CS-1319: PLDI - Monsoon 23 **Team Name:** julius-stabs-back

Assignment #4

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. Fronsto (GitHub)
2. RKJenamani (GitHub)
3. Brainstorming with Group 26.

Makefile

Our main `c` file looks as follows:

```
.SUFFIXES:
translator: 3_A4.y 3_A4_translator.c 3_A4.l
        bison -d 3_A4.y
        flex -o lex.yy.c 3_A4.l
        gcc 3_A4_translator.c 3_A4.tab.c lex.yy.c -lfl -Werror -o
         ↪ translator
build: translator
test: translator
        ./translator < 3_A4.nc > test.out
clean:
        rm -f translator lex.yy.c 3_A4.tab.c 3_A4_translator.tab.h
         ↪ test.out 3_A4.tab.o 3_A4.tab.h
```

Header File: `3_A4_translator.h`

SYMBOL TABLE

The Global Symbol Table:

Symbol Table: Global Parent: NULL

Name	Type	Category	Initial Value	Size	Nested Table
printInt	function	NULL	-	0	printInt
readInt	function	NULL	-	0	readInt
printStr	function	NULL	-	0	printStr
arr	array(10,int)	global	0	40	-
binarySearch	function	NULL	-	0	binarySearch
main	function	NULL	-	0	main

The Nested Symbol Table:

Symbol Table: binarySearch   Parent: ST.Global

Name	Type	Category	Initial Value	Size	Nested Table
l	int	param	-	4	-
r	int	param	-	4	-
x	int	param	-	4	-
retValue	int	local	-	4	-
mid	int	local	-	4	-
t2	int	temp	-	4	-
t3	int	temp	2	4	-
t4	int	temp	-	4	-
t5	int	temp	-	4	-
t6	int	temp	-	4	-
t7	int	temp	-	4	-
t8	int	temp	1	4	-
t9	int	temp	-	4	-
t10	int	temp	-	4	-
t11	int	temp	1	4	-
t12	int	temp	-	4	-
t13	int	temp	-	4	-
t14	int	temp	1	4	-
t15	int	temp	-	4	-

The structure of the symbol table is as follows:

```
struct symboltable{                 // Structure of a symbol table (TABLE)
    char* name;                     // Name of symbol table
    struct symboltable* parent; // Pointer to parent symbol table
    int count;                      // Count of entries in symbol table
    // int tempCount;                  // Count of temporary variables in symbol table
    int paramCount;                 // Count of parameters in symbol table
    symboltableentry** _argList; // List of arguments of function
    symboltableentry** table_entries;    // Pointer to entries in symbol table -- linked list of entries
    symboltype* _retVal;            // Return type of function
    struct symboltable* next;             // Pointer to next symbol table
    int returnLabel;                // Return label
};
typedef struct symboltable symboltable;
```

The `symboltable` is a dynamic block structure which will have entries in a lined list fashion. The `next` is the entry linking one table to another (not implemented as nested blocks may be skipped). We keep the count of number of entries. `_argList` (Parameter List) and `table_entries` are the Linked List data structure of type `symboltableentry`. Each row is `symboltableentry` which is linked to every next `symboltableentry`.

The structure of the symboltableentry is as follows:

```
struct symboltableentry{            // Structure of a symbol table entry (ROW) for variables
    char* name;                     // Name of symbol
```

```
    symboltype* type;           // Type of symbol
    char* initial_value;         // Initial value of symbol
    int size;                   // Size of symbol
    int offset;                 // Offset of symbol
    enum category_enum category; // Category of symbol
    struct symboltable* next; // Pointer to next symboltableentry --
    ↪   nested
};
typedef struct symboltableentry symboltableentry;
```

The symbol table entry is a row entry containing name, `symboltype* type`, initial value, size, offset, category and the link to next row entry as Linked List.

Symbol Type:

```
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)
};
typedef struct symboltype symboltype;
```

The the need for `symboltype` is because of implementation of arrays and pointers. If a symbol is a pointer, the next symboltype LinkedList stores the type of pointer. The width stores the size of the array.

Auxiliary enums:

```
enum symboltype_enum {
    TYPE_VOID,
    TYPE_INT,
    TYPE_CHAR,
    TYPE_PTR,
    TYPE_FUNC,
    TYPE_ARRAY,
    TYPE_STRING,
    TYPE_BLOCK
};
enum category_enum {
    TYPE_LOCAL,
    TYPE_GLOBAL,
    TYPE_PARAM,
    TYPE_TEMP
};
```

Function Definitions associated with Symbol Table:

```c
symboltableentry *lookup(symboltable* currST, char* yytext); // Lookup a symbol in the symbol table
symboltableentry* parentLookup(symboltable* currST, char* yytext); // Lookup a symbol in the parent symbol table
symboltable* create_symboltable(char* name, symboltable* parent); // Create a new symbol table
symboltype* create_symboltype(enum symboltype_enum type, int width, symboltype* ptr); // Create a new symbol type
symboltableentry* gentemp(symboltype* type, char* initial_value); // Generate a temporary variableint get_size(symboltype* type); // Get the width of a symbol
symboltableentry* genparam(symboltype* type, char* initial_value); // Generate a parameter
void update_return_ST(symboltable* currST, int update); // Update the return type of a function
void update_type(symboltableentry* entry, symboltype* type); // Update the type of a symbol
void print_ST(symboltable *currST); // Print the symbol table
void update_ST(symboltable* currST, symboltableentry* entry); // Update the symbol table
char* printType(symboltype* type); // Print the type of a symbol
char* printCategory(enum category_enum category); // Print the category of a symbol
int typecheck(symboltype* type1, symboltype* type2); // Check if two types are equal
void push_args(symboltable* currST, symboltableentry* arg); // Push arguments to the symbol table
```

QUADS and TAC

Quads and QuadArrays:

```c
struct quad{
    enum op_code op;                    // Operator
    char* arg1;                // Argument 1
    char* arg2;                // Argument 2
    char* result;              // Result
};
typedef struct quad quad;
struct qArray{         // Linked list of quads
    quad* arr;            // Array of quads
    int count;            // Total number of quads
    struct qArray* nextQuad; // Pointer to next qArray
};
typedef struct qArray qArray;
```

A quad is the collection of `operator`, `arg1`. `agr2`, and `result`.

A `quadArray` is an linkedList which stores all the quads. These will be used to store and print the Three Address Code.

The functions associated with these structures are:

```
void print_quadArray(qArray* head); // Print the quads
void print_quad(quad* q);              // Print a single quad
void emit(enum op_code op, char* arg1, char* arg2, char* result);  //
    ↪ Emit a quad -- add to quadArray
char* printOP(enum op_code op); // Print the operator
int nextInstr(); // Get the next instruction number
qArray* quadArray_initialize(qArray* head); // Initialize the quadArray
```

Expression, Statement, and Function Structures

These data structures are associated with the grammar rules for statements, and expressions.

```
struct expression{
    symboltableentry *loc; // Pointer to symbol table entry of variable
    symboltableentry *arrBase; // Pointer to symbol table entry of
        ↪ array base
    bool isBool;          // Is expression boolean?
    bool isPtr;           // Is expression pointer?
    bool isArray;         // Is expression array?
    bool isFunc;          // Is expression function?
    int* trueList;        // List of true labels
    int* falseList;       // List of false labels
    int* nextList;        // List of next labels
    int returnLabel;      // Return label
};
typedef struct expression expression;

struct statement{
    int* nextList;        // List of next labels
    int returnLabel;      // Return label
    enum symboltype_enum Type; // Does statement have a type?
};
typedef struct statement statement;
```

An expression holds a pointer to location of symbol table row (en entry) which can be a normal entry or an array entry.
The expression also holds the checks for `isBool, isArray, isPtr, isFunc`.
These will be helpful when dealing with different grammar rules handling each of these specifics.
The `trueList, falseList, nextList` are an array of lables.. The `returnLabel` is a check associated with return of a function. This will be helpful when dealing with return values.

The Statement holds the `nextList` and the `returnLabel`, again helpful for handling dangling statements and function calls. The functions associated with expressions and statements are:

```
void backpatch(int* list, int label); // Backpatch a list of labels
↪   with a label
int* makelist(int label); // Make a list of labels
int* merge(int* list1, int* list2); // Merge two lists of labels
statement* create_statement();
expression* create_expression();
expression* bool2int(expression* e); // Convert a boolean expression to
↪   an integer expression
expression* int2bool(expression* e); // Convert an integer expression
↪   to a boolean expression
```

VARIABLE STACK

We use a Stack for holding the incoming variable types (`INT, CHAR, VOID`) and a Linkedlist to hold the incoming Strings.

```
// stack for variable type
struct var_type_stack{
    enum symboltype_enum type[MAX_STACK]; // Type of symbol
    int top;                    // Top of stack
};
typedef struct var_type_stack var_type_stack;

// linked list for string
struct string_list{
    char* str;
    int entries;
    struct string_list* next;
};
typedef struct string_list string_list;
```

The auxiliary helper functions of stack and Linkedlist are defined below:

```
// Stack Functions
void stack_intialize(var_type_stack *s);
void push(var_type_stack* stack, enum symboltype_enum type); // Push a
↪   type to the stack
enum symboltype_enum pop(var_type_stack *s); // Pop a type from the
↪   stack
// String List Functions
string_list* string_list_initialize();
void ll_insert(string_list* head, char* str);
void ll_delete(string_list* head);
```

A simple logic tells us that we push the incoming varibale type on the `type_specifier` grammar rules whcih expands to the varibale tipe and , pop on `IDENTIFIER` where we associate the type with name of variable.

Other

The remaing things in header file are as follows:

```
#define size_of_void 0;
#define size_of_char 1;
#define size_of_int 4;
#define size_of_pointer 4;
#define MAX_STACK 100

// Global Symbol Tables
extern struct symboltable* currST;       // Current Symbol Table
extern struct symboltable* globalST;     // Global Symbol Table
extern struct symboltable* new_ST;       // New Symbol Table -- used in function declaration
extern struct var_type_stack var_type;// stack for storing the type of the variable
extern struct string_list* string_head; // head of the string list
extern char* yytext;
extern void yyerror(char *s);
extern int yyparse(void);
```

These define the sizes of types and externally declare the variable to be able to use across all files.

C File: `3_A4_translator.c`

main()

We start by initializing the symbol tables. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. The $'globalST'$ is the global symbol table and $'currST'$ is set to $'globalST'$. The stack $'var\_type'$ and the string list $'string_head'$ are initialized. These are used to hold variable types and strings respectively during the parsing process. Then, the quad array $'quadArray'$ is initialized. Each entry in the quad array represents an operation that needs to be performed. We then start parser with the $'yyparse'$ function. This function will parse the input according to the grammar rules defined in the yacc file. During this, it will populate the symbol tables, the quad array, and other data structures with the information extracted from the source code. After the parsing is complete, we are printing the global symbol table and the quad array.

```
int main(){
    printf("Initializing Symbol Tables\n");
```

```
    globalST = create_symboltable("Global", NULL);
    currST = globalST;
    stack_intialize(&var_type);
    string_head = string_list_initialize();
    quadArray = quadArray_initialize(quadArray);
    // gentemp test
    // symboltableentry* temp = gentemp(create_symboltype(TYPE_INT, 1, NULL), "69");
    // gentemp update test
    printf("Starting Parser\n");
    yyparse();
    printf("Global Symbol Table:\n");
    print_ST(globalST);
    printf("\n\n\n");
    print_quadArray(quadArray);
    return 0;
}
```

Global Declarations

```
qArray* quadArray;               // pointer to the head of the quad array linked list
var_type_stack var_type;         // declare the stack
string_list* string_head;        // linked list for string literals
symboltable* globalST;           // pointer to Global Symbol Table
symboltable* currST;             // pointer to Current Symbol Table
symboltable* new_ST;             // pointer to new Symbol Table  -- used in function declaration
static int tempCount = 0;        // count of the temporary variables
```

Expression, Statement, and Function Structures Functions

create_expression()

We initialize all fields to NULL and the nextList is given a value of -1 to represent the end. Output is a pointer to expression.

```
expression* create_expression(){
    expression* newExp = (expression*)malloc(sizeof(expression));
    newExp->isBool = false;
    newExp->isArray = false;
    newExp->isPtr = false;
    newExp->isFunc = false;
    newExp->loc = NULL;
    newExp->arrBase = NULL;
    newExp->trueList = NULL;
```

```
    newExp->falseList = NULL;
    newExp->nextList = (int*)malloc(sizeof(int));
    newExp->nextList[0] = -1;
    newExp->returnLabel = 0;
    return newExp;
}
```

`create_statement()`

We initialize returnLable to 0 and the nextList is given a value of -1 to represent the end. Output is a pointer to `statement`.

```
statement* create_statement(){
    statement* newStmt = (statement*)malloc(sizeof(statement));
    // end of list is -1
    newStmt->nextList = (int*)malloc(sizeof(int));
    newStmt->nextList[0] = -1;
    newStmt->returnLabel = 0;
    return newStmt;
}
```

`backpatch()`

This will backpatch a list of labels with a label. The `nextList` is updated by `nextInstr()` which already increments the count. This will point directly to the required instruction in quadArray.

```
void backpatch(int* list, int label){
    char str[50];
    sprintf(str, "%d", label);
    if (str==NULL){
        str[0] = '0';
    }
    // list contains the list of labels to be backpatched
    int i=0;
    while(list[i] != -1){
        // go to list[i] of quadArray and update the result
        // quadArray is a linked list of quad arrays
        qArray* curr = quadArray;
        while(curr->count != list[i]){
            curr = curr->nextQuad;
        }
        // update the result
        curr->arr->result = strdup(str);
        i=i+1;
    }
    return;
}
```

makelist()

makes a dynamic array with first entry set to the incoming lable. The end is represented by -1.

```
// makelist
int* makelist(int label){
    int* list = (int*)malloc(2*sizeof(int));
    list[0] = label;
    list[1] = -1;
    return list;
}
```

merge()

This fucntion will take two dynamic arrays and output the merged array. We will also check for duplicate entries in between.

```
int* merge(int* list1, int* list2){
    // get lengths of both lists
    int len1 = 0;
    while(list1[len1] != -1){
        len1++;
    }
    int len2 = 0;
    while(list2[len2] != -1){
        len2++;
    }
    // create a new list
    int* newMerged = (int*)calloc((len1+len2+1), sizeof(int));
    int i=0;
    while(list1[i] != -1){
        newMerged[i] = list1[i];
        i++;
    }
    int j=0;
    while(list2[j] != -1){
        // Check if list2[j] is already in newMerged
        int k;
        for (k = 0; k < i; ++k) {
            if (newMerged[k] == list2[j]) {
                break;
            }
        }
        // If list2[j] is not already in newMerged, add it
        if (k == i) {
            newMerged[i] = list2[j];
            i++;
        }
```

```
        j++;
    }
    newMerged[i] = -1;
    return newMerged;
}
```

bool2int()

Since we need to evaluate the equality expressions and we cannot use the `boolean` directly. We are using the function 'bool2int' to convert a boolean expression to an integer. The function first checks if the expression is a boolean. If it's not, the function simply returns the expression as it is. Then, we generate a new temporary variable: If the expression is a boolean, a new temporary variable of integer type is generated to hold the result of the conversion. The true list of the expression is backpatched with the next instruction. This means that all the true results of the expression will now point to the next instruction. Now, we will emit the quad. It is done with the operation $'OP\_ASSIGN'$, assigning the value "true" to the temporary variable. Then we are jumping to the end of the false list. A jump instruction is emitted to skip the false part of the expression if the expression is true.

Similar to the true list, the false list of the expression is backpatched with the next instruction. Then a quad is emitted with the operation $'OP\_ASSIGN'$, assigning the value "false" to the temporary variable. Finally, the function returns the expression, which now has an integer value instead of a boolean value.

```
expression* bool2int(expression* expr){
    if(expr->isBool){
        // generate a new temp
        expr->loc = gentemp(create_symboltype(TYPE_INT, 1, NULL), NULL);
        // backpatch the true list with next instruction
        backpatch(expr->trueList, nextInstr());
        // emit the quad for true
        emit(OP_ASSIGN, "true", NULL, expr->loc->name);
        // goto the end of the false list
        char str[100];
        int pNext = nextInstr()+1;
        sprintf(str, "%d", pNext);
        emit(OP_GOTO, NULL, NULL, str);
        // backpatch the false list with next instruction
        backpatch(expr->falseList, nextInstr());
        // emit the quad for false
        emit(OP_ASSIGN, "false", NULL, expr->loc->name);
    }
    return expr;
}
```

int2bool()

The Loop, logical (AND, OR) and conditional statements, need the expressions in boolean to evaulate the results. The function 'int2bool' is used to convert an integer expression to a boolean. Just like previously explained, The function first checks if the expression is not a boolean. If it's a boolean, the function simply returns the expression as it is. If the expression is an integer, a false list is created with the next instruction. This list will hold the places in the code where the expression evaluates to false. A quad is emitted with the operation $'OP\_EQUALS'$, comparing the value of the expression with "0". If the expression is equal to 0, it is considered false. A true list is created with the next instruction. This list will hold the places in the code where the expression evaluates to true. A jump instruction is emitted to skip to the next part of the code. Finally, the function returns the expression, which now has a boolean value instead of an integer value.

```
expression* int2bool(expression* expr){
    if(expr->isBool == false){
        expr->falseList = makelist(nextInstr());
        // emit == 0
        emit(OP_EQUALS, expr->loc->name, "0", NULL);
        // print_quadArray(quadArray);
        expr->trueList = makelist(nextInstr());
        // emit goto
        emit(OP_GOTO, NULL, NULL, NULL);
    }
    return expr;
}
```

VARIABLE STACK Aux Function

These contains the basic functions necessary for initialization, push, and pop on stack. ANd the initialization of LinkedList with insert (at end) and delete functions.

(Decided not to include in PDF due to its trivial nature)

SYMBOL TABLE FUNCTIONS

lookup()

This function is one of the important function of entire translator. It populates the symboltable by generating new entires and storing them as a address reference to current symbol table. We first lookup the current symbol table. If we do not find the entries we lookup the parent (GLOBAL). If parent lookup fails then we can generate a new synmbol and insert it in current Symbol Table.

```
symboltableentry *lookup(symboltable* currST, char* yytext){
    for(int i=0; i <currST->count; i++){ // for all the entries in the
    ↪  symbol table, check if the name matches
```

```c
        if(strcmp((currST->table_entries[i])->name, yytext) == 0){
            return (currST->table_entries[i]); // return the entry if found
        }
    }
    // check if the entry is in the parent symbol table
    if(currST->parent != NULL){
        symboltableentry* parentEntry = parentLookup(currST->parent, yytext);
        if(parentEntry){
            return parentEntry;
        }
    }
    // Create a new entry if not found
    symboltableentry* entry = (symboltableentry*)malloc(sizeof(symboltableentry));
    entry->name = strdup(yytext);
    entry->type = NULL;
    entry->initial_value = NULL;
    entry->size = 0;
    entry->offset = 0;
    entry->next = NULL;
    (currST->parent)?(entry->category = TYPE_LOCAL):(entry->category = TYPE_GLOBAL); // if parent is not null, then it is local, else global
    // insert the entry in the symbol table
    currST->table_entries = (symboltableentry**)realloc(currST->table_entries, sizeof(symboltableentry)*(currST->count+1));
    currST->table_entries[currST->count] = entry;
    currST->count++;
    return entry;
}
```

create_symboltable()

This is a function create a new symbol table with ONLY name set.

create_symboltype()

This is a function create a new symbol type with type. width by default is 1. Pointer is NULL by default. These functions are trivail and not included.

update_type()

```c
void update_type(symboltableentry* entry, symboltype* type){
    // printf("Pointer to entry: %p\n", entry);
    // printf("Pointer to type: %p\n", type);
```

```
    entry->type = type;
    entry->size = get_size(type);
    // printf("Updated type of %s to %s\n", entry->name, printType(type));
    return;
}
```

gentemp()

This function generates a temnporary variable for intermediate code generation –the lookup function generates and stores the entry in currentST.

```
symboltableentry* gentemp(symboltype* type, char* initial_value) {
    char tempName[20];
    sprintf(tempName, "t%d", tempCount++);
    // Lookup or create a new entry for the temporary variable
    symboltableentry* tempEntry = lookup(currST, tempName);
    // Update type and initial value
    update_type(tempEntry, type);
    (initial_value==NULL)?(tempEntry->initial_value = NULL):(tempEntry->initial_value = strdup(initial_value));
    tempEntry->category = TYPE_TEMP;
    return tempEntry;
}
```

genparam()

This is a similar function as gentemp but generates a parameter type.

push_args()

This is a function to add a new argument to the argument list of the function. (Linked List end addition)

```
void push_args(symboltable* currST, symboltableentry* arg){
    if(currST->_argList == NULL){
        currST->_argList = (symboltableentry**)malloc(sizeof(symboltableentry*));
        currST->_argList[0] = arg;
        return;
    }
    int count = 0;
    while(currST->_argList[count] != NULL){
        count++;
    }
    currST->_argList = (symboltableentry**)realloc(currST->_argList, sizeof(symboltableentry*)*(count+1));
    currST->_argList[count] = arg;
```

```
    return;
}
```

update_return_ST(), update_ST(), print_ST()

Functions to Update count of return label and add a new entry to the symbol table. The `print_ST()` prints the symbol table.

QUADS and TAC

Trivial functions

Functions like `print_quad`, `print_quadArray` and `quadArray_initialize` are trivial since they are only used to print quads and quad arrays, or just initialize this.

emit()

This function does the emits, which basically takes in the following arguments:

```
enum op_code op,
char* arg1,
char* arg2,
char* result
```

These inputs are the parts of each member of the Quad Array (`struct qArray`, whose pointer is added at the end of the list after every emit.
The emit function adds a new operation (quad) to a list (quadArray). If the list is empty, it creates the first quad. If not, it adds the new quad to the end. Each quad has an operation and up to three arguments. The function duplicates the arguments to ensure they remain valid even if they change elsewhere in the code.

```
// Emit a quad -- add to quadArray
void emit(enum op_code op, char* arg1, char* arg2, char* result){
    // initial case -- nextQuad is NULL
    if(quadArray->arr == NULL){
        quadArray->arr = (quad*)malloc(sizeof(quad));
        quadArray->arr->op = op;
        (arg1 == NULL)?(quadArray->arr->arg1 = NULL):(quadArray->arr->arg1 = strdup(arg1));
        (arg2 == NULL)?(quadArray->arr->arg2 = NULL):(quadArray->arr->arg2 = strdup(arg2));
        (result == NULL)?(quadArray->arr->result = NULL):(quadArray->arr->result = strdup(result));
        quadArray->nextQuad = NULL;
        return;
    }
    // if the quadArray is not empty, then add the quad to the end of the linked list
```

```
    qArray* curr = quadArray;
    while(curr->nextQuad != NULL){
        curr = curr->nextQuad;
    }
    curr->nextQuad = (qArray*)malloc(sizeof(qArray));
    curr->nextQuad->arr = (quad*)malloc(sizeof(quad));
    curr->nextQuad->arr->op = op;
    (arg1 == NULL)?(curr->nextQuad->arr->arg1 =
    ↪  NULL):(curr->nextQuad->arr->arg1 = strdup(arg1));
    (arg2 == NULL)?(curr->nextQuad->arr->arg2 =
    ↪  NULL):(curr->nextQuad->arr->arg2 = strdup(arg2));
    (result == NULL)?(curr->nextQuad->arr->result =
    ↪  NULL):(curr->nextQuad->arr->result = strdup(result));
    curr->nextQuad->count = curr->count + 1;
    curr->nextQuad->nextQuad = NULL;
    return;
}
```

nextInstr()

This function just fetches the next instruction number. The quads start from 1 and the next instruction number is the count of the latest quad inserted in the array + 1.

```
int nextInstr(){
    qArray* curr = quadArray;
    while(curr->nextQuad != NULL){
        curr = curr->nextQuad;
    }
    return curr->count+1;
}
```

Grammar Rules

About

About the parser: We have used the following declarations and data types that we will be used by the outputs of the grammar rules. We include the following declarations:

```
/* Declarations */
%{
    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>
    #include "3_A4_translator.h"
    extern int yylex();      // Lexical Analyzer generated by Flex
    void yyerror(char *s);   // Error function for Bison
    extern char* yytext;     // yytext declaration
```

```
    extern int yylineno;    // yylineno declaration
    extern var_type_stack var_type;                  // push on
    ↪  type_specifier, pop on IDENTIFIER
    extern string_list* string_head;
    char* function_name;
    int next_instr_addr = 0;
%}
```

The data types we are using in this parser file for the grammar rules are listed below. We have also mentioned which rules, as well as tokens, are of the data below mentioned data types:

```
%union {
    char* intval;
    char* strval;
    char* charval;
    char* u_op;
    int count;
    int instr_no;
    char UNIARY_OPERATOR;
    struct symboltableentry* sym_entry;
    struct symboltype* sym_type;
    struct expression* expr;
    struct statement* stmt;
};

%token <sym_entry> IDENTIFIER
%token <intval> INTEGER_CONSTANT
%token <charval> CHARACTER_CONSTANT
%token <strval> STRING_LITERAL

%type <expr> primary_expression expression postfix_expression
↪  unary_expression multiplicative_expression
%type <expr> additive_expression relational_expression constant
↪  equality_expression logical_and_expression
%type <expr> logical_or_expression conditional_expression
↪  assignment_expression expression_statement expression_opt
%type <stmt> statement compound_statement selection_statement
↪  iteration_statement jump_statement
%type <stmt> block_item block_item_list block_item_list_opt M
%type <sym_entry> initializer direct_declarator init_declarator
↪  declarator identifier_opt func_ID
%type <u_op> pointer_opt pointer unary_operator
%type <count> argument_expression_list_opt argument_expression_list
%type <instr_no> N
```

Primary Expressions

```
primary_expression : IDENTIFIER {
                            // new expression
                            $$ = create_expression();
                            $$->loc = $1;
                            printf("primary-expression\n");
                        }
                    | constant {
                            printf("primary-expression\n");
                            $$ = $1;
                        }
                    | STRING_LITERAL {
                            $$ = create_expression();
                            $$->loc = gentemp(create_symboltype(TYPE_PTR, 1, NULL), $1);
                            $$->loc->type->ptr = create_symboltype(TYPE_CHAR, 1, NULL);
                            ll_insert(string_head, $1);
                            emit(OP_ASSIGN_STR, $1, NULL, $$->loc->name);
                            printf("primary-expression\n");
                        }
                    | L_PARENTHESIS expression R_PARENTHESIS {
                            $$ = $2;
                            printf("primary-expression\n");
                        }
                    ;
```

For IDENTIFIER, using the `create_expression()` function, we initialize the Expression structure, and assign it to $$, and set the symbol $1 for its `loc`.

For a string literal, an expression is assigned as above to $$, but the symbol which is set at its `loc` is a temporary variable, which is created using the `gentemp()` function, which is used to store string literal entry temporarily. We futher initialize the temporary variable's symbol struct by assigning its type and category, then storing the string in a linked lists of strings. After this, we emit such that the string literal is assigned to the temporary variable in the generated TAC.

The parenthesis just keep the expressions as it is.

For constant, We expand this rule to:

```
constant : INTEGER_CONSTANT {
                $$ = create_expression();
                $$->loc = gentemp(create_symboltype(TYPE_INT, 1, NULL), $1);
                emit(OP_ASSIGN, $1, NULL, $$->loc->name);
```

```

            }
        | CHARACTER_CONSTANT{
                $$ = create_expression();
                $$->loc = gentemp(create_symboltype(TYPE_CHAR, 1, NULL),
                 ↪  $1);
                emit(OP_ASSIGN, $1, NULL, $$->loc->name);
            }
        ;
```

For both these constants, we initialize an expression and create a temporary variable, after which they are emitted to and assign the values to it.

Note

Since there are many grammar rules to explain, we do not want to populate the PDF with unnecessary code, and redundant explanation. We have written comments in our .y file, which explain the function of their respective rules.

We have explained the data structures we have used for this assignment, the architecture and the supporting functions to create the Three-Address-Code.

As mentioned above, we have used the following sources for reference:

1. Fronsto (GitHub)
2. RKJenamani (GitHub)
3. Brainstorming with Group 26.