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:

```
.SUFFIXES:
GROUP = 3_A5
TEST_DIR = A5_Tests

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: 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}/test1.nc
	./compiler 2 ${TEST_DIR}/test2.nc
	./compiler 2 ${TEST_DIR}/test3.nc
	./compiler 2 ${TEST_DIR}/test4.nc
	./compiler 2 ${TEST_DIR}/test5.nc
	mv 3_A5_quads1.out ${TEST_DIR}/tac && mv 3_A5_quads1.asm ${TEST_DIR}/x86
	mv 3_A5_quads2.out ${TEST_DIR}/tac && mv 3_A5_quads2.asm ${TEST_DIR}/x86
	mv 3_A5_quads3.out ${TEST_DIR}/tac && mv 3_A5_quads3.asm ${TEST_DIR}/x86
	mv 3_A5_quads4.out ${TEST_DIR}/tac && mv 3_A5_quads4.asm ${TEST_DIR}/x86
	mv 3_A5_quads5.out ${TEST_DIR}/tac && mv 3_A5_quads5.asm ${TEST_DIR}/x86
```

```
	as ${TEST_DIR}/x86/3_A5_quads1.asm -o ${TEST_DIR}/x86/3_A5_quads1.o
	as ${TEST_DIR}/x86/3_A5_quads2.asm -o ${TEST_DIR}/x86/3_A5_quads2.o
	as ${TEST_DIR}/x86/3_A5_quads3.asm -o ${TEST_DIR}/x86/3_A5_quads3.o
	as ${TEST_DIR}/x86/3_A5_quads4.asm -o ${TEST_DIR}/x86/3_A5_quads4.o
	as ${TEST_DIR}/x86/3_A5_quads5.asm -o ${TEST_DIR}/x86/3_A5_quads5.o
	gcc ${TEST_DIR}/x86/3_A5_quads2.o -o ${TEST_DIR}/exe/3_A5_quads2 -L. -lmyl -no-pie
	gcc ${TEST_DIR}/x86/3_A5_quads3.o -o ${TEST_DIR}/exe/3_A5_quads3 -L. -lmyl -no-pie
	gcc ${TEST_DIR}/x86/3_A5_quads4.o -o ${TEST_DIR}/exe/3_A5_quads4 -L. -lmyl -no-pie
	gcc ${TEST_DIR}/x86/3_A5_quads5.o -o ${TEST_DIR}/exe/3_A5_quads5 -L. -lmyl -no-pie
	gcc ${TEST_DIR}/x86/3_A5_quads1.o -o ${TEST_DIR}/exe/3_A5_quads1 -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.

```
// Set Offsets after table is generated
void set_offset(symboltable* currST){
    int offset = 0;
    for(int i=0; i< currST->count; i++){
        symboltableentry* entry = (currST->table_entries[i]);
        if(entry->category == TYPE_FUNCTION){
            set_offset(entry->next);
        }
        entry->offset = offset;
        offset += entry->size;
    }
    return;
}
```

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.

```
#define MAX_HASH_AR 500

struct HashAR{
    char* key;
    int value;
    struct HashAR* next;
};
typedef struct HashAR HashAR;

unsigned int hash_ar(char *key);
void insert_ar(char *key, int value, HashAR* hashmap[]);
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:

```
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:

```
// 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;
```

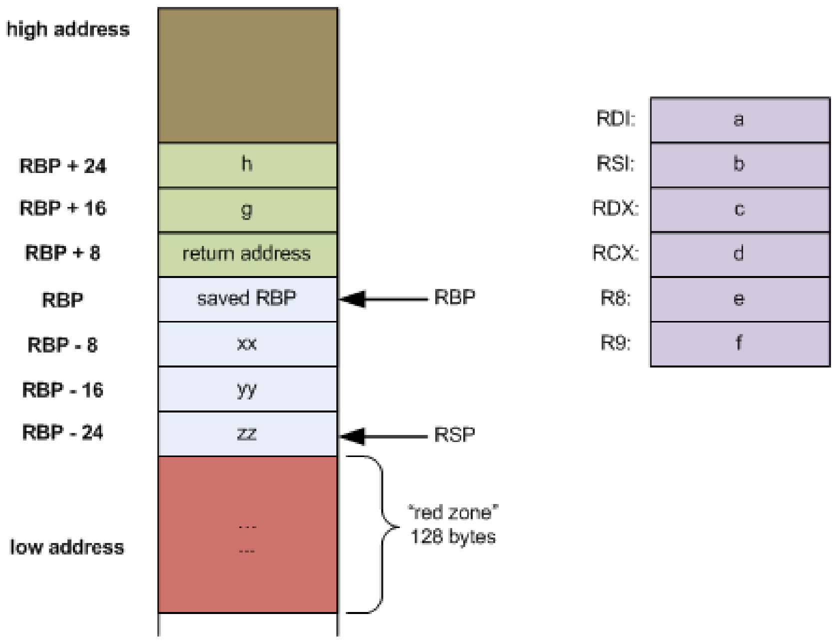
```c
}

// 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;
    }
    int hashIndex = hash_ar(key);
    HashAR* temp = hashmap[hashIndex];
    while(temp != NULL){
        if(strcmp(temp->key, key) == 0){
            return temp->value;
        }
        temp = temp->next;
    }
    return 0;
}
```

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:




```c
void gen_activation_record(symboltable* currST){
    int local = -20;
    int param = -24;

    // iterate over the symbol table entries
    for(int i=0; i < currST->count; i++){
        symboltableentry* entry = currST->table_entries[i];
        // printf("Entry: %s\n", entry->name);
        if(entry->category == TYPE_PARAM){
            // printf("Param: %s\n", entry->name);
            param += entry->size;
            insert_ar(entry->name, param, currST->_aRecord);
        }
        else if(entry->category == TYPE_RETURN){
            // printf("Local: %s\n", entry->name);
            continue;
        }
        else if(entry->category == TYPE_FUNCTION){
            // printf("Function: %s\n", entry->name);
            gen_activation_record(entry->next);
        }
        else{
            // printf("Local: %s\n", entry->name);
            local -= entry->size;
            insert_ar(entry->name, local, currST->_aRecord);
        }
    }
    return;
}
```

In the above code, The function `gen_activation_record(symboltable* currST)` begins 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

```c
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
```

```c
param_list* param_list_initialize(); // Function to initialize a new parameter list
param_list* param_list_delete(param_list* head); // Function to delete the parameter list
void param_list_insert(param_list* head, char* str); // Function to insert a new parameter into the parameter list
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:

```c
param_list* param_list_initialize(){
    param_list* head = (param_list*)malloc(sizeof(param_list));
    head->param = NULL;
    head->next = NULL;
    return head;
}

// insert a new string at the end of the linked list
void param_list_insert(param_list* head, char* str){
    // check if it is the first entry
    if(head->param == NULL){
        head->param = strdup(str);
        return;
    }
    // insert new entry at the end
    param_list* temp = (param_list*)malloc(sizeof(param_list));
    temp->param = strdup(str);
    temp->next = NULL;
    param_list* curr = head;
    while(curr->next != NULL){
        curr = curr->next;
    }
    curr->next = temp;
}

// delete the linked list end entry
```

```c
param_list* param_list_delete(param_list* head){
    param_list* curr = head;
    while(curr != NULL){
        param_list* temp = curr;
        curr = curr->next;
        free(temp);
    }
    head = param_list_initialize();
    return head;
}
```

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

```c
globalVars* _globalVars[MAX_HASH_GLOBAL];
struct globalVars{
    char* key; // The key stored in this node (name of the global variable)
    bool value; // scope of the global variable
    struct globalVars* next; / Pointer to the next node in the list
};
typedef struct globalVars globalVars;

unsigned int hash_global(char *key); //Generating a hash value for a given key
void insert_global(char *key, bool value, globalVars *hashmap[]); //inserting a new key-value pair into the hash table
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

```
// 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;
```

```
    }
    return false;
}

```

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.

```
HashLabel* _lablesRecord[MAX_HASH_LABEL];
struct HashLabel{
    int key;
    int value;
    struct HashLabel* next;
};
unsigned int hash_label(int key);
void insert_label(int key, int value, HashLabel *hashmap[]);
bool label_count(int key, HashLabel *hashmap[]);
HashLabel* label_at(int key, HashLabel *hashmap[]);
```

The function definitions are as follows:

```
// HASH for Label (return label)
unsigned int hash_label(int key){
    return key % MAX_HASH_LABEL;
}
void insert_label(int key, int value, HashLabel *hashmap[]){
    if(key < 0){
        // printf("Error: Key cannot be negative.\n");
        return;
    }
    int hashIndex = hash_label(key);
    HashLabel* temp = hashmap[hashIndex];
    while(temp != NULL){
```

```c
        if(temp->key == key){
            temp->value = value;
            return;
        }
        temp = temp->next;
    }
    HashLabel* newNode = (HashLabel*)malloc(sizeof(HashLabel));
    newNode->key = key;
    newNode->value = value;
    newNode->next = hashmap[hashIndex];
    hashmap[hashIndex] = newNode;
}
bool label_count(int key, HashLabel *hashmap[]){
    if(key < 0){
        // printf("Error: Key cannot be negative.\n");
        return false;
    }
    int hashIndex = hash_label(key);
    HashLabel* temp = hashmap[hashIndex];
    while(temp != NULL){
        if(temp->key == key){
            return true;
        }
        temp = temp->next;
    }
    return false;
}
// lable_at() -- return the reference to the label
HashLabel* label_at(int key, HashLabel *hashmap[]){
    if(key < 0){
        // printf("Error: Key cannot be negative.\n");
        return NULL;
    }
    int hashIndex = hash_label(key);
    HashLabel* temp = hashmap[hashIndex];
    while(temp != NULL){
        if(temp->key == key){
            return temp;
        }
        temp = temp->next;
    }
    return NULL;
}
```

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.

```
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]);
```

```
        if(entry->category != TYPE_FUNCTION){
            // It is a Global Variable
            // CHAR
            if(entry->type->type == TYPE_CHAR){
                if(entry->initial_value == NULL){
                    // printf("Global Char: %s\n", entry->name);
                    printf("\t.comm\t%s,1,1\n", entry->name);
                }
                else{
                    // printf("Global Char: %s = %s\n", entry->name, entry->initial_value);
                    printf("\t.globl\t%s\n", entry->name);
                    // printf("\t.data\n");
                    printf("\t.type\t%s, @object\n", entry->name);
                    printf("\t.size\t%s, 1\n", entry->name);
                    printf("%s:\n", entry->name);
                    printf("\t.byte\t%d\n", atoi(entry->initial_value));
                }
                // insert into global hashmap
                insert_global(entry->name, true, _globalVars);
            }
            // INT
            if(entry->type->type == TYPE_INT){
                if(entry->initial_value == NULL){
                    // printf("Global Int: %s\n", entry->name);
                    printf("\t.comm\t%s,4,4\n", entry->name);
                }
                else{
                    // printf("Global Int: %s = %s\n", entry->name, entry->initial_value);
                    printf("\t.globl\t%s\n", entry->name);
                    printf("\t.data\n");
                    printf("\t.align 4\n");
                    printf("\t.type\t%s, @object\n", entry->name);
                    printf("\t.size\t%s, 4\n", entry->name);
                    printf("%s:\n", entry->name);
                    // long -> int
                    printf("\t.long\t%d\n", atoi(entry->initial_value));
                }
                // insert into global hashmap
                insert_global(entry->name, true, _globalVars);
            }
            // ARRAY
            if(entry->type->type == TYPE_ARRAY){
                printf("\t.comm\t%s,%d,4\n", entry->name, entry->size);
                // insert into global hashmap
                insert_global(entry->name, true, _globalVars);
            }
```

```
        }
    }

    // STRINGS -- LL at string_head
    string_list* currString = string_head;
    // get size
    int string_head_size = ll_length(string_head);
    if(string_head_size>0){
        printf("\t.section\t.rodata\n");
        currString = string_head;
        while(currString != NULL){
            printf(".LC%d:\n", currString->entries);
            printf("\t.string\t%s\n", currString->str);
            currString = currString->next;
        }
    }

    // TEXT SECTION
    printf("\t.text \n");
    // initialize params list
    param_list* params_head = param_list_initialize();
    currST = globalST;
    bool make_quad = false;
    // iterate over the quadArray
    currentQArray = quadArray;

    while(currentQArray != NULL && currentQArray->arr != NULL &&
        currentQArray->count != 0){
        int iterator = currentQArray->count; // -1 ?
        if(label_count(iterator, _lablesRecord)){
            int count = label_at(iterator, _lablesRecord)->value;
            printf(".L%d:\n", 2 * _LabelCount + count + 2);
        }

        char* op = printOP(currentQArray->arr->op);
        char* arg1 = (currentQArray->arr->arg1 ==
            NULL)?(NULL):(strdup(currentQArray->arr->arg1));
        char* arg2 = (currentQArray->arr->arg2 ==
            NULL)?(NULL):(strdup(currentQArray->arr->arg2));
        char* result = (currentQArray->arr->result ==
            NULL)?(NULL):(strdup(currentQArray->arr->result));
        char* s = arg2;

        // Activation Record of Result
        char* result_ar;
        if(search_global(result, _globalVars)){
            // concatenate "(%rip)" to result
            result_ar = (char*)malloc(sizeof(char)*(strlen(result)+7));
```

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

        // Activation Record of Arg1
        char* arg1_ar;
        if(search_global(arg1, _globalVars)){
            // concatenate "(%rip)" to arg1
            arg1_ar = (char*)malloc(sizeof(char)*(strlen(arg1)+7));
            sprintf(arg1_ar, "%s(%%rip)", arg1);
        }
        else{
            //convert result of search_ar(arg1, currST->_aRecord) to string and concatenate "(%rbp)" to it
            int offset = search_ar(arg1, currST->_aRecord);
            arg1_ar = (char*)malloc(sizeof(char)*15);
            sprintf(arg1_ar, "%d(%%rbp)", offset);
        }

        // Activation Record of Arg2
        char* arg2_ar;
        if(search_global(arg2, _globalVars)){
            // concatenate "(%rip)" to arg2
            arg2_ar = (char*)malloc(sizeof(char)*(strlen(arg2)+7));
            sprintf(arg2_ar, "%s(%%rip)", arg2);
        }
        else{
            //convert result of search_ar(arg2, currST->_aRecord) to string and concatenate "(%rbp)" to it
            int offset = search_ar(arg2, currST->_aRecord);
            arg2_ar = (char*)malloc(sizeof(char)*15);
            sprintf(arg2_ar, "%d(%%rbp)", offset);
        }
        // parameter type
        if(op == "param"){
            // push arg1 to params_head
            param_list_insert(params_head, result);
        }
        else{
            printf("\t");
            // Binary Operations
            // Addition
```

```
            if (op == "+") {
                bool flag = true;
                if(s==NULL || ((!isdigit(s[0])) && (s[0] != '-') && (s[0] != '+'))){
                    flag = false;
                }
                else{
                    char* p;
                    strtol(s, &p, 10);
                    if(*p == 0)
                        flag = true;
                    else
                        flag = false;
                }
                if(flag){
                    printf("addl \t$%d, %s\n", atoi(arg2), arg1_ar);
                }
                else{
                    // AMBIGIOUS
                    printf("\tmovl\t%s, %%eax\n", arg1_ar);
                    printf("\tmovl\t%s, %%edx\n", arg2_ar);
                    printf("\taddl \t%%edx, %%eax\n");
                    printf("\tmovl\t%%eax, %s\n", result_ar);
                }
            }
            // Subtraction
            else if (op == "-"){
                printf("\tmovl\t%s, %%eax\n", arg1_ar);
                printf("\tmovl\t%s, %%edx\n", arg2_ar);
                printf("\tsubl \t%%edx, %%eax\n");
                printf("\tmovl\t%%eax, %s\n", result_ar);
            }
            // multiplication
            else if(op == "*"){
                printf("\tmovl\t%s, %%eax\n", arg1_ar);
                bool flag = true;
                if(s==NULL || ((!isdigit(s[0])) && (s[0] != '-') && (s[0] != '+'))){
                    flag = false;
                }
                else{
                    char* p;
                    strtol(s, &p, 10);
                    if(*p == 0)
                        flag = true;
                    else
                        flag = false;
                }
```

```
            if(flag){
                printf("# %s = %s * %s\n", result, arg1, arg2);
                printf("\timull \t$%d, %%eax\n", atoi(arg2));
                symboltable* tempTab = currST;
                char* val;
                // check if arg1 is a global variable
                for(int i=0; i <tempTab->count; i++){
                    if(strcmp((tempTab->table_entries[i])->name, arg1) == 0){
                        val = strdup((tempTab->table_entries[i])->initial_value); // value found, propagate the name
                    }
                }
            }
            else{
                printf("\timull \t%s, %%eax\n", arg2_ar);
                printf("\tmovl\t%%eax, %s\n", result_ar);
            }
        }
        // division
        else if(op=="/"){
            printf("\tmovl\t%s, %%eax\n", arg1_ar);
            printf("\tcltd\n");
            printf("\tidivl \t%s\n", arg2_ar);
            printf("\tmovl\t%%eax, %s\n", result_ar);
        }
        // modulo
        else if(op == "%"){
            printf("\tmovl\t%s, %%eax\n", arg1_ar);
            printf("\tcltd\n");
            printf("\tidivl \t%s\n", arg2_ar);
            printf("\tmovl\t%%edx, %s\n", result_ar);
        }
        // assign
        else if(op =="="){
            if(make_quad == true){
                printf("\tmovq \t%s, %%rax\n", arg1_ar);
                printf("\tmovq \t%%rax, %s\n", result_ar);
                make_quad = false;
            }
            else{
                s = arg1;//(arg1 == NULL)?(NULL):(strdup(arg1));
                bool flag = true;
                if(s==NULL || ((!isdigit(s[0])) && (s[0] != '-') && (s[0] != '+'))){
                    flag = false;
                }
```

```
                    else{
                        char* p;
                        strtol(s, &p, 10);
                        if(*p == 0)
                            flag = true;
                        else
                            flag = false;
                    }
                    if(flag){
                        printf("movl\t$%d, %%eax\n", atoi(arg1));
                    }
                    else{
                        printf("\tmovl\t%s, %%eax\n", arg1_ar);
                    }
                    printf("\tmovl\t%%eax, %s\n", result_ar);
                }
            }
            else if(op=="=str"){
                printf("movq \t$.LC%s, %s\n", arg1, result_ar);
            }
            // Relational
            else if(op=="=="){
                printf("\tmovl\t%s, %%eax\n", arg1_ar);
                printf("\tcmpl\t%s, %%eax\n", arg2_ar);
                int tempCount = label_at(atoi(result),
                 ↪  _lablesRecord)->value;
                printf("\tje .L%d\n", 2 * _LabelCount + tempCount + 2);
            }
            else if(op=="!="){
                printf("\tmovl\t%s, %%eax\n", arg1_ar);
                printf("\tcmpl\t%s, %%eax\n", arg2_ar);
                int tempCount = label_at(atoi(result),
                 ↪  _lablesRecord)->value;
                printf("\tjne .L%d\n", 2 * _LabelCount + tempCount + 2);
            }
            else if(op=="<"){
                printf("\tmovl\t%s, %%eax\n", arg1_ar);
                printf("\tcmpl\t%s, %%eax\n", arg2_ar);
                int tempCount = label_at(atoi(result),
                 ↪  _lablesRecord)->value;
                printf("\tjl .L%d\n", 2 * _LabelCount + tempCount + 2);
            }
            else if(op==">"){
                printf("\tmovl\t%s, %%eax\n", arg1_ar);
                printf("\tcmpl\t%s, %%eax\n", arg2_ar);
                int tempCount = label_at(atoi(result),
                 ↪  _lablesRecord)->value;
                printf("\tjg .L%d\n", 2 * _LabelCount + tempCount + 2);
```

```
            }
            else if(op=="<="){
                printf("\tmovl\t%s, %%eax\n", arg1_ar);
                printf("\tcmpl\t%s, %%eax\n", arg2_ar);
                int tempCount = label_at(atoi(result),
                ↪  _lablesRecord)->value;
                printf("\tjle .L%d\n", 2 * _LabelCount + tempCount + 2);
            }
            else if(op==">="){
                printf("\tmovl\t%s, %%eax\n", arg1_ar);
                printf("\tcmpl\t%s, %%eax\n", arg2_ar);
                int tempCount = label_at(atoi(result),
                ↪  _lablesRecord)->value;
                printf("\tjge .L%d\n", 2 * _LabelCount + tempCount + 2);
            }
            else if(op=="goto"){
                if (result != NULL) {
                    int tempCount = label_at(atoi(result),
                    ↪  _lablesRecord)->value;
                    printf("jmp .L%d\n", 2 * _LabelCount + tempCount +
                    ↪  2);
                }
            }

            // Unary Operations
            else if(op == "=&"){
                printf("# %s = &%s\n", result, arg1);
                printf("\tleaq\t%s, %%rax\n", arg1_ar);
                printf("\tmovq \t%%rax, %s\n", result_ar);
                make_quad = true;
            }
            else if(op=="=*"){
                printf("# %s = *%s\n", result, arg1);
                printf("\tmovq\t%s, %%rax\n", arg1_ar);
                printf("\tmovl\t(%%rax), %%eax\n");
                printf("\tmovl\t%%eax, %s\n", result_ar);
            }
            else if(op=="*="){
                printf("# *%s = %s\n", result, arg1);
                printf("\tmovl\t%s, %%eax\n", result_ar);
                printf("\tmovl\t%s, %%edx\n", arg1_ar);
                // cout << "\tmovl\t%edx, (%eax)";
                printf("\tmovl\t%%edx, (%%eax)\n");
            }
            else if(op=="uminus"){
                printf("\tmovl\t%s, %%eax\n", arg1_ar);
                printf("\tnegl\t%%eax\n");
                printf("\tmovl\t%%eax, %s\n", result_ar);
```

```
            }
            else if(op == "=[]"){
                printf("# =[] operation ; ");
                printf("%s = %s[%s]\n", result, arg1, arg2);
                if(search_global(arg1, _globalVars)){
                    printf("\tmovl\t%s, %%eax\n", arg2_ar);
                    printf("\tmovslq\t%%eax, %%rdx\n");
                    printf("\tleaq\t0(,%%rdx,4), %%rdx\n");
                    printf("\tleaq\t%s, %%rax\n", arg1_ar);
                    printf("\tmovl\t(%%rdx,%%rax), %%eax\n");
                    printf("\tmovl\t%%eax, %s\n", result_ar);
                }
                else{
                    printf("\tmovl\t%s, %%ecx\n", arg2_ar);
                    printf("\tmovl\t%d(%%rbp,%%rcx,4), %%eax\n", search_ar(arg1, currST->_aRecord));
                    printf("\tmovl\t%%eax, %s\n", result_ar);
                }
            }
            else if(op=="[]="){
                printf("# []= operation ; ");
                printf("%s[%s] = %s\n", result, arg1, arg2);
                if(search_global(result, _globalVars)){
                    printf("\tmovl\t%s, %%eax\n", arg2_ar);
                    printf("\tmovl\t%s, %%edx\n", arg1_ar);
                    printf("\tmovslq\t%%edx, %%rdx\n");
                    printf("\tleaq\t0(,%%rdx,4), %%rcx\n");
                    printf("\tleaq\t%s, %%rdx\n", result_ar);
                    printf("\tmovl\t%%eax, (%%rcx,%%rdx)\n");
                }
                else{
                    printf("\tmovl\t%s, %%eax\n", arg1_ar);
                    printf("\tmovl\t%s, %%edx\n", arg2_ar);
                    printf("\tmovl\t%%edx, %d(%%rbp,%%rax,4)\n", search_ar(result, currST->_aRecord));
                }
            }
            else if(op=="return"){
                if(result != NULL){
                    printf("\tmovl\t%s, %%eax\n", result_ar);
                }
                // jump to the end of the function -- epilogue
                printf("\tjmp .LFE%d\n", _LabelCount);
            }
            else if(op=="param"){
                // push arg1 to params_head
                param_list_insert(params_head, result);
            }
```

```

            // function call
            else if(op=="call"){
                // 4 registers are used for passing parameters -- rdi, rsi, rdx, rcx
                param_list* tempPara = params_head;

                int paraCOUNT = 0;
                while(tempPara != NULL){
                    paraCOUNT++;
                    tempPara = tempPara->next;
                }
                tempPara = params_head;

                for(int i=0; i<paraCOUNT; i++){
                    if(i==0){
                        // first parameter
                        int val = search_ar(tempPara->param, currST->_aRecord);
                        printf("movl\t%d(%%rbp), %%eax\n", val);
                        printf("\tmovq\t%d(%%rbp), %%rdi\n", val);
                    }
                    else if(i==1){
                        int val = search_ar(tempPara->param, currST->_aRecord);
                        printf("movl\t%d(%%rbp), %%eax\n", val);
                        printf("\tmovq\t%d(%%rbp), %%rsi\n", val);
                    }
                    else if(i==2){
                        int val = search_ar(tempPara->param, currST->_aRecord);
                        printf("movl\t%d(%%rbp), %%eax\n", val);
                        printf("\tmovq\t%d(%%rbp), %%rdx\n", val);
                    }
                    else if(i==3){
                        int val = search_ar(tempPara->param, currST->_aRecord);
                        printf("movl\t%d(%%rbp), %%eax\n", val);
                        printf("\tmovq\t%d(%%rbp), %%rcx\n", val);
                    }
                    else{
                        int val = search_ar(tempPara->param, currST->_aRecord);
                        printf("\tmovq\t%d(%%rbp), %%rdi\n", val);
                    }
                    tempPara = tempPara->next;
                }
                // clear para stack
```

```
                params_head = param_list_delete(params_head);
                printf("\tcall\t%s\n", arg1);
                printf("\tmovl\t%%eax, %s\n", result_ar);
            }

            else if(op == "function"){
                // function begins -- prologue
                printf(".globl\t%s\n", result);
                printf("\t.type\t%s, @function\n", result);
                printf("%s: \n", result);
                printf(".LFB%d: \n", _LabelCount);
                printf("\t.cfi_startproc\n");
                printf("\tpushq\t%%rbp\n");
                printf("\t.cfi_def_cfa_offset 8\n");
                printf("\t.cfi_offset 5, -8\n");
                printf("\tmovq\t%%rsp, %%rbp\n");
                printf("\t.cfi_def_cfa_register 5\n");
                currST = lookup(globalST, result)->next;
                // get last entry of the symbol table -- count starts from 0
                symboltableentry* lastEntry = currST->table_entries[currST->count-1];
                // get the size of the symbol table
                int sizeTemp = lastEntry->offset;
                // rsp register holds the address of the top of the stack
                printf("\tsubq\t$%d, %%rsp\n", sizeTemp+24);  // MAX BUFFER: 4 Para + retVal + RA = 24 = (4+1+1)*4

                // function table -- paramaters section
                for(int i=0; i < currST->paramCount; i++){
                    symboltableentry* entry = currST->_argList[i];
                    if(i==0){
                        printf("\tmovq\t%%rdi, %d(%%rbp)\n", search_ar(entry->name, currST->_aRecord));
                    }
                    else if(i==2){
                        printf("\tmovq\t%%rsi, %d(%%rbp)\n", search_ar(entry->name, currST->_aRecord));
                    }
                    else if(i==3){
                        printf("\tmovq\t%%rdx, %d(%%rbp)\n", search_ar(entry->name, currST->_aRecord));
                    }
                    else if(i==4){
                        printf("\tmovq\t%%rcx, %d(%%rbp)\n", search_ar(entry->name, currST->_aRecord));
                    }
```

```
                }
            }
            // function end -- epilogue
            else if(op=="end"){
                printf(".LFE%d: \n", _LabelCount++);
                printf("leave\n");
                printf("\t.cfi_restore 5\n");
                printf("\t.cfi_def_cfa 4, 4\n");
                printf("\tret\n");
                printf("\t.cfi_endproc\n");
                printf("\t.size\t%s, .-%s\n", result, result);
            }
            else{
                printf("op: %s\n", op);
            }
            // printf("\n");
        }
        currentQArray = currentQArray->nextQuad;
    }
    // footer
    printf("\t.ident\t\"group-03-julius-stabs-back\"\n");
    printf("\t.section\t.note.GNU-stack,\"\",@progbits\n");
}
```

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:

```
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);
```

```
    currST = globalST;
    printf("Initializing Variable STack\n");
    stack_intialize(&var_type);
    printf("Initializing String Linked List\n");
    string_head = string_list_initialize();
    printf("Initializing Quad Array\n");
    quadArray = quadArray_initialize(quadArray);

    printf("Starting Parser\n");
    // send input to parser
    FILE* file = fopen(inname, "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    yyin = file;
    yyparse();
    printf("Parser Done, Updating Symbol Tables Offset\n");
    set_offset(globalST);
    printf("Generating Activation Records\n");
    gen_activation_record(globalST);

    if(strcmp(argv[1], "1") == 0){
        printf("Writing TACs\n");

        // open file for writing outname.out
        char* temp_o = "3_A5_quads";
        char* extension_out = (char*)malloc(sizeof(char)*6);
        sprintf(extension_out, "%c.out", outname);
        char* outname_out = (char*)malloc(sizeof(char)*25);
        sprintf(outname_out, "%s%s", temp_o, extension_out);

        FILE* file_out = fopen(outname_out, "w");
        if (!file_out) {
            perror("Error opening file");
            return 1;
        }
        printf("\tWriting TACs -- Activation Record\n");
        print_activationRecord(file_out, globalST);
        printf("\tWriting TACs -- Symbol Table\n");
        print_ST(file_out, globalST);
        printf("\tWriting TACs -- TAC\n");
        print_quadArray(file_out, quadArray);
        printf("\n\n\n");
        // close file
        fclose(file_out);
    }
```

```
    if(strcmp(argv[1], "2") == 0){
        printf("Writing TACs\n");

        // open file for writing outname.out
        char* temp_o = "3_A5_quads";
        char* extension_out = (char*)malloc(sizeof(char)*6);
        sprintf(extension_out, "%c.out", outname);
        char* outname_out = (char*)malloc(sizeof(char)*25);
        sprintf(outname_out, "%s%s", temp_o, extension_out);

        FILE* file_out = fopen(outname_out, "w");
        if (!file_out) {
            perror("Error opening file");
            return 1;
        }
        printf("\tWriting TACs -- Activation Record\n");
        print_activationRecord(file_out, globalST);
        printf("\tWriting TACs -- Symbol Table\n");
        print_ST(file_out, globalST);
        printf("\tWriting TACs -- TAC\n");
        print_quadArray(file_out, quadArray);
        printf("TAC DONE\n\n");
        // close file
        fclose(file_out);
        printf("Generating ASM\n");

        // open file for writing outname.asm
        char* temp_s = "3_A5_quads";
        char* extension_asm = (char*)malloc(sizeof(char)*4);
        sprintf(extension_asm, "%c.s", outname);
        char* outname_asm = (char*)malloc(sizeof(char)*25);
        sprintf(outname_asm, "%s%s", temp_s, extension_asm);

        FILE* file_asm = fopen(outname_asm, "w");
        if (!file_asm) {
            perror("Error opening file");
            return 1;
        }
        tac2x86(file_asm);
        // close file
        fclose(file_asm);
        printf("ASM DONE\n\n\n");
    }
    fclose(file);
    return 0;
}
```

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;
    }

```

```
    // Convert the integer to a character array (string) in reverse order
    if (num == 0) {
        buffer[index++] = '0';
    }
        else {
        if (num < 0) {
            buffer[index++] = '-';
            num = -num;
        }

        while (num) {
            int digit = num % 10;
            buffer[index++] = (char)(digit + '0');
            num /= 10;
        }

        // Reverse the string if it's a negative number
        if (buffer[0] == '-') {
            int start = 1, end = index - 1;
            while (start < end) {
                char temp = buffer[start];
                buffer[start++] = buffer[end];
                buffer[end--] = temp;
            }
        }
                else {
            // Reverse the string if it's a positive number
            int start = 0, end = index - 1;
            while (start < end) {
                char temp = buffer[start];
                buffer[start++] = buffer[end];
                buffer[end--] = temp;
            }
        }
    }
        // Calculate the number of bytes in the string
    bytes = index;

    // Use the write system call to print the integer string to STDOUT
    write(1, buffer, bytes);

    return bytes;
}

/*
A function to read an integer from STDIN
We go through the following steps:
```

```
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;
```

```
            return 0;
        }
        for (i = 1; i < len; i++) {
            if (numStr[i] == '-') {
                // Set error flag if there is more than one minus sign
                *error = ERR;
            }
            num *= 10;
            num += (int)(numStr[i] - '0');
        }
        // Make the number negative if the first character is a minus sign
        num = -num;
    } else {
        for (i = 0; i < len; i++) {
            if (numStr[i] == '-') {
                // Set error flag if there is a minus sign in the middle of the number
                *error = ERR;
            }
            num *= 10;
            num += (int)(numStr[i] - '0');
        }
    }
    return num;
}
```

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.

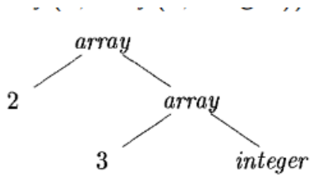
The `readInt` function reads an integer from the standard input. It reads one character at a time from the standard input to a buffer. If the character is a tab, newline, or space, it breaks the loop. If the character is not a digit or a minus sign, it sets an error flag. Otherwise, it adds the character 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.

TEST CASES Fails

Test Case 1:

```c
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

```
        .globl        a
        .data
        .align 4
        .type        a, @object
        .size        a, 4
a:
        .long        0
        .text
        .ident        "group-03-julius-stabs-back"
        .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:

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

void func(int a, int * b){

}
void main(){
    int a = 1;
    a = a + 1;
    int b = 1 - a;
```

```
10      int c = 1/b;
11      c = c * 2 + 10;
12      int * d;
13      *d = c * c % a;
14  }
```

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.