NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

TDT4205 COMPILER CONSTRUCTION, SPRING 2024

Problem Set 4

Omdal, Tobias Stenersen

September 27, 2024

1 Introduction

This document contains the answer to Part 1 of Problem Set 4, the Three-Address Code translation. It also contains screenshots, comments and some insights of the code in Part 2, which is also documented in the code-base itself. Therefore, Section 3 is just added as a bonus, but does not contain anything essential to the delivery.

2 Three-Address Code (TAC)

To turn the VSL interpretation of a Fibonacci-calculator into TAC, there are only a couple of things we need to change. The naming conventions follow the lectures and the dragon book. We start by declaring all the variables by themselves like the following example from the *Compilers Principles, Techniques & Tools* book[1]:

$$x = y \text{ op } z$$

Figure 1: Three-Address Code Format

Applying it to the initialization of our variables in main we get the following:

t1 = 5
t2 = t1 + 1
t3 = 0
t4 = 1
t5 = 0
t6 = 1

The highest number of addresses we have here is t2 with its 3: t2 = t1 op 1. Next, since we don't have access to any conventional keywords including 'while', we have deconstruct it into goto statements. We first define the entry-point of the loop, meaning the conditional.

if t6 <t2 goto L2

Conditional jump:	if x goto L	← go to L if x is true
	ifFalse x goto L	← go to L if x is false
	if x < y goto L	← comparison operators
	if x >= y goto L	
	if x != y goto L	

Figure 2: Conditional Jumps

This is valid TAC syntax, as can be seen from the screenshot of the lecture foils *(Lecture 17, Compiler Construction, Spring 2024)* in Figure 2. In our case we are interested in case 1: if x goto L. Combining it with a label L1, allows us to do a jump back to the conditional after every iteration of the loop. In addition to the L1 label we use another one to skip over the loop in-case the conditional fails. We then have:

L1:
    if t6 <t2 goto L2
    goto L3

Next, we need to define the body of the loop, done below the L2 label. Everything inside the while statement is already TAC valid. This means we just have to change the variable-names and go back to the conditional after an iteration by using another goto statement.

L2:
    t5 = t3 + t4
    t3 = t4
    t4 = t5
    t6 = t6 + 1
    goto L1

Finally, after calculating the nth Fibonacci number in the loop, we have to print the number and exit the program. To do this, we have to break the print function down into multiple steps. First of all, we need to define the strings that will be passed. We define them in t7 and t8. Then we have to load the params using param t7; param t8 etc. Finally, we call the print function with the 4 parameters by passing them in the call statement. After calling we can then just return.

L3:
    t7 = ”The ”
    t8 = ”th number in the sequence is: ”
    param t7
    param t1
    param t8
    param t5
    call print, 4
    return 0

Now, before looking at the full program put together, we have to define our assumptions.
Assumptions:

- Assumes the use of begin/end to define the start and end of the control flow, even if they aren't strictly necessary in TAC, they give a nice overview.
- Assumes the need to define and call the program entry-point (like main in c), in this case also main, even though it might be predefined and unnecessary.
- Assumes functions are defined using func identifier() instead of just having a label for ”main:”.
- Assumes the print function defined, as mentioned in the task description for brevity.
- Assumes no need for loading main's parameters using param, since the Fibonacci number is hard-coded.

call main
goto program_end

```
func main()
begin
        t1 = 5
        t2 = t1 + 1
        t3 = 0
        t4 = 1
        t5 = 0
        t6 = 1

    L1:
        if t6 <t2 goto L2
        goto L3

    L2:
        t5 = t3 + t4
        t3 = t4
        t4 = t5
        t6 = t6 + 1
        goto L1

    L3:
        t7 = "The "
        t8 = "th number in the sequence is: "
        param t7
        param t1
        param t8
        param t5
        call print, 4
        return 0

end
program_end:
```

This covers the requirements defined by the book; it should have three operands, each three-address assignment instruction has at most 1 operator on the right side, there must be a temporary value to hold every value computed, and we can also have less than three operands[1].

3 Symbol Table Creation

This section only contains screenshot of the code base and some explanations. Some things may be different from the screenshots to the code, but not anything major.

3.1 Table Building Entry Point

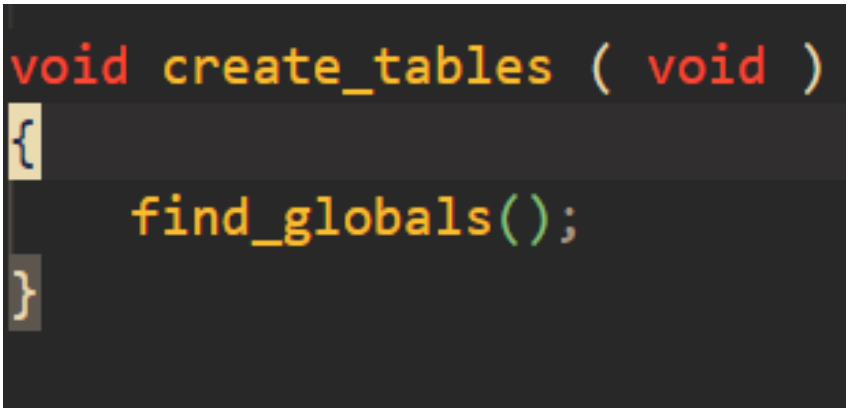

Figure 3: Table Creation Entry Point

We start by defining the entry point of the table builder, as shown in Figure 3. The find_globals() function will traverse every child of root and define behavior for each of them.

3.2 Finding Globals

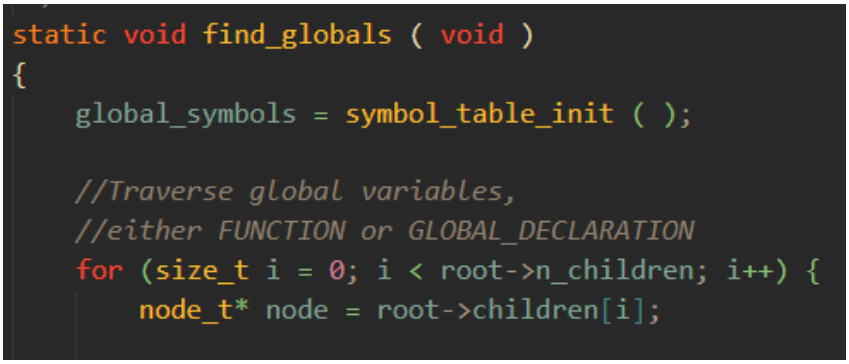

Figure 4: Function find_globals()

The find_globals() function will first define the *Global Symbol Table* and iterate over the children of root.

```c
if (node->type == FUNCTION) {
    //Create new scope/local table for function
    symbol_table_t* local_symbols = symbol_table_init();
    //Parameters are always second child of function,
    //func identifier is always first and block/statement is third.
    node_t* first_node = node->children[1];

    //If we have parameters we call helper function
    if (first_node->type == LIST && first_node->n_children != 0) {
        handle_function_parameters(first_node, local_symbols);
    }

    //Call recurive bind_names() function on the function block.
    bind_names(local_symbols, node->children[2]);

    //Create new symbol for function identifier,
    //insert it into the global symbol table.
    symbol_t* symbol = create_symbol(node->children[0], SYMBOL_FUNCTION, global_symbols, local_symbols);
    symbol_table_insert(global_symbols, symbol);
```

Figure 5: FUNCTION Type Handling

When iterating over the children we will compare their types to the global scope functions / variables. If we find a function, we define a new scope, handle all its parameters, recursively handle the function body (explained later), and then add the function identifier as a symbol to the global symbol table.

```c
} else if (node->type == GLOBAL_DECLARATION) {
    /*
        global variables is always under:
        GLOBAL_DECLARATION > LIST > IDENTIFIER_DATA,
        so we have to tranverse into list before adding variables
    */
    node = node->children[0];

    //If it somehow isn't LIST, we skip the node
    if (node != NULL && node->type != LIST) break;

    //We then traverse the list of variables
    for (size_t j = 0; j < node->n_children; j++) {
        node_t *child = node->children[j];


        //Global vars can be variables or an array, var as default.
        int type = SYMBOL_GLOBAL_VAR;

        //If an array we traverse into it and change type
        if (child->type == ARRAY_INDEXING) {
            child = child->children[0];
            type = SYMBOL_GLOBAL_ARRAY;
        }

        if (child->type != IDENTIFIER_DATA) continue;

        //create symbol using helper function
        symbol_t* symbol = create_symbol(child, type, global_symbols, NULL);
        //Insert symbol into global symbol table
        symbol_table_insert(global_symbols, symbol);
    }
}
```

Figure 6: GLOBAL_DECLARATION Handling

If we however encounter a global variable, we only need to check whether its a variable or an array, and likewise, add it to the global symbol table. Note that the handle_function_parameters(node_t* node, symbol_table_t* local_symbols) and create_symbols(node_t* node, const int type, symbol_table_t* symbol_table, symbol_table_t* symbol_table_pointer) are helper functions.

```c
static void handle_function_parameters(node_t* node, symbol_table_t* local_symbols) {
    //Iterate the parameter list
    for (size_t i = 0; i < node->n_children; i++) {
        node_t* local_node = node->children[i];
        //Create new symbol for every param and insert into symbol table.
        symbol_t* symbol = create_symbol(local_node, SYMBOL_PARAMETER, local_symbols, local_symbols);
        symbol_table_insert(local_symbols, symbol);
    }
}
```

Figure 7: Function Parameters

The handle_function_parameters(node_t* node, symbol_table_t* local_symbols) function just iterates its children, creates a symbol and adds it to the symbol table. This function, unlike create_symbols(node_t* node, const int type, symbol_table_t* symbol_table, symbol_table_t* symbol_table_pointer) won't be used any further, but was defined for readability. The parameters are also defined in find_globals() to make it easier to handle the recursion cases in bind_names(symbol_table_t *local_symbols, node_t *node).

```c
static symbol_t* create_symbol(node_t* node, const int type, symbol_table_t* symbol_table, symbol_table_t* symbol_table_pointer) {
    //Allocate memory for new symbol.
    symbol_t *symbol = malloc(sizeof(symbol_t));
    if (symbol != NULL) {
        //Set all symbol attributes.
        symbol->name = node->data;
        symbol->type = type;
        symbol->node = node;
        //Sequence number is different for global/local tables,
        //so they are taken in as arguments
        symbol->sequence_number = symbol_table->n_symbols;
        /*
            GLOBALS/ARRAYS -> NULL
            FUNCTIONS -> Own symbol table
            PARAMS/LOCALS -> Symbol table they belong to
        */
        symbol->function_symtable = symbol_table_pointer;
    }
    return symbol;
}
```

Figure 8: Function create_symbol()

The create_symbols(node_t* node, const int type, symbol_table_t* symbol_table, symbol_table_t* symbol_table_pointer) function was used a lot, and was therefore assigned its own function. It mainly allocates memory for a symbol, defined the symbol's attributes from the node, and handles the varying symbol_table_t cases, as can be seen in Figure 8.

3.3 Finding Local Symbols and Strings

```
static void bind_names ( symbol_table_t *local_symbols, node_t *node )
{
    /*
        switch(type)
            case STRING
            case BLOCK
            case EXRPESSION
            case RELATION
            case RETURN
            case ASSIGNMENT
            case PRINT
            case IDENTIFIER
            default
    */
```

Figure 9: Recursive Function bind_names()

bind_names(symbol_table_t *local_symbols, node_t *node) is a recursive function that handles the local symbol table, references etc inside of the function body. As can be seen from Figure 9 it handles the passed nodes using a switch statement that checks the types.

```
switch(node->type) {
    case STRING_DATA: {
        //Calls string helper function
        append_string(node);
        break;
    }
```

Figure 10: STRING_DATA Type Handling

If the type is a STRING_DATA it calls the helper function append_string(node_t* node) that just calls add_string(node->data), changes the type to STRING_LIST_REFERENCE and changes the node->data.

```
case BLOCK: {
    //Creates new Hashmap, binds it to local symbol table,
    //sets backup hashmap to old hashmap
    symbol_hashmap_t* new_hashmap = symbol_hashmap_init();
    symbol_hashmap_t* current_hashmap = local_symbols->hashmap;
    local_symbols->hashmap = new_hashmap;
    new_hashmap->backup = current_hashmap;

    //Recursively call function for all children of node
    for (size_t i = 0; i < node->n_children; i++) {
        bind_names(local_symbols, node->children[i]);
    }

    //CLEANUP -> skip current hashmap and destroy it
    local_symbols->hashmap = current_hashmap;
    symbol_hashmap_destroy(new_hashmap);
    break;
}
```

Figure 11: BLOCK Type Handling

If the type is a block, we need to create a new scope, by initializing a hashmap and inserting it between the original hashmap and the symbol table and then linking it to the original hashmap. We then recursively call bind_names() for the rest of the BLOCK body and finally skip over the new hashmap and destroy it after the body is finished running.

```
case PRINT_STATEMENT:
case RETURN_STATEMENT:
case EXPRESSION:
case RELATION:
case ASSIGNMENT_STATEMENT: {
    node_t* child = node->children[0];

    //If the child is not LIST, for example when returning a var, we bind the reference,
    //and bind_symbol checks if type(child) is an IDENTIFIER to be referenced.
    bind_symbol(child, local_symbols);

    //If we have a structure like PRINT > LIST > IDENTIFIER or
    //PRINT > LIST > STRING_DATA etc
    for (size_t i = 0; i < child->n_children; i++) {
        bind_symbol(child->children[i], local_symbols);
        if(child->children[i]->type == STRING_DATA) {
            append_string(child->children[i]);
        }
    }

    //If we have something like ASSIGNMENT > IDENTIFIER1, IDENTIFIER2 or
    //EXPRESSION > IDENTIFIER1, IDENTIFIER2 etc.
    if (node->n_children > 1) {
        node_t* second_child = node->children[1];
        bind_symbol(second_child, local_symbols);
        for (size_t i = 0; i < second_child->n_children; i++) {
            bind_symbol(second_child->children[i], local_symbols);
        }
    }
    break;
}
```

Figure 12: Multi-identifier Statements

Because of the similarity in all the statements in Figure 12, we combine them all into one. The fall-through in this case is intentional, although its not common to use case fall-through over something like an if statement. Some of the statements have a somewhat different structure than the others, so we need special cases to iterate over the children etc. The reason we combine them all like this is because of the symbol referencing, done through bind_symbol(). We for example use pre-existing variables when using an ASSIGNMENT_STATEMENT, meaning we don't want to create a new symbol, but instead reference the already existing one.

```
case IDENTIFIER_DATA: {
    //Create symbol using helper function and insert into symbol table.
    symbol_t* symbol = create_symbol(node, SYMBOL_LOCAL_VAR, local_symbols, local_symbols);
    symbol_table_insert(local_symbols, symbol);
    break;
}
default: {
    //If node is none of the nodes above, for example LIST,
    //we recursively call its children
    for (size_t i = 0; i < node->n_children; i++) {
        bind_names(local_symbols, node->children[i]);
    }
    break;
}
```

Figure 13: Identifier and default Type Handling

Next, if the node is an IDENTIFIER_DATA we create a symbol from the node and insert it into the local symbol table. Finally, if the statement is not defined above, we recurse its children, which is the case for nodes like LIST.

3.4 Printing and Cleanup

```
/* Adds the given string to the global string list, resizing if needed.
 * Takes ownership of the string, and returns its position in the string list.
 */
static size_t add_string ( char *string )
{
    // TODO: Write a helper function you can use during bind_names(),
    // to easily add a string into the dynamically growing string_list
    if (string_list_len >= string_list_capacity) {
        if (string_list_capacity == 0) {
            string_list_capacity = 8;
        } else {
            string_list_capacity *= 2;
        }

        char **new_string_list = realloc(string_list, string_list_capacity * sizeof(char*));
        if (new_string_list != NULL)
            string_list = new_string_list;
    }
    string_list[string_list_len++] = string;
    return string_list_len-1;
}
```

Figure 14: Adding String to List of Strings

Whenever we encounter a string while traversing the nodes, we add it to the list of strings, like mentioned earlier. The function just sets the capacity to 8 if its 0 and doubles otherwise. It

then reallocates the list of strings based on the capacity, sets the list of strings to the reallocated one, inputs the string and increments the length of the list.

```c
/* Prints all strings added to the global string list */
static void print_string_list ( void )
{
    for (int i = 0; i < string_list_len; i++) {
        printf("%d: %s\n", i, string_list[i]);
    }
}
```

Figure 15: Print Function for List of Strings

```c
/* Frees all strings in the global string list, and the string list itself */
static void destroy_string_list ( void )
{
    for (size_t i = 0; i < string_list_len; i++) {
        free(string_list[i]);
    }
    free(string_list);
}
```

Figure 16: String List Cleanup

The above two functions in Figure 15 and Figure 16 are just called at the end of the program. print_list_string() just iterates the string list and prints its index and value. destroy_string_list() iterates all the strings, frees them and then frees the list itself.

```c
static void print_symbol_table ( symbol_table_t *table, int nesting )
{
    //Tab/Spacing amount
    const int spacing = 4 * nesting;

    char *spaces = (char*)malloc(spacing+1);//Allocates space for spacing and + 1 for null terminal
    if (spaces == NULL) fprintf(stderr, "Failed to allocate spaces\n");

    for (int i = 0; i < spacing; i++)
        spaces[i] = ' ';
    spaces[spacing] = '\0'; //string finished building

    for (size_t i = 0; i < table->n_symbols; i++) {
        symbol_t* symbol = table->symbols[i];
        printf("%s%d: %s(%s)\n",
            spaces,
            (int)symbol->sequence_number,
            SYMBOL_TYPE_NAMES[symbol->type],
            symbol->name);
        //Recursively call print_symbol_table for functions with 1 more level of indentation
        if (symbol->function_symtable != NULL && symbol->function_symtable != table) {
            print_symbol_table(symbol->function_symtable, nesting+1);
        }
    }
    free(spaces);
}
```

Figure 17: Print Function for Symbol Tables

The print_symbol_table() function, similarly to print_list_string(), iterates over the symbols and prints its values in a specific format. The statements before the loop handle the spacing that occurs upon function body indentation. It just calculates how much spacing is needed, which is 1tab/4spaces * the nesting level and then allocates the memory for the string, since its dynamic. Then we just iteratively append the spaces. Finally we check if there are any functions inside the symbol table which recurses the function for that local symbol table with an incremented level of nesting.

```c
/* Frees up the memory used by the global symbol table, all local symbol tables, and their symbols */
static void destroy_symbol_tables ( void )
{
    //Calls helper function that allows recursion.
    destroy_sym_tables(global_symbols);
}
```

Figure 18: Cleanup Function That Calls Helper Function

The destroy_symbol_tables() function just calls the helper function, to allow for recursion.

```c
static void destroy_sym_tables(symbol_table_t* sym_t) {
    for (size_t i = 0; i < sym_t->n_symbols; i++) {
        symbol_t* symbol =  sym_t->symbols[i];
        if (symbol->type == SYMBOL_FUNCTION) {
            destroy_sym_tables(symbol->function_symtable);
        }
    }
    symbol_table_destroy(sym_t);
}
```

Figure 19: Cleanup Helper Function

We iterate over the symbols, check if its a function, recurse the function on their own local tables, and then destroy all the tables bottom-up. Unlikely for functions to be defined inside other functions, but this handles it anyways just in-case.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques & Tools*. Pearson Education Limited, 2007.