OP_EQUALS || currQ->op ==
            OP_NOT_EQUALS) {
            if (currQ->result == NULL) {
                currentQArray = currentQArray->nextQuad;
                continue;
            };
            int instr_no = atoi(currQ->result);
            insert_label(instr_no, 1, _lablesRecord);
        }
        currentQArray = currentQArray->nextQuad;
    }
    // second loop -- update _lablesRecord values and count
    // AMBIGIOUS
    for(int i=0; i < MAX_HASH_LABEL; i++){
        HashLabel* temp = _lablesRecord[i];
        while(temp != NULL){
            /*
            int instr_no = temp->key;
            int count = temp->value;
            int new_instr_no = instr_no + count;
            temp->value = new_instr_no;
            temp = temp->next;
            */
            // _LabelCount++;
            temp->value = ++_LabelCount;
            temp = temp->next;
        }
    }
    // begin the .s file here
    for(int i=0; i< globalST->count; i++){
        symboltableentry* entry = (globalST->table_entries[i]);
```

```c
36          if(entry->category != TYPE_FUNCTION){
37              // It is a Global Variable
38              // CHAR
39              if(entry->type->type == TYPE_CHAR){
40                  if(entry->initial_value == NULL){
41                      // printf("Global Char: %s\n", entry->name);
42                      printf("\t.comm\t%s,1,1\n", entry->name);
43                  }
44                  else{
45                      // printf("Global Char: %s = %s\n", entry->name,
                        ↪ entry->initial_value);
46                      printf("\t.globl\t%s\n", entry->name);
47                      // printf("\t.data\n");
48                      printf("\t.type\t%s, @object\n", entry->name);
49                      printf("\t.size\t%s, 1\n", entry->name);
50                      printf("%s:\n", entry->name);
51                      printf("\t.byte\t%d\n", atoi(entry->initial_value));
52                  }
53                  // insert into global hashmap
54                  insert_global(entry->name, true, _globalVars);
55              }
56              // INT
57              if(entry->type->type == TYPE_INT){
58                  if(entry->initial_value == NULL){
59                      // printf("Global Int: %s\n", entry->name);
60                      printf("\t.comm\t%s,4,4\n", entry->name);
61                  }
62                  else{
63                      // printf("Global Int: %s = %s\n", entry->name,
                        ↪ entry->initial_value);
64                      printf("\t.globl\t%s\n", entry->name);
65                      printf("\t.data\n");
66                      printf("\t.align 4\n");
67                      printf("\t.type\t%s, @object\n", entry->name);
68                      printf("\t.size\t%s, 4\n", entry->name);
69                      printf("%s:\n", entry->name);
70                      // long -> int
71                      printf("\t.long\t%d\n", atoi(entry->initial_value));
72                  }
73                  // insert into global hashmap
74                  insert_global(entry->name, true, _globalVars);
75              }
76              // ARRAY
77              if(entry->type->type == TYPE_ARRAY){
78                  printf("\t.comm\t%s,%d,4\n", entry->name, entry->size);
79                  // insert into global hashmap
80                  insert_global(entry->name, true, _globalVars);
81              }
```

```
82              }
83          }
84
85          // STRINGS -- LL at string_head
86          string_list* currString = string_head;
87          // get size
88          int string_head_size = ll_length(string_head);
89          if(string_head_size>0){
90              printf("\t.section\t.rodata\n");
91              currString = string_head;
92              while(currString != NULL){
93                  printf(".LC%d:\n", currString->entries);
94                  printf("\t.string\t%s\n", currString->str);
95                  currString = currString->next;
96              }
97          }
98
99          // TEXT SECTION
100         printf("\t.text \n");
101         // initialize params list
102         param_list* params_head = param_list_initialize();
103         currST = globalST;
104         bool make_quad = false;
105         // iterate over the quadArray
106         currentQArray = quadArray;
107
108         while(currentQArray != NULL && currentQArray->arr != NULL &&
           ↪  currentQArray->count != 0){
109             int iterator = currentQArray->count; // -1 ?
110             if(label_count(iterator, _lablesRecord)){
111                 int count = label_at(iterator, _lablesRecord)->value;
112                 printf(".L%d:\n", 2 * _LabelCount + count + 2);
113             }
114
115             char* op = printOP(currentQArray->arr->op);
116             char* arg1 = (currentQArray->arr->arg1 ==
               ↪  NULL)?(NULL):(strdup(currentQArray->arr->arg1));
117             char* arg2 = (currentQArray->arr->arg2 ==
               ↪  NULL)?(NULL):(strdup(currentQArray->arr->arg2));
118             char* result = (currentQArray->arr->result ==
               ↪  NULL)?(NULL):(strdup(currentQArray->arr->result));
119             char* s = arg2;
120
121             // Activation Record of Result
122             char* result_ar;
123             if(search_global(result, _globalVars)){
124                 // concatenate "(%rip)" to result
125                 result_ar = (char*)malloc(sizeof(char)*(strlen(result)+7));
```

```
126        sprintf(result_ar, "%s(%%rip)", result);
127    }
128    else{
129        //convert result of search_ar(result, currST->_aRecord) to
               ↪  string and concatenate "(%rbp)" to it
130        int offset = search_ar(result, currST->_aRecord);
131        result_ar = (char*)malloc(sizeof(char)*15);
132        sprintf(result_ar, "%d(%%rbp)", offset);
133    }

135    // Activation Record of Arg1
136    char* arg1_ar;
137    if(search_global(arg1, _globalVars)){
138        // concatenate "(%rip)" to arg1
139        arg1_ar = (char*)malloc(sizeof(char)*(strlen(arg1)+7));
140        sprintf(arg1_ar, "%s(%%rip)", arg1);
141    }
142    else{
143        //convert result of search_ar(arg1, currST->_aRecord) to
               ↪  string and concatenate "(%rbp)" to it
144        int offset = search_ar(arg1, currST->_aRecord);
145        arg1_ar = (char*)malloc(sizeof(char)*15);
146        sprintf(arg1_ar, "%d(%%rbp)", offset);
147    }

149    // Activation Record of Arg2
150    char* arg2_ar;
151    if(search_global(arg2, _globalVars)){
152        // concatenate "(%rip)" to arg2
153        arg2_ar = (char*)malloc(sizeof(char)*(strlen(arg2)+7));
154        sprintf(arg2_ar, "%s(%%rip)", arg2);
155    }
156    else{
157        //convert result of search_ar(arg2, currST->_aRecord) to
               ↪  string and concatenate "(%rbp)" to it
158        int offset = search_ar(arg2, currST->_aRecord);
159        arg2_ar = (char*)malloc(sizeof(char)*15);
160        sprintf(arg2_ar, "%d(%%rbp)", offset);
161    }
162    // parameter type
163    if(op == "param"){
164        // push arg1 to params_head
165        param_list_insert(params_head, result);
166    }
167    else{
168        printf("\t");
169        // Binary Operations
170        // Addition
```

```
171             if (op == "+") {
172                 bool flag = true;
173                 if(s==NULL || ((!isdigit(s[0])) && (s[0] != '-') &&
    ↪  (s[0] != '+'))){
174                     flag = false;
175                 }
176                 else{
177                     char* p;
178                     strtol(s, &p, 10);
179                     if(*p == 0)
180                         flag = true;
181                     else
182                         flag = false;
183                 }
184                 if(flag){
185                     printf("addl \t$%d, %s\n", atoi(arg2), arg1_ar);
186                 }
187                 else{
188                     // AMBIGIOUS
189                     printf("\tmovl\t%s, %%eax\n", arg1_ar);
190                     printf("\tmovl\t%s, %%edx\n", arg2_ar);
191                     printf("\taddl \t%%edx, %%eax\n");
192                     printf("\tmovl\t%%eax, %s\n", result_ar);
193                 }
194             }
195             // Subtraction
196             else if (op == "-"){
197                 printf("\tmovl\t%s, %%eax\n", arg1_ar);
198                 printf("\tmovl\t%s, %%edx\n", arg2_ar);
199                 printf("\tsubl \t%%edx, %%eax\n");
200                 printf("\tmovl\t%%eax, %s\n", result_ar);
201             }
202             // multiplication
203             else if(op == "*"){
204                 printf("\tmovl\t%s, %%eax\n", arg1_ar);
205                 bool flag = true;
206                 if(s==NULL || ((!isdigit(s[0])) && (s[0] != '-') &&
    ↪  (s[0] != '+'))){
207                     flag = false;
208                 }
209                 else{
210                     char* p;
211                     strtol(s, &p, 10);
212                     if(*p == 0)
213                         flag = true;
214                     else
215                         flag = false;
216                 }
```

```c
217                    if(flag){
218                        printf("# %s = %s * %s\n", result, arg1, arg2);
219                        printf("\timull \t$%d, %%eax\n", atoi(arg2));
220                        symboltable* tempTab = currST;
221                        char* val;
222                        // check if arg1 is a global variable
223                        for(int i=0; i <tempTab->count; i++){
224                            if(strcmp((tempTab->table_entries[i])->name,
                                 arg1) == 0){
225                                val =
                                     strdup((tempTab->table_entries[i])->initial_value);
                                     // value found, propagate the name
226                            }
227                        }
228                    }
229                    else{
230                        printf("\timull \t%s, %%eax\n", arg2_ar);
231                        printf("\tmovl\t%%eax, %s\n", result_ar);
232                    }
233                }
234                // division
235                else if(op=="/"){
236                    printf("\tmovl\t%s, %%eax\n", arg1_ar);
237                    printf("\tcltd\n");
238                    printf("\tidivl \t%s\n", arg2_ar);
239                    printf("\tmovl\t%%eax, %s\n", result_ar);
240                }
241                // modulo
242                else if(op == "%"){
243                    printf("\tmovl\t%s, %%eax\n", arg1_ar);
244                    printf("\tcltd\n");
245                    printf("\tidivl \t%s\n", arg2_ar);
246                    printf("\tmovl\t%%edx, %s\n", result_ar);
247                }
248                // assign
249                else if(op =="="){
250                    if(make_quad == true){
251                        printf("\tmovq \t%s, %%rax\n", arg1_ar);
252                        printf("\tmovq \t%%rax, %s\n", result_ar);
253                        make_quad = false;
254                    }
255                    else{
256                        s = arg1;//(arg1 == NULL)?(NULL):(strdup(arg1));
257                        bool flag = true;
258                        if(s==NULL || ((!isdigit(s[0])) && (s[0] != '-') &&
                             (s[0] != '+'))){
259                            flag = false;
260                        }
```

```
261              else{
262                  char* p;
263                  strtol(s, &p, 10);
264                  if(*p == 0)
265                      flag = true;
266                  else
267                      flag = false;
268              }
269              if(flag){
270                  printf("movl\t$%d, %%eax\n", atoi(arg1));
271              }
272              else{
273                  printf("\tmovl\t%s, %%eax\n", arg1_ar);
274              }
275              printf("\tmovl\t%%eax, %s\n", result_ar);
276          }
277      }
278      else if(op=="=str"){
279          printf("movq \t$.LC%s, %s\n", arg1, result_ar);
280      }
281      // Relational
282      else if(op=="=="){
283          printf("\tmovl\t%s, %%eax\n", arg1_ar);
284          printf("\tcmpl\t%s, %%eax\n", arg2_ar);
285          int tempCount = label_at(atoi(result),
                ↪ _lablesRecord)->value;
286          printf("\tje .L%d\n", 2 * _LabelCount + tempCount + 2);
287      }
288      else if(op=="!="){
289          printf("\tmovl\t%s, %%eax\n", arg1_ar);
290          printf("\tcmpl\t%s, %%eax\n", arg2_ar);
291          int tempCount = label_at(atoi(result),
                ↪ _lablesRecord)->value;
292          printf("\tjne .L%d\n", 2 * _LabelCount + tempCount + 2);
293      }
294      else if(op=="<"){
295          printf("\tmovl\t%s, %%eax\n", arg1_ar);
296          printf("\tcmpl\t%s, %%eax\n", arg2_ar);
297          int tempCount = label_at(atoi(result),
                ↪ _lablesRecord)->value;
298          printf("\tjl .L%d\n", 2 * _LabelCount + tempCount + 2);
299      }
300      else if(op==">"){
301          printf("\tmovl\t%s, %%eax\n", arg1_ar);
302          printf("\tcmpl\t%s, %%eax\n", arg2_ar);
303          int tempCount = label_at(atoi(result),
                ↪ _lablesRecord)->value;
304          printf("\tjg .L%d\n", 2 * _LabelCount + tempCount + 2);
```

```
305              }
306              else if(op=="<="){
307                  printf("\tmovl\t%s, %%eax\n", arg1_ar);
308                  printf("\tcmpl\t%s, %%eax\n", arg2_ar);
309                  int tempCount = label_at(atoi(result),
                  ↪  _lablesRecord)->value;
310                  printf("\tjle .L%d\n", 2 * _LabelCount + tempCount + 2);
311              }
312              else if(op==">="){
313                  printf("\tmovl\t%s, %%eax\n", arg1_ar);
314                  printf("\tcmpl\t%s, %%eax\n", arg2_ar);
315                  int tempCount = label_at(atoi(result),
                  ↪  _lablesRecord)->value;
316                  printf("\tjge .L%d\n", 2 * _LabelCount + tempCount + 2);
317              }
318              else if(op=="goto"){
319                  if (result != NULL) {
320                      int tempCount = label_at(atoi(result),
                      ↪  _lablesRecord)->value;
321                      printf("jmp .L%d\n", 2 * _LabelCount + tempCount +
                      ↪  2);
322                  }
323              }
324
325              // Unary Operations
326              else if(op == "=&"){
327                  printf("# %s = &%s\n", result, arg1);
328                  printf("\tleaq\t%s, %%rax\n", arg1_ar);
329                  printf("\tmovq \t%%rax, %s\n", result_ar);
330                  make_quad = true;
331              }
332              else if(op=="=*"){
333                  printf("# %s = *%s\n", result, arg1);
334                  printf("\tmovq\t%s, %%rax\n", arg1_ar);
335                  printf("\tmovl\t(%%rax), %%eax\n");
336                  printf("\tmovl\t%%eax, %s\n", result_ar);
337              }
338              else if(op=="*="){
339                  printf("# *%s = %s\n", result, arg1);
340                  printf("\tmovl\t%s, %%eax\n", result_ar);
341                  printf("\tmovl\t%s, %%edx\n", arg1_ar);
342                  // cout << "\tmovl\t%edx, (%eax)";
343                  printf("\tmovl\t%%edx, (%%eax)\n");
344              }
345              else if(op=="uminus"){
346                  printf("\tmovl\t%s, %%eax\n", arg1_ar);
347                  printf("\tnegl\t%%eax\n");
348                  printf("\tmovl\t%%eax, %s\n", result_ar);
```

```
349                }
350             else if(op == "=[]"){
351                 printf("# =[] operation ; ");
352                 printf("%s = %s[%s]\n", result, arg1, arg2);
353                 if(search_global(arg1, _globalVars)){
354                     printf("\tmovl\t%s, %%eax\n", arg2_ar);
355                     printf("\tmovslq\t%%eax, %%rdx\n");
356                     printf("\tleaq\t0(,%%rdx,4), %%rdx\n");
357                     printf("\tleaq\t%s, %%rax\n", arg1_ar);
358                     printf("\tmovl\t(%%rdx,%%rax), %%eax\n");
359                     printf("\tmovl\t%%eax, %s\n", result_ar);
360                 }
361                 else{
362                     printf("\tmovl\t%s, %%ecx\n", arg2_ar);
363                     printf("\tmovl\t%d(%%rbp,%%rcx,4), %%eax\n",
                     ↪  search_ar(arg1, currST->_aRecord));
364                     printf("\tmovl\t%%eax, %s\n", result_ar);
365                 }
366             }
367             else if(op=="[]="){
368                 printf("# []= operation ; ");
369                 printf("%s[%s] = %s\n", result, arg1, arg2);
370                 if(search_global(result, _globalVars)){
371                     printf("\tmovl\t%s, %%eax\n", arg2_ar);
372                     printf("\tmovl\t%s, %%edx\n", arg1_ar);
373                     printf("\tmovslq\t%%edx, %%rdx\n");
374                     printf("\tleaq\t0(,%%rdx,4), %%rcx\n");
375                     printf("\tleaq\t%s, %%rdx\n", result_ar);
376                     printf("\tmovl\t%%eax, (%%rcx,%%rdx)\n");
377                 }
378                 else{
379                     printf("\tmovl\t%s, %%eax\n", arg1_ar);
380                     printf("\tmovl\t%s, %%edx\n", arg2_ar);
381                     printf("\tmovl\t%%edx, %d(%%rbp,%%rax,4)\n",
                     ↪  search_ar(result, currST->_aRecord));
382                 }
383             }
384             else if(op=="return"){
385                 if(result != NULL){
386                     printf("\tmovl\t%s, %%eax\n", result_ar);
387                 }
388                 // jump to the end of the function -- epilogue
389                 printf("\tjmp .LFE%d\n", _LabelCount);
390             }
391             else if(op=="param"){
392                 // push arg1 to params_head
393                 param_list_insert(params_head, result);
394             }
```

```
395
396              // function call
397              else if(op=="call"){
398                  // 4 registers are used for passing parameters -- rdi,
                     ↪  rsi, rdx, rcx
399                  param_list* tempPara = params_head;
400
401                  int paraCOUNT = 0;
402                  while(tempPara != NULL){
403                      paraCOUNT++;
404                      tempPara = tempPara->next;
405                  }
406                  tempPara = params_head;
407
408                  for(int i=0; i<paraCOUNT; i++){
409                      if(i==0){
410                          // first parameter
411                          int val = search_ar(tempPara->param,
                             ↪  currST->_aRecord);
412                          printf("movl\t%d(%%rbp), %%eax\n", val);
413                          printf("\tmovq\t%d(%%rbp), %%rdi\n", val);
414                      }
415                      else if(i==1){
416                          int val = search_ar(tempPara->param,
                             ↪  currST->_aRecord);
417                          printf("movl\t%d(%%rbp), %%eax\n", val);
418                          printf("\tmovq\t%d(%%rbp), %%rsi\n", val);
419                      }
420                      else if(i==2){
421                          int val = search_ar(tempPara->param,
                             ↪  currST->_aRecord);
422                          printf("movl\t%d(%%rbp), %%eax\n", val);
423                          printf("\tmovq\t%d(%%rbp), %%rdx\n", val);
424                      }
425                      else if(i==3){
426                          int val = search_ar(tempPara->param,
                             ↪  currST->_aRecord);
427                          printf("movl\t%d(%%rbp), %%eax\n", val);
428                          printf("\tmovq\t%d(%%rbp), %%rcx\n", val);
429                      }
430                      else{
431                          int val = search_ar(tempPara->param,
                             ↪  currST->_aRecord);
432                          printf("\tmovq\t%d(%%rbp), %%rdi\n", val);
433                      }
434                      tempPara = tempPara->next;
435                  }
436                  // clear para stack
```

```
437                    params_head = param_list_delete(params_head);
438                    printf("\tcall\t%s\n", arg1);
439                    printf("\tmovl\t%%eax, %s\n", result_ar);
440                }
441
442            else if(op == "function"){
443                // function begins -- prologue
444                printf(".globl\t%s\n", result);
445                printf("\t.type\t%s, @function\n", result);
446                printf("%s: \n", result);
447                printf(".LFB%d: \n", _LabelCount);
448                printf("\t.cfi_startproc\n");
449                printf("\tpushq\t%%rbp\n");
450                printf("\t.cfi_def_cfa_offset 8\n");
451                printf("\t.cfi_offset 5, -8\n");
452                printf("\tmovq\t%%rsp, %%rbp\n");
453                printf("\t.cfi_def_cfa_register 5\n");
454                currST = lookup(globalST, result)->next;
455                // get last entry of the symbol table -- count starts
                   ↪  from 0
456                symboltableentry* lastEntry =
                   ↪  currST->table_entries[currST->count-1];
457                // get the size of the symbol table
458                int sizeTemp = lastEntry->offset;
459                // rsp register holds the address of the top of the
                   ↪  stack
460                printf("\tsubq\t$%d, %%rsp\n", sizeTemp+24);  // MAX
                   ↪  BUFFER: 4 Para + retVal + RA = 24 = (4+1+1)*4
461
462                // function table -- paramaters section
463                for(int i=0; i < currST->paramCount; i++){
464                    symboltableentry* entry = currST->_argList[i];
465                    if(i==0){
466                        printf("\tmovq\t%%rdi, %d(%%rbp)\n",
                           ↪  search_ar(entry->name, currST->_aRecord));
467                    }
468                    else if(i==2){
469                        printf("\tmovq\t%%rsi, %d(%%rbp)\n",
                           ↪  search_ar(entry->name, currST->_aRecord));
470                    }
471                    else if(i==3){
472                        printf("\tmovq\t%%rdx, %d(%%rbp)\n",
                           ↪  search_ar(entry->name, currST->_aRecord));
473                    }
474                    else if(i==4){
475                        printf("\tmovq\t%%rcx, %d(%%rbp)\n",
                           ↪  search_ar(entry->name, currST->_aRecord));
476                    }
```

```
477                    }
478                }
479                // function end -- epilogue
480                else if(op=="end"){
481                    printf(".LFE%d: \n", _LabelCount++);
482                    printf("leave\n");
483                    printf("\t.cfi_restore 5\n");
484                    printf("\t.cfi_def_cfa 4, 4\n");
485                    printf("\tret\n");
486                    printf("\t.cfi_endproc\n");
487                    printf("\t.size\t%s, .-%s\n", result, result);
488                }
489                else{
490                    printf("op: %s\n", op);
491                }
492                // printf("\n");
493            }
494            currentQArray = currentQArray->nextQuad;
495        }
496        // footer
497        printf("\t.ident\t\"group-03-julius-stabs-back\"\n");
498        printf("\t.section\t.note.GNU-stack,\"\",@progbits\n");
499 }
```

The Function `tac2x86()` is responsible for converting Three Address Code (TAC) into x86-64 assembly language.

For label identification and update, The first loop in `tac2x86()` scans the TAC instructions and identifies any `jump` labels, which are used for control flow operations like `goto`, `if`, etc. These labels are stored in a hash map `_lablesRecord`. The second loop updates these labels with their corresponding instruction numbers. This is necessary because the TAC and `x86-64` assembly may not have a one-to-one correspondence between instructions.

We are handling many operations and translating them into corresponding x86-64 assembly instructions. Here's how we are doing it:

1. For Binary Operations,like addition (`'+'`), subtraction (`'-'`), multiplication (`'*'`), (`'/'`), and modulo (`'%'`), the operands are moved to the appropriate registers (`'%eax)'`, `'%edx)'`, etc.), the operation is performed, and the result is stored back in the appropriate location.

2. For Relational Operations like equals (`'=='`), not equals (`'!='`), less than (`'<'`), greater than (`'>'`), less than or equal to (`'<='`), and greater than or equal to (`'>='`), the operands are compared using the `'cmp'` instruction, and then a conditional jump instruction (`'je'`, `'jne'`, `'jl'`, `'jg'`, `'jle'`, `'jge'`) is used to jump to the appropriate label based on the result of the comparison.

3. For Unary Operations like address of (`'=&'`), dereference (`'=*'`), negation (`'uminus'`), and array indexing (`'=[]'`), the code generates the appropriate assembly instructions. For example, for the address of operation, it uses the `'leaq'` instruction to load the address of a variable into a register.

4. For Function Calls (`call`), the code is pushing the arguments onto the stack and then calls the function. For function definitions (`function`), it generates the function prologue (which sets up the stack frame) and epilogue (which cleans up the stack frame). The 'return' statement is also handled here.

5. The code handles global variables and strings. It allocates space for global variables in the '.data' section of the assembly code. It also stores strings in the '.rodata' section.

6. The code maintains activation records for each function. These records store information about the function's local variables and parameters. The code uses these records to generate the correct assembly instructions for accessing these variables and parameters.

`main()` function

Our main function is simple compiler that can generate either TAC or Assembly code from a given source code file. various stages of the compilation process, including parsing, intermediate code generation, and final code generation are done here. It also shows how to handle command-line arguments and perform file I/O operations in C. Our `main()` function looks like follows:

```c
int main(int argc, char** argv){
    printf("\n\n");
    // check if the input < 3 or if first para is non digit
    if (argc < 3 ) {
        printf("Usage: %s <1=out/2=asm> <nanoC file>\n", argv[0]);
        return 1;
    }
    const char* inname = argv[2];
    // remove the extension
    printf("Starting Compiler\n");
    int dot_at = strlen(inname)-3;
    char outname = inname[strlen(inname)-4]; // =
    ↪   (char*)malloc(sizeof(char)*dot_at+1);
    // for(int i=0; i!=dot_at; i++){
    //     outname[i] = inname[i];
    // }
    outname = inname[dot_at-1];
    printf("Input File: %s\n", inname);
    printf("Test Case: %c\n", outname);

    for(int i = 0; i < MAX_HASH_LABEL; i++){
        _lablesRecord[i] = NULL;
        _globalVars[i] = NULL;
    }

    printf("Initializing Symbol Tables\n");
    globalST = create_symboltable("Global", NULL);
```

```
27    currST = globalST;
28    printf("Initializing Variable STack\n");
29    stack_intialize(&var_type);
30    printf("Initializing String Linked List\n");
31    string_head = string_list_initialize();
32    printf("Initializing Quad Array\n");
33    quadArray = quadArray_initialize(quadArray);
34
35    printf("Starting Parser\n");
36    // send input to parser
37    FILE* file = fopen(inname, "r");
38    if (!file) {
39        perror("Error opening file");
40        return 1;
41    }
42
43    yyin = file;
44    yyparse();
45    printf("Parser Done, Updating Symbol Tables Offset\n");
46    set_offset(globalST);
47    printf("Generating Activation Records\n");
48    gen_activation_record(globalST);
49
50    if(strcmp(argv[1], "1") == 0){
51        printf("Writing TACs\n");
52
53        // open file for writing outname.out
54        char* temp_o = "3_A5_quads";
55        char* extension_out = (char*)malloc(sizeof(char)*6);
56        sprintf(extension_out, "%c.out", outname);
57        char* outname_out = (char*)malloc(sizeof(char)*25);
58        sprintf(outname_out, "%s%s", temp_o, extension_out);
59
60        FILE* file_out = fopen(outname_out, "w");
61        if (!file_out) {
62            perror("Error opening file");
63            return 1;
64        }
65        printf("\tWriting TACs -- Activation Record\n");
66        print_activationRecord(file_out, globalST);
67        printf("\tWriting TACs -- Symbol Table\n");
68        print_ST(file_out, globalST);
69        printf("\tWriting TACs -- TAC\n");
70        print_quadArray(file_out, quadArray);
71        printf("\n\n\n");
72        // close file
73        fclose(file_out);
74    }
```

```
75
76      if(strcmp(argv[1], "2") == 0){
77          printf("Writing TACs\n");
78
79          // open file for writing outname.out
80          char* temp_o = "3_A5_quads";
81          char* extension_out = (char*)malloc(sizeof(char)*6);
82          sprintf(extension_out, "%c.out", outname);
83          char* outname_out = (char*)malloc(sizeof(char)*25);
84          sprintf(outname_out, "%s%s", temp_o, extension_out);
85
86          FILE* file_out = fopen(outname_out, "w");
87          if (!file_out) {
88              perror("Error opening file");
89              return 1;
90          }
91          printf("\tWriting TACs -- Activation Record\n");
92          print_activationRecord(file_out, globalST);
93          printf("\tWriting TACs -- Symbol Table\n");
94          print_ST(file_out, globalST);
95          printf("\tWriting TACs -- TAC\n");
96          print_quadArray(file_out, quadArray);
97          printf("TAC DONE\n\n");
98          // close file
99          fclose(file_out);
100         printf("Generating ASM\n");
101
102         // open file for writing outname.asm
103         char* temp_s = "3_A5_quads";
104         char* extension_asm = (char*)malloc(sizeof(char)*4);
105         sprintf(extension_asm, "%c.s", outname);
106         char* outname_asm = (char*)malloc(sizeof(char)*25);
107         sprintf(outname_asm, "%s%s", temp_s, extension_asm);
108
109         FILE* file_asm = fopen(outname_asm, "w");
110         if (!file_asm) {
111             perror("Error opening file");
112             return 1;
113         }
114         tac2x86(file_asm);
115         // close file
116         fclose(file_asm);
117         printf("ASM DONE\n\n\n");
118     }
119     fclose(file);
120     return 0;
121 }
```

The program begins by checking the number of command-line arguments provided by the user. This is done in lines 4 to 7. If the number of arguments is less than 3, the program prints a usage message and exits. Upon accepting the command-line arguments, the program goes on to get the input filename, which is expected to be the second argument ('`argv[2]`'). This is done in lines 8 to 16. The program also removes the extension from the filename to prepare for the creation of the output filename. This is necessary because the output file will have a different extension based on whether it contains Three Address Code (TAC) or Assembly code.

Next, we initialize various data structures that are used to store information about the source code during the compilation process. This is done in lines 19 to 32. These data structures include symbol tables, a variable stack, a string linked list, and a quad array.

Once the data structures are initialized, we open the input file and sends it to the parser. This is done in lines 34 to 43. The parser is responsible for analyzing the source code and generating an intermediate representation. After the parsing is complete, our program updates the symbol tables with offset information and generates activation records.

The program then checks the first command-line argument ('`argv[1]`') to determine whether to generate TAC or Assembly code. If the user has chosen to generate TAC (by providing '1' as the argument), the program writes the activation record, symbol table, and TAC to an output file. This is done in lines 48 to 67. If the user has chosen to generate Assembly code (by providing '2' as the argument), the program does the same but also converts the TAC to x86 Assembly code. This is done in lines 69 to 100.

Finally, in lines 102 and 103, the program closes the input and output files and exits. This marks the end of the compilation process. The output file now contains either TAC or Assembly code, depending on the user's choice. This code is used for further processing or execution.

Other

Some other minor changes include changing functions `printf() -> fprintf()` in the functions `print_activationRecord()`, `print_ST()`, `print_quadArray()`, `tac2x86()` so that they write the outputs to appropriate files.

Bonus

Implementing IO Library

Reference: Input-output system calls in C.
We implemented a IO library which reads and prints an integer and a string.

```c
#include "myl.h"
#include <unistd.h>

/*
Function to print a string
We will use the write system call to print the string to STDOUT
Reference:
↪    https://www.geeksforgeeks.org/input-output-system-calls-c-create
-open-close-read-write/
*/
int printStr(char *str) {
    int len = 0;
    // Calculate the length of the string
    while (str[len] != '\0') {
        len++;
    }

    // Use the write system call to print the string to STDOUT
        // 1 is the file descriptor for STDOUT
    write(1, str, len);
    return len;
}


/*
Function to print an integer
We go through the following steps:
1. Allocate dynamic memory for a character array (string) to store the
↪    integer
2. Convert the integer to a character array (string) in reverse order
3. Reverse the string if it's a negative number
4. Use the write system call to print the integer string to STDOUT
*/
int printInt(int num) {
    char buffer[100];
    int index = 0;
        int bytes;

    if (buffer == NULL) {
        return ERR;
    }

```

```
41      // Convert the integer to a character array (string) in reverse
   ↪   order
42      if (num == 0) {
43          buffer[index++] = '0';
44      }
45          else {
46          if (num < 0) {
47              buffer[index++] = '-';
48              num = -num;
49          }
50
51          while (num) {
52              int digit = num % 10;
53              buffer[index++] = (char)(digit + '0');
54              num /= 10;
55          }
56
57          // Reverse the string if it's a negative number
58          if (buffer[0] == '-') {
59              int start = 1, end = index - 1;
60              while (start < end) {
61                  char temp = buffer[start];
62                  buffer[start++] = buffer[end];
63                  buffer[end--] = temp;
64              }
65          }
66              else {
67          // Reverse the string if it's a positive number
68          int start = 0, end = index - 1;
69          while (start < end) {
70              char temp = buffer[start];
71              buffer[start++] = buffer[end];
72              buffer[end--] = temp;
73          }
74      }
75  }
76      // Calculate the number of bytes in the string
77      bytes = index;
78
79      // Use the write system call to print the integer string to STDOUT
80      write(1, buffer, bytes);
81
82      return bytes;
83  }
84
85  /*
86  A function to read an integer from STDIN
87  We go through the following steps:
```

```c
   1. Allocate dynamic memory for a character array (string) to store the
   →   integer
   2. Read one character at a time from STDIN to the buffer
   3. Check for invalid characters and set error flag if necessary
   4. Convert the character array (string) to an integer
*/
int readInt(int *error) {
    char buffer[1];
    char numStr[100];
    int num = 0, len = 0, i;

    if (numStr == NULL) {
        // Handle memory allocation failure
        *error = ERR;
        return 0;
    }

    while (1) {
        // Use the read system call to read one character from STDIN
        →   to buffer
        read(0, buffer, 1);

        if (buffer[0] == '\t' || buffer[0] == '\n' || buffer[0] == ' ')
        →   {
            break;
        } else if ((buffer[0] < '0' || buffer[0] > '9') && buffer[0] !=
        →   '-') {
            // Set error flag if the character is not a digit or a
            →   minus sign
            *error = ERR;
        } else {
            // Store the digit character in the numStr array
            numStr[len++] = buffer[0];
        }
    }

    // Check for invalid length or empty input
    if (len > 9 || len == 0) {
        *error = ERR;
        return 0;
    }

    // Convert the numStr array to an integer
    if (numStr[0] == '-') {
        if (len == 1) {
            // Set error flag if there is only a minus sign without
            →   digits
            *error = ERR;
```

```
130            return 0;
131        }
132        for (i = 1; i < len; i++) {
133            if (numStr[i] == '-') {
134                // Set error flag if there is more than one minus sign
135                *error = ERR;
136            }
137            num *= 10;
138            num += (int)(numStr[i] - '0');
139        }
140        // Make the number negative if the first character is a minus
       ↪   sign
141        num = -num;
142    } else {
143        for (i = 0; i < len; i++) {
144            if (numStr[i] == '-') {
145                // Set error flag if there is a minus sign in the
             ↪   middle of the number
146                *error = ERR;
147            }
148            num *= 10;
149            num += (int)(numStr[i] - '0');
150        }
151    }
152    return num;
153 }
```

This C code includes three functions: `printStr`, `printInt`, and `readInt`.

The `printStr` function prints a string to the standard output. It first calculates the length of the string by iterating through each character until it encounters the null character ($'0'$). Then, it uses the 'write' system call to print the string to the standard output. The number '1' is the file descriptor for standard output. The function finally returns the length of the string.

The `printInt` function prints an integer to the standard output. It first checks if the integer is zero, and if so, it simply adds '0' to the buffer. If the integer is negative, it adds a minus sign to the buffer and makes the integer positive. The function then converts the integer to a string in reverse order by continuously dividing the integer by 10 and adding the remainder to the buffer. After that, it reverses the string in the buffer. Finally, it uses the 'write' system call to print the integer string to the standard output.

The `readInt` function reads an integer from the standard input. It reads one character at a time from the standard input to a buffer. If the character is a tab, newline, or space, it breaks the loop. If the character is not a digit or a minus sign, it sets an error flag. Otherwise, it adds the character to a string. After reading all characters, it checks the length of the string and sets an error flag if the length is greater than 9 or if the string is empty. It then converts the string to an integer. If the first character of the string is a minus sign, it treats the number as negative. Finally, it returns the integer.

Supporting `char` type
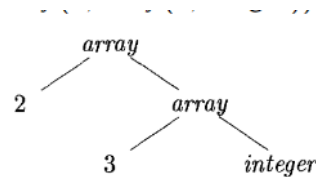
We used the same program as before.
Our Lexer and Parser recognizes the character inputs.
Other than storing global `char` seperately, the translator file remains more of less the same.
We generate a temporary variable in `primary_expression` of TYPE_CHAR of size 1. All other rules follow normally.

Supporting `arrays`

We followed the approach given in slides.

Array is just basically a linked list of SYMBOL_TYPE. In this the first `type` is a TYPE_PTR and the next follow up the type of array.

```
struct symboltype{
    enum symboltype_enum type;  // Type of symbol
    int width;                  // Size of Array (default 1)
    struct symboltype* ptr;     // Pointer to type of symbol (for
    ↪  TYPE_PTR and TYPE_ARRAY)
};
```

Next we edited emit functions accordingly. Eg:

```
direct_declarator : ...
            | IDENTIFIER L_BOX_BRACKET INTEGER_CONSTANT R_BOX_BRACKET {
              $1 = lookup(currST, $1->name);
              update_type($1, create_symboltype(pop(&var_type), 1, NULL));
              update_type($1, create_symboltype(TYPE_ARRAY, atoi($3), $1->type));
              $$ = $1;
          }
          ...
```

One change was done in the function calculating the size of data types. For a array, it recursively calls to find out the size. The other changes were made in the function generating ASM.

Supporting `char` and `char*` in function parameters

Since we already support the `char` and TYPE_PTR, we could add this support easily.
In `parameter_declaration` rule, we check if a parameter has a pointer associated with it, if yes we act accordindly

```
parameter_declaration : type_specifier pointer_opt identifier_opt {
        if($2 != NULL && $3 != NULL){
            $3 = lookup(currST, $3->name);
            struct symboltype* tempType = create_symboltype(TYPE_PTR, 1, NULL);
            tempType->ptr = create_symboltype(pop(&var_type), 1, NULL);
            update_type($3, tempType);
            $3->category = TYPE_PARAM;
            push_args(currST, $3);
        }
        else if($2 == NULL && $3 != NULL){
            $3 = lookup(currST, $3->name);
            update_type($3, create_symboltype(pop(&var_type), 1, NULL));
            $3->category = TYPE_PARAM;
            push_args(currST, $3);
        }
        else{
                // do nothing
        }
    }
;
```

The `parameter_declaration` rule is defined to handle three components: `type_specifier`, `pointer_opt`, and `identifier_opt`. These represent the type of the parameter, whether it's a pointer, and the identifier (name) of the parameter, respectively.

The code then checks if a parameter has a pointer associated with it. If both a pointer and an identifier are present ('2! = NULL && 3! = NULL'), it looks up the identifier in the current symbol table ("currST"), creates a new symbol type for a pointer, and updates the type of the identifier in the symbol table. It also sets the category of the identifier to `TYPE_PARAM` and pushes it to the argument list of the current symbol table. If there's no pointer but an identifier is present ('2 == NULL && 3! = NULL'), it performs similar actions but without creating a pointer type.

If neither a pointer nor an identifier is present, it does nothing. This is to handle the case for function declarations without any parameters or with void parameters.

We emit and store the count of the number of parameters in `argument_expression_list` by keeping a `paraCount` for each Symbol Table Entry of `TYPE_FUNC`.

Supporting `char` and `char*` in function return types

For this section, we just added one thing, to store the a `TYPE_RETURN` entry in a sumbol table which in turns stores the type of variable regardless of `void`, `int`, `char`, `PTR`, etc. We also added a return check to make sure that the function returning type matches the type of value being returned.

```
jump_statement : RETURN expression_opt SEMICOLON {
        if($2->returnLabel == 1){
            // return statement without any expression
            // function return type is void -- check
            if(!typecheck(currST->_retVal, create_symboltype(TYPE_VOID, 1, NULL))){
                yyerror("Return type mismatch with Function type");
```

```
                }
                $$ = create_statement();
                emit(OP_RETURN_VOID, NULL, NULL, NULL);
            }
            else{
                // check that the expression type is same as function return type
                if(!typecheck(currST->_retVal, $2->loc->type)){
                    yyerror("Return type mismatch with Function type");
                }
                $$ = create_statement();
                emit(OP_RETURN, NULL, NULL, $2->loc->name);
            }
        }
    ;
```

We did not implement the following bonus problems:

- Minimizing load-store by reusing variables in registers.

- Supporting type conversion from `char` to `int`, `int` to `char`, `int` to `bool` and `bool` to `int`.

- Supporting type conversion from any pointer type to any pointer type.

Converting `int` to `bool`

```
1   // Int to Bool
2   expression* int2bool(expression* expr){
3       if(expr->isBool == false){
4       // If the expression is not already a boolean, proceed with the
    ↪   conversion
5
6           expr->falseList = makelist(nextInstr());
7           // Create a list of instructions for the 'false' case.
8           // 'nextInstr()' returns the index of the next instruction to
    ↪   be executed.
9
10          emit(OP_EQUALS, expr->loc->name, "0", NULL);
11          // Emit an instruction to check if the expression is equal to
    ↪   0. Any non-zero value is considered 'true' and 0 is
    ↪   considered 'false'.
12
13          expr->trueList = makelist(nextInstr());
14          // Create a list of instructions for the 'true' case.
15
16          emit(OP_GOTO, NULL, NULL, NULL);
17          // Emit a 'goto' instruction, to skip over the 'true' case if
    ↪   the 'false' case is executed.
18      }
19      return expr;
```

```
20        // Return the modified expression.
21  }
```

Converting `bool` to `int`

```
1  // Bool to Int
2  expression* bool2int(expression* expr){
3      if(expr->isBool){
4      // If the expression is a boolean, proceed with the conversion
5
6          // Generate a new temporary variable of integer type
7          expr->loc = gentemp(create_symboltype(TYPE_INT, 1, NULL), NULL);
8
9          // backpatch the true list with next instruction
10         backpatch(expr->trueList, nextInstr());
11         // Update the 'true' list with the index of the next
          ↪  instruction to be executed
12
13         // emit the quad for true
14         emit(OP_ASSIGN, "true", NULL, expr->loc->name);
15         // Emit an instruction to assign the value 'true' (represented
          ↪  as 1) to the temporary variable
16
17         // goto the end of the false list
18         char str[100];
19         int pNext = nextInstr()+1;
20         sprintf(str, "%d", pNext);
21         emit(OP_GOTO, NULL, NULL, str);
22         // Emit a 'goto' instruction to skip to the end of the 'false'
          ↪  list
23
24
25         backpatch(expr->falseList, nextInstr());
26         // Update the 'false' list with the index of the next
          ↪  instruction to be executed
27
28
29
30         emit(OP_ASSIGN, "false", NULL, expr->loc->name);
31         // Emit an instruction to assign the value 'false' (likely
          ↪  represented as 0) to the temporary variable
32     }
33     return expr;
34  }
```

These two conversions are implemented in the `selection_statement` and `equality_expression` rules for grammar respectively as discussed in the slides.

TEST CASES Fails

Test Case 1:

```
1  int a;
```

We were able to generate its `.tac` file:

```
================================================================
ACTIVATION RECORD: Global
----------------------------------------------------------------
Name            Category        Offset          Nested Table
----------------------------------------------------------------
a               global          -24             -
================================================================


====================================================================
Symbol Table: Global                                Parent: NULL
--------------------------------------------------------------------
Name    Type    Category    Initial Value   Size    Offset  Nested Table
--------------------------------------------------------------------
a       int     global      0               4       0       -
====================================================================



============================
THREE ADDRESS CODE
----------------------------
NULL
============================
```

And the associated Assembly file

```
1          .globl          a
2          .data
3          .align 4
4          .type           a, @object
5          .size           a, 4
6  a:
7          .long           0
8          .text
9          .ident          "group-03-julius-stabs-back"
10         .section        .note.GNU-stack,"",@progbits
```

But we could not generate the final executable file.
On running the commands

```
    gcc -c test1.s
    gcc test1.o -o test1 -L. -lmyl -no-pie
```

We received the following error:

```
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/11/../../../x86_64-linux-gnu/crt1.o:
    in function `_start':
(.text+0x1b): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

This error states that, for an executable to build, we need an entry point which is the `main()` for `C` program files. Since we do not have that, the program does not compile. The rest of the program compiles successfully.

Test Case 3:

This test case compiled successfuly and generated an output file. However running the output file resulted in a segmentation fault.
The code for this test was:

```c
void func(int a, int *b);

void func(int a, int * b){

}
void main(){
    int a = 1;
    a = a + 1;
    int b = 1 - a;
    int c = 1/b;
    c = c * 2 + 10;
    int * d;
    *d = c * c % a;
}
```

The reason it resulted in a segmentation fault was due to like 12 and 13.
At line 12 a pointer is being declared and not initialized with any address. However we are trying to de-reference its memory location and storing some value there. Since the address does not exist, it results in a segmentation fault.
It is a run-time error and not a compile time error.

Test Case 5:

This test case does not pass the parsing phase itself.

```c
int d = 99;
int fun(int c){
    if(c%d == 0) return fun2(c);
    else return fun(c+1);
}
int * fun2(int b){
    int * e = 0;
    return e;
```

```
9   }
10  int main(){
11      int a = 1;
12      int * ret = fun(a);
13      return 0;
14  }
```

In test case above, the function `func2()` is being called before its definiton in `func()`. Since there is no forward definition present for `func2()`, it causes the error.

    Error: Function not defined for fun2 on ')' at line 3

We edited out this error by putting definition of `fun2()` initially. This will still result in error as the return type of `func2()` is of type `int*` where as the return type of function `fun()` is of type `int`. Hence when `fun()` is returning `fun2()`, it needs to return a `int` and not `int*`.
We make edit in line:

    if(c%d == 0) return *fun2(c);

To adjust for this error. Then the program runs correctly.

Test Case 6:

This test case does not pass the parsing phase itself.

```
1   int armstrong(int lim){
2       int num = 0;
3       int cnt = 0;
4       for (num = 0; num <= lim; num = num + 1) {
5           int copy = 0;
6           int sum = 0;
7           copy = num;
8           sum = 0;
9           while (num != 0) {
10              int rem = 0;
11              rem = num % 10;
12              sum = sum + (rem * rem * rem);
13              num = num / 10;
14          }
15      if (copy == sum) cnt = cnt + 1;
16      num = copy;
17      }
18      return cnt;
19  }
20  int main() {
21      int lim;
22      lim = 10000;
23
24      int cnt = armstrong(lim);
25  }
```

In the test case above, we are using `while()` loop. Our current grammar does not support functionality for `while()` loop, only `for()` loop. Hence it failed with error:

```
Error: Function not defined for while on ')' at line 9
```

Test Case 7:

This test case does not pass the parsing phase itself.

```
1   int choose(int n, int r){
2       if(r == 0) return 1;
3       if(n == 0) return 0;
4       return fact(n)/(fact(r) * fact(n-r));
5   }
6
7   int fact(int n){
8       if(n == 0) return 1;
9       return fact(n-1) * n;
10  }
11
12  int main() {
13      int n = 5; int r = 2;
14      int res = choose(n, r);
15  }
```

In test case above, the function `func2()` is being called before its definiton in `func()`. Since there is no forward definition present for `func2()`, it causes the error.

```
Error: Function not defined for fact on ')' at line 4
```

We edited out this error by putting definition of `fact()` initially.

Extra Credit Test Case 8:

This test case failed at parser stage itself. As we did not attempt the part for converting pointer types.

```
1   int main(){
2       char * s = "abc\n";
3       int * a = s;
4       void * b = a;
5       b = s;
6       s = b;
7       a = b;
8   }
```

Supporting type conversion from any pointer type to any pointer type is not implemented.

```
Error: Type mismatch in assignment expression on ';' at line 5
```