NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

TDT4205 COMPILER CONSTRUCTION, SPRING 2024

# Problem Set 6

Omdal, Tobias Stenersen

September 27, 2024

## Introduction

This document contains the answers to Part [1] of problem set 6. It will contain thorough explanations of how I got to my answers, and also serves as an aid in learning from the problem set. The section on Control flow graphs (1.1) starts by breaking the problem into smaller chunks, before explaining why I am doing what I am doing and finally demonstrating the answer to the question. The section on Reaching definitions (1.2) follows the same structure. Lastly, I have also provided screenshots from the code-base, but there are no explanations, as I have already explained in detail using comments. If you are not interested in seeing my thought process, the answer to question 1 will be in figure [5], and the answer to question can be found at the end of section 1.2(2).

All the answers will be based on either *Cherubin, Stefano's* presentations on Control flow Graphs and data-flow analysis, *Gaustad, Anders K.'s* lecture foils from the PS6 recitation lecture or the courses book of choice: *Compilers Principles, Techniques & Tools* book[1].

## 1 Data flow analysis (40%)

```
func fibonacci()
begin
    var n, first, second, next, i
    n := 10
    first := 0
    second := 1
    i := 0
    while i < n do
    begin
        if i <= 1 then
            next := i
        else
        begin
            next := first + second
            first := second
            second := next
        end
        print next
        i := i + 1
    end
    return 0
end
```

Figure 1: Fibonacci Sequence Function

### 1.1 Control flow graph (15%)

**Q1:** Draw a corresponding control flow graph, and label the program points.

**A:** To draw the *Control Flow Graph* (CFG) of the program above, we will follow a couple of steps:

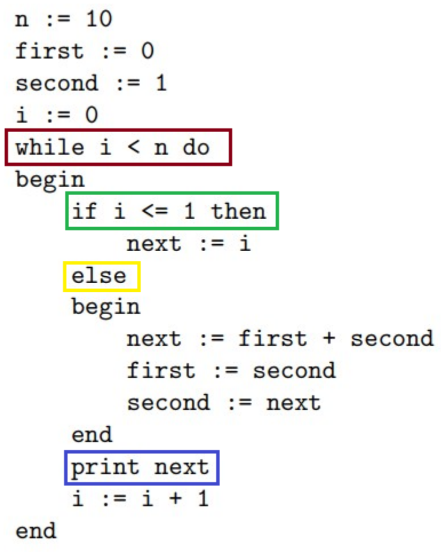
- Remove data about the function, as we are only interested in the *contents*. we will make the assumption that there is no in or out blocks outside of the while loop itself. As explained in *Gaustad, Anders K.*'s recitation lecture, we will remove everything from the function definition, to the variable declarations and return statement:

```
n := 10
first := 0
second := 1
i := 0
while i < n do
begin
    if i <= 1 then
        next := i
    else
    begin
        next := first + second
        first := second
        second := next
    end
    print next
    i := i + 1
end
```

Figure 2: Fibonacci Function Without Function Data

- Next, we need to identify statements that can be grouped within the same *basic block*. This includes looking at **labels** and **jumps**, as they are what breaks the sequential execution of the program. Not doing this step wouldn't exactly result in an accurate, nor interesting analysis of the control flow. Statements using labels or jumps like *while statements, if statements, else statements, and function calls* end the basic block before it and start a new one. The following list will refer to Figure (3) and discuss why it will break the sequential flow of the previous basic block.
  - While loop
    - A while statement includes both a label and a jump. The label is used for jumping back after an iteration, and the jump is used for exiting if the condition fails.
  - If statement
    - If statements consist of a jump. In an if-then statement we for example continue downward if the code succeeds and jump past the code if it fails.
  - Else statement
    - Else statements have a label. The label is used for when an if statement condition fails and it needs to jump to some other piece of code; the else's label.
  - Function call

    * Important to be aware that function calls is essentially just a jump to another memory location. A call is basically just a jump with more instructions, like pushing the instruction pointer to the stack, and using the return address to get back to where we were. In other words, this part will also break the sequential execution.



```
n := 10
first := 0
second := 1
i := 0
while i < n do
begin
    if i <= 1 then
        next := i
    else
    begin
        next := first + second
        first := second
        second := next
    end
    print next
    i := i + 1
end
```

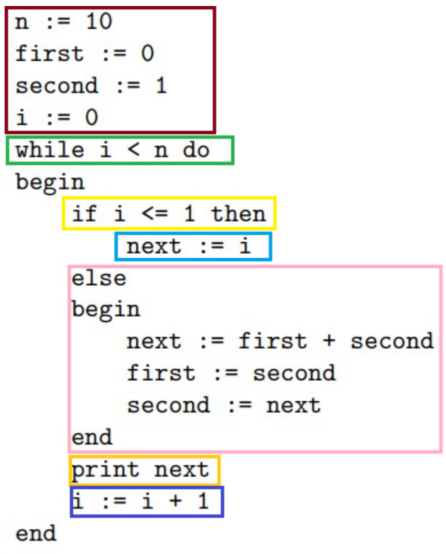
Figure 3: Highlight of statements with Jumps and Labels

- Now we need to recursively break larger statements into smaller statements. If we don't, we could for example represent the while loop as a single statement, meaning we would have two blocks, S1 and S2. Therefore, given the while label, we need to repeatedly break the statement into smaller and smaller sections.
- Basic block
  - All the statements here will just continue sequentially, therefore we have our first basic block.
- While loop header
  - This is the header of the loop. As mentioned earlier it performs a comparison and is just a label and a jump. Since it breaks the previous block with the label and creates a new one immediately with the jump, it is its own block.
- If statement
  - The if statement is just a comparison and a jump. Therefore, if the previous section wasn't a jump, it could've been apart of the earlier block. Here we will either jump to some other label, or continue execution into the next block.
- Basic block
  - Just a single, simple instruction following a label. Since it is both the start and the end of the if-then block, we will immediately perform a jump into the join section between the if and else statements.

- Else statement
  - The else block will contain a label that is jumped to from the if statement. Since the label only starts a new block, the instructions following it will also be included in the block, until the end, where we perform another jump to the same section as the if block.

- Print call
  - As mentioned previously, the call is pretty much the same as a jump, meaning it will break the control flow.

- Basic block
  - Just a single instruction at the end of the program. After the incrementation, another unconditional jump will be performed to the while statement, which decides whether another increment will be performed.



```
n := 10
first := 0
second := 1
i := 0
while i < n do
begin
    if i <= 1 then
        next := i
    else
    begin
        next := first + second
        first := second
        second := next
    end
    print next
    i := i + 1
end
```

Figure 4: Overview of the blocks in the execution

- Now we can add the different basic blocks into a graph. The nodes in the graph will represent the blocks, and the arcs will determine the relationship between the nodes. As *Cherubin* mentions, an **outgoing arc** means that the destination *may* be a successor to the basic block, and an **incoming arc** means that the source block *may* be a predecessor to the basic block.

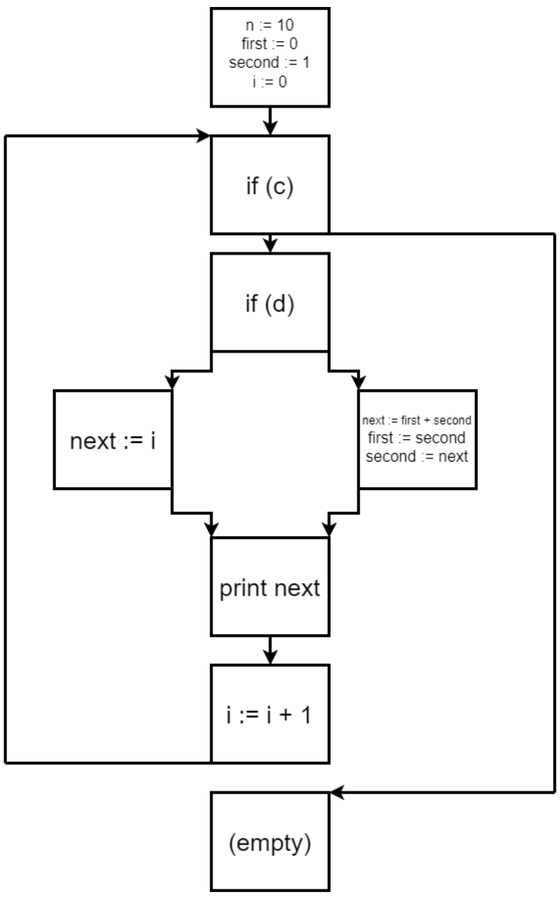


Figure 5: Control Flow Graph

1. The first block is the variable definitions.
2. The second block is the while statement header, with the conditional **c**.
3. The third block is the if statement with the conditional **d**.
4. The fourth block is the if-then block.
5. The fifth block is the if-then-else block.
6. The sixth block is the call/print subroutine statement.
7. The seventh block is the incrementation block, that is followed by the jump to the while header.
8. The eighth and final block is just a pointer to the end of the while loop. It could be any form of statement or block, like a return statement at the end of a function for example.

- A final step we could perform is **pruning**, where we would remove the empty blocks or statements. In our case, the empty block at the end could be removed for another block, but given our assumption that there are no other blocks or statements outside of the while loop, we need to have it for the while statement to be able to jump.

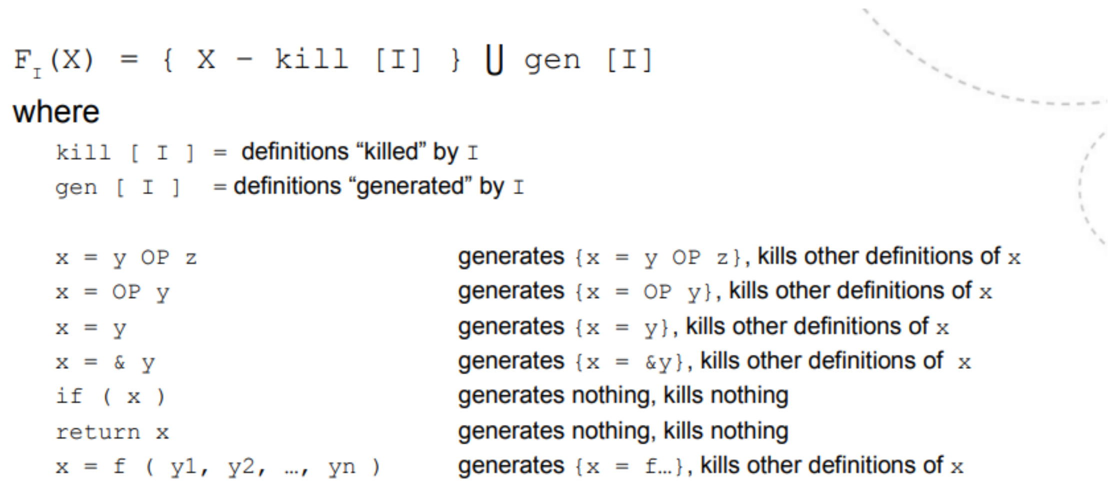
## 1.2 Reaching definitions (25%)

**Q2:** State and solve its data flow equations for reaching definitions analysis.

**A:** To perform definitions analysis on the execution above, we need to find all the program points where the assigned value *may* be observed (Cherubin, Stefano. *Data flow analysis instances*. p 20). This means we have to look at the sets of assignments. The first thing we have to do is define the forward function and the meet operation:

- Forward Analysis: $out[B] = F_B(in[B])$
- Meet Operation: $in[B] = \cup\{out[B'] | B' \in pred(B)\}$

Using these we can create relationships between the program points. As *Cherubin* displays in his presentation about transfer functions, we can use these transfer functions to find where the definitions are still available at some program point, or not.



$F_I(X) = \{ X - kill [I] \} \cup gen [I]$

where

`kill [ I ]` = definitions “killed” by `I`

`gen [ I ]` = definitions “generated” by `I`

| | |
|---|---|
| `x = y OP z` | generates `{x = y OP z}`, kills other definitions of `x` |
| `x = OP y` | generates `{x = OP y}`, kills other definitions of `x` |
| `x = y` | generates `{x = y}`, kills other definitions of `x` |
| `x = & y` | generates `{x = &y}`, kills other definitions of `x` |
| `if ( x )` | generates nothing, kills nothing |
| `return x` | generates nothing, kills nothing |
| `x = f ( y1, y2, …, yn )` | generates `{x = f…}`, kills other definitions of `x` |

Figure 6: Generating definitions

By going through all the definitions in the program and assigning them to some label, we get the following result:

- d1: n = 10
- d2: first = 0
- d3: second = 1
- d4: i = 0
- d5: next = i
- d6: next = first + second
- d7: first = second
- d8: second = next

- d9: i = i + 1

Next, we label all the program points of the Control Flow Graph, so that we can define the relationships between them:

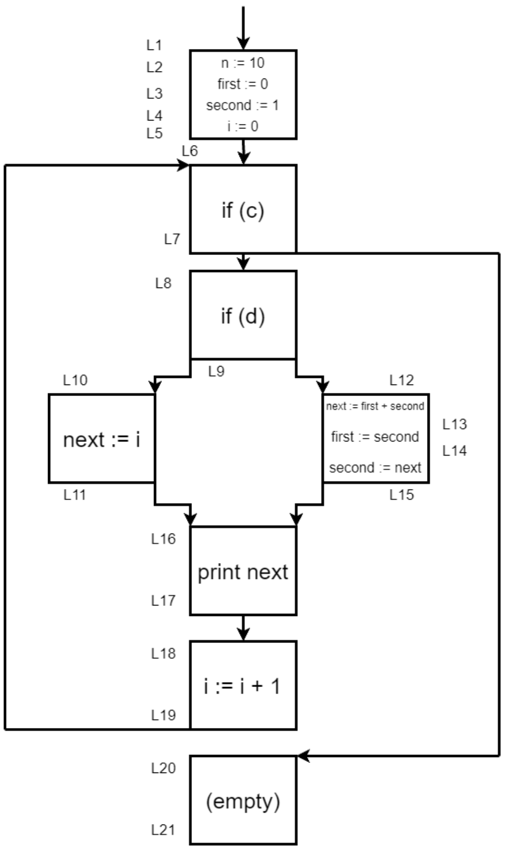


Figure 7: CFG with labels

## Data Flow Equations

L1 = {} Assumption: nothing before L1
L2 = $\{L1\} \cup d1$
L3 = $\{L2 - d7\} \cup d2$ Join definition and remove other redefinitions
L4 = $\{L3 - d8\} \cup d3$
L5 = $\{L4 - d9\} \cup d4$
L6 = $\{L5 \cup L19\} - \{L5 \cap !L19\} - >$ Join {L5, L19}, remove elements only in L5, but not in L19
L11 = $\{L10 - d6\} \cup d5$
L13 = $\{L12 - d5\} \cup d6$
L14 = $\{L13 - d2\} \cup d7$
L15 = $\{L14 - d3\} \cup d8$
L16 = $L11 \cup L15$ Join the if-then block and else block
L19 = $\{L18 - d4\} \cup d9$

## Iteration 1

L1 = {}

**Initial Definitions:**
L2 = $L1 \cup d1$ = d1
L3 = $L2 \cup d2$ = d1, d2
L4 = $L3 \cup d3$ = d1, d2, d3
L5 = $L4 \cup d4$ = d1, d2, d3, d4

**Control Flow, but no definitions:**
L6 = d1, d2, d3, d4 - L19 not in question yet, so omitted for brevity
L7 = d1, d2, d3, d4
L8 = d1, d2, d3, d4
L9 = d1, d2, d3, d4
L10 = d1, d2, d3, d4

**If-then block:**
L11 = $L10 \cup d5$ = d1, d2, d3, d4, d5

**Else block:**
L12 = d1, d2, d3, d4
L13 = $L12 \cup d6$ = d1, d2, d3, d4, d6
L14 = $L13 \cup d7$ = d1, d3, d4, d6, d7
L15 = $L14 \cup d8$ = d1, d4, d6, d7, d8

**Control flow join:**
L16 = $L11 \cup L15$ = d1, d2, d3, d4, d5, d6, d7, d8

**Print statement:**
L17 = d1, d2, d3, d4, d5, d6, d7, d8

**Increment:**
L18 = d1, d2, d3, d4, d5, d6, d7, d8
L19 = $\{L18 - d4\} \cup d9$ = d1, d2, d3, d5, d6, d7, d8, d9

**End of while statement, if condition c fails, only earlier variable definitions hold:**
L20 = d1, d2, d3, d4

## Iteration 2

L1 = {}

**Initial Definitions: (stays the same, not part of loop)**
L2 = d1
L3 = d1, d2
L4 = d1, d2, d3
L5 = d1, d2, d3, d4

**Control Flow, but no definitions:**
L6 = $\{L5 \cup L19\} - \{L5 \cap !L19\}$ = d1, d2, d3, d5, d6, d7, d8, d9 - removed d4, since only in L5, not L19
L7 = d1, d2, d3, d5, d6, d7, d8, d9
L8 = d1, d2, d3, d5, d6, d7, d8, d9
L9 = d1, d2, d3, d5, d6, d7, d8, d9
L10 = d1, d2, d3, d5, d6, d7, d8, d9

**If-then block:**
L11 = $\{L10 - d6\} \cup d5$ = d1, d2, d3, d5, d7, d8, d9

**Else block:**
L12 = d1, d2, d3, d5, d6, d7, d8, d9 - same as L9
L13 = $\{L12 - d5\} \cup d6$ = d1, d2, d3, d6, d7, d8, d9
L14 = $\{L13 - d2\} \cup d7$ = d1, d3, d6, d7, d8, d9
L15 = $\{L14 - d3\} \cup d8$ = d1, d6, d7, d8, d9
**Control flow join:**

L16 = $L11 \cup L15$ = d1, d2, d3, d5, d6, d7, d8, d9

**Print statement:**
L17 = d1, d2, d3, d5, d6, d7, d8, d9

**Increment:**
L18 = d1, d2, d3, d5, d6, d7, d8, d9
L19 = $\{L18 - d4\} \cup d9$ = d1, d2, d3, d5, d6, d7, d8, d9

**End of while statement. Since L6 changed, L20 also changes:**
L20 = d1, d2, d3, d5, d6, d7, d8, d9

## Iteration 3

L1 = {}

**Initial Definitions: (stays the same, not part of loop)**
L2 = d1
L3 = d1, d2
L4 = d1, d2, d3
L5 = d1, d2, d3, d4

**Control Flow, but no definitions:**
L6 = $\{L5 \cup L19\} - \{L5 \cap !L19\}$ = d1, d2, d3, d5, d6, d7, d8, d9
L7 = d1, d2, d3, d5, d6, d7, d8, d9
L8 = d1, d2, d3, d5, d6, d7, d8, d9
L9 = d1, d2, d3, d5, d6, d7, d8, d9
L10 = d1, d2, d3, d5, d6, d7, d8, d9

**If-then block:**
L11 = $\{L10 - d6\} \cup d5$ = d1, d2, d3, d5, d7, d8, d9

**Else block:**
L12 = d1, d2, d3, d5, d6, d7, d8, d9 - same as L9
L13 = $\{L12 - d5\} \cup d6$ = d1, d2, d3, d6, d7, d8, d9
L14 = $\{L13 - d2\} \cup d7$ = d1, d3, d6, d7, d8, d9
L15 = $\{L14 - d3\} \cup d8$ = d1, d6, d7, d8, d9
**Control flow join:**

L16 = $L11 \cup L15$ = d1, d2, d3, d5, d6, d7, d8, d9

**Print statement:**
L17 = d1, d2, d3, d5, d6, d7, d8, d9

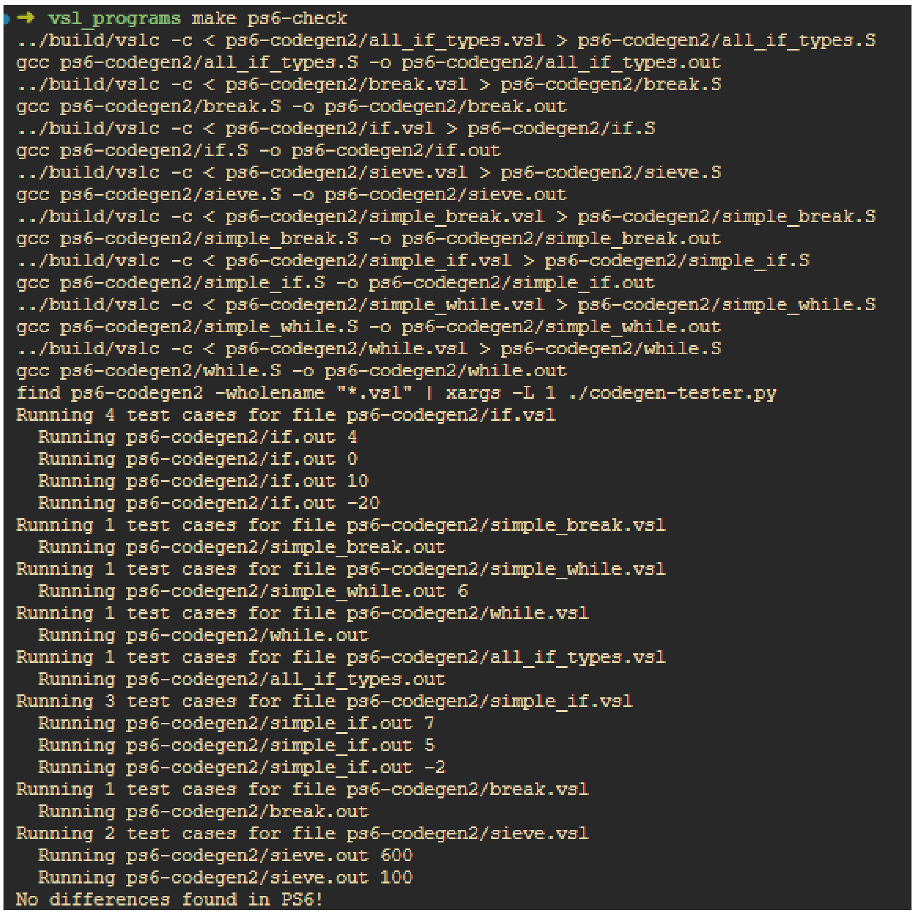
**Increment:**
L18 = d1, d2, d3, d5, d6, d7, d8, d9
L19 = $\{L18 - d4\} \cup d9$ = d1, d2, d3, d5, d6, d7, d8, d9

**End of while statement:**
L20 = d1, d2, d3, d5, d6, d7, d8, d9

From *iteration 2* to *iteration 3* there were no changes at all, meaning we are done.

## 2 Code generation (60%)

This section just contains screenshots of the code itself. The code itself is relatively small, and the source code contain in-depth explanations themselves.



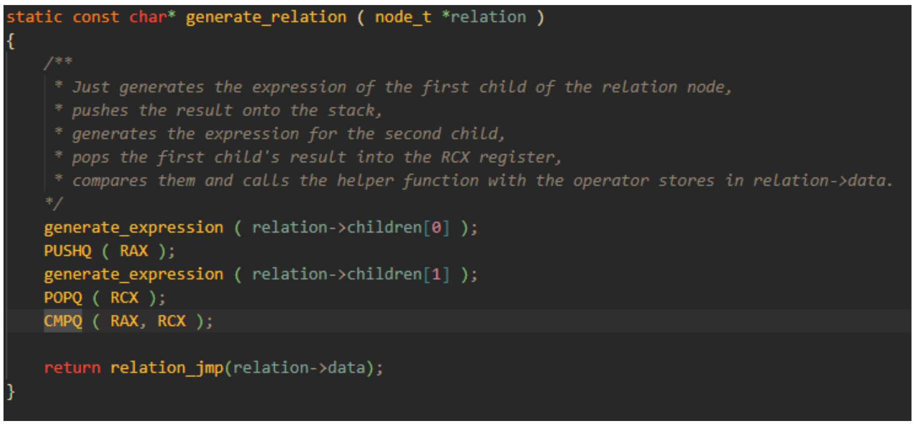
```
➜ vsl_programs make ps6-check
../build/vslc -c < ps6-codegen2/all_if_types.vsl > ps6-codegen2/all_if_types.S
gcc ps6-codegen2/all_if_types.S -o ps6-codegen2/all_if_types.out
../build/vslc -c < ps6-codegen2/break.vsl > ps6-codegen2/break.S
gcc ps6-codegen2/break.S -o ps6-codegen2/break.out
../build/vslc -c < ps6-codegen2/if.vsl > ps6-codegen2/if.S
gcc ps6-codegen2/if.S -o ps6-codegen2/if.out
../build/vslc -c < ps6-codegen2/sieve.vsl > ps6-codegen2/sieve.S
gcc ps6-codegen2/sieve.S -o ps6-codegen2/sieve.out
../build/vslc -c < ps6-codegen2/simple_break.vsl > ps6-codegen2/simple_break.S
gcc ps6-codegen2/simple_break.S -o ps6-codegen2/simple_break.out
../build/vslc -c < ps6-codegen2/simple_if.vsl > ps6-codegen2/simple_if.S
gcc ps6-codegen2/simple_if.S -o ps6-codegen2/simple_if.out
../build/vslc -c < ps6-codegen2/simple_while.vsl > ps6-codegen2/simple_while.S
gcc ps6-codegen2/simple_while.S -o ps6-codegen2/simple_while.out
../build/vslc -c < ps6-codegen2/while.vsl > ps6-codegen2/while.S
gcc ps6-codegen2/while.S -o ps6-codegen2/while.out
find ps6-codegen2 -wholename "*.vsl" | xargs -L 1 ./codegen-tester.py
Running 4 test cases for file ps6-codegen2/if.vsl
  Running ps6-codegen2/if.out 4
  Running ps6-codegen2/if.out 0
  Running ps6-codegen2/if.out 10
  Running ps6-codegen2/if.out -20
Running 1 test cases for file ps6-codegen2/simple_break.vsl
  Running ps6-codegen2/simple_break.out
Running 1 test cases for file ps6-codegen2/simple_while.vsl
  Running ps6-codegen2/simple_while.out 6
Running 1 test cases for file ps6-codegen2/while.vsl
  Running ps6-codegen2/while.out
Running 1 test cases for file ps6-codegen2/all_if_types.vsl
  Running ps6-codegen2/all_if_types.out
Running 3 test cases for file ps6-codegen2/simple_if.vsl
  Running ps6-codegen2/simple_if.out 7
  Running ps6-codegen2/simple_if.out 5
  Running ps6-codegen2/simple_if.out -2
Running 1 test cases for file ps6-codegen2/break.vsl
  Running ps6-codegen2/break.out
Running 2 test cases for file ps6-codegen2/sieve.vsl
  Running ps6-codegen2/sieve.out 600
  Running ps6-codegen2/sieve.out 100
No differences found in PS6!
```

Figure 8: PS6 Test Results

### 2.1 IF_STATEMENT nodes (30%)

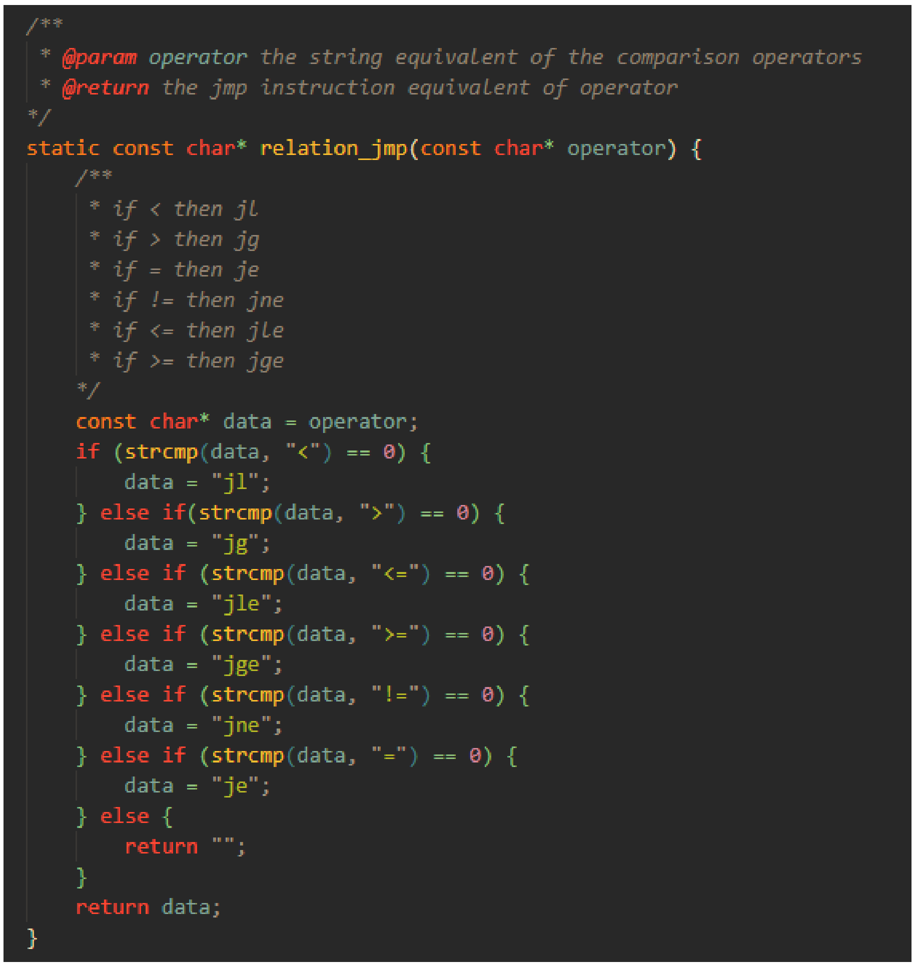
**Q3:** Implement generate_relation() in generator.c

**A:**



```
static const char* generate_relation ( node_t *relation )
{
    /**
     * Just generates the expression of the first child of the relation node,
     * pushes the result onto the stack,
     * generates the expression for the second child,
     * pops the first child's result into the RCX register,
     * compares them and calls the helper function with the operator stores in relation->data.
    */
    generate_expression ( relation->children[0] );
    PUSHQ ( RAX );
    generate_expression ( relation->children[1] );
    POPQ ( RCX );
    CMPQ ( RAX, RCX );

    return relation_jmp(relation->data);
}
```

Figure 9: Relation Node Handler

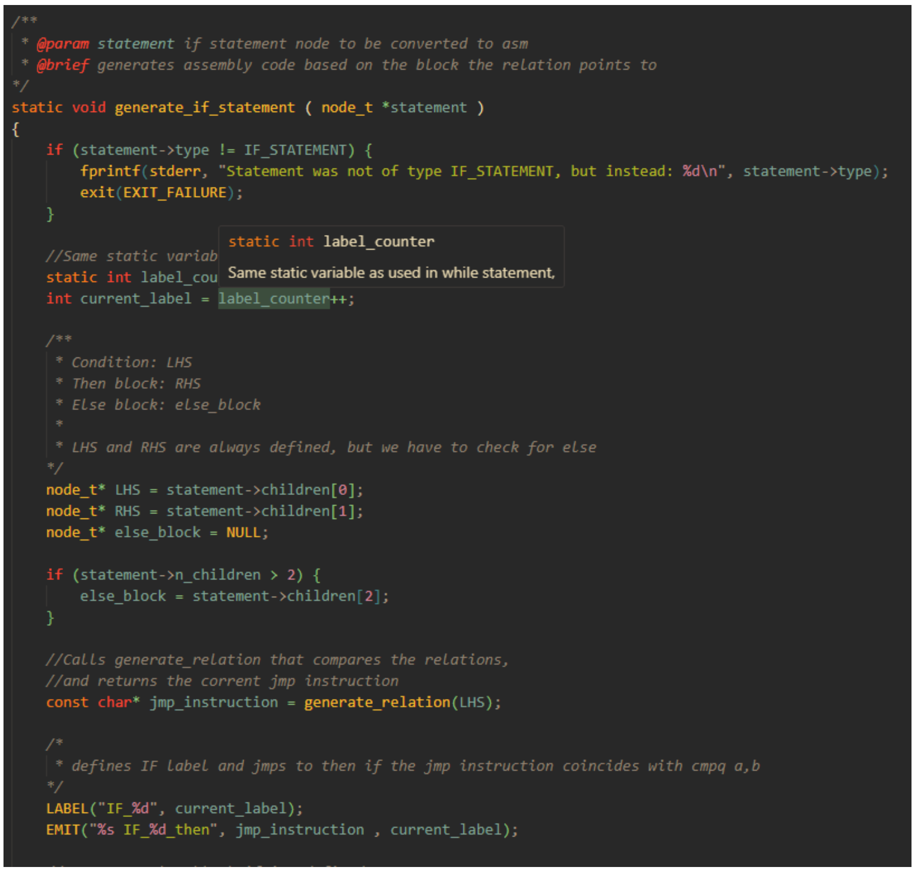


```
/**
 * @param operator the string equivalent of the comparison operators
 * @return the jmp instruction equivalent of operator
*/
static const char* relation_jmp(const char* operator) {
    /**
     * if < then jl
     * if > then jg
     * if = then je
     * if != then jne
     * if <= then jle
     * if >= then jge
    */
    const char* data = operator;
    if (strcmp(data, "<") == 0) {
        data = "jl";
    } else if(strcmp(data, ">") == 0) {
        data = "jg";
    } else if (strcmp(data, "<=") == 0) {
        data = "jle";
    } else if (strcmp(data, ">=") == 0) {
        data = "jge";
    } else if (strcmp(data, "!=") == 0) {
        data = "jne";
    } else if (strcmp(data, "=") == 0) {
        data = "je";
    } else {
        return "";
    }
    return data;
}
```

Figure 10: Relation operator to jmp instruction converter

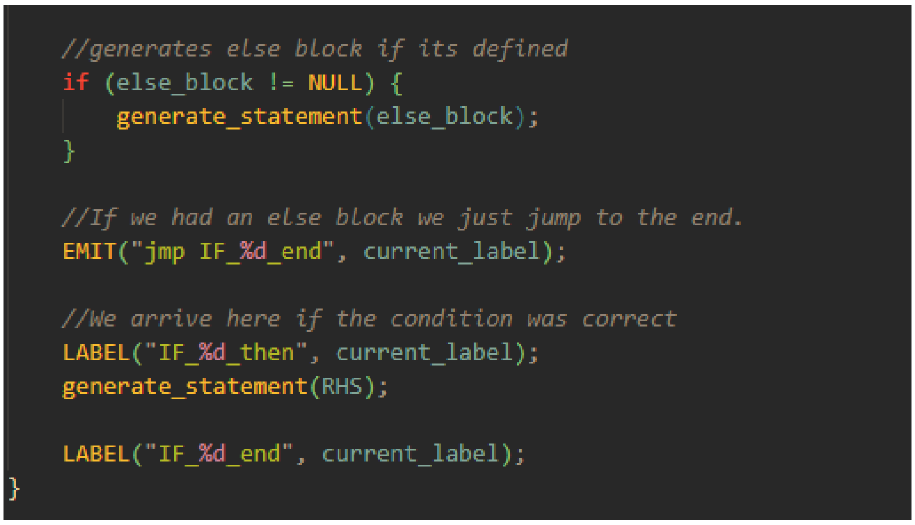
**Q4:** Now implement generate_if_statement(), which should handle both if-then and if-then-else statements

**A:**



```c
/**
 * @param statement if statement node to be converted to asm
 * @brief generates assembly code based on the block the relation points to
*/
static void generate_if_statement ( node_t *statement )
{
    if (statement->type != IF_STATEMENT) {
        fprintf(stderr, "Statement was not of type IF_STATEMENT, but instead: %d\n", statement->type);
        exit(EXIT_FAILURE);
    }
                      static int label_counter
    //Same static variab
    static int label_cou Same static variable as used in while statement,
    int current_label = label_counter++;

    /**
     * Condition: LHS
     * Then block: RHS
     * Else block: else_block
     *
     * LHS and RHS are always defined, but we have to check for else
    */
    node_t* LHS = statement->children[0];
    node_t* RHS = statement->children[1];
    node_t* else_block = NULL;

    if (statement->n_children > 2) {
        else_block = statement->children[2];
    }

    //Calls generate_relation that compares the relations,
    //and returns the corrent jmp instruction
    const char* jmp_instruction = generate_relation(LHS);

    /*
     * defines IF label and jmps to then if the jmp instruction coincides with cmpq a,b
    */
    LABEL("IF_%d", current_label);
    EMIT("%s IF_%d_then", jmp_instruction , current_label);
```

Figure 11: If Statement Part I



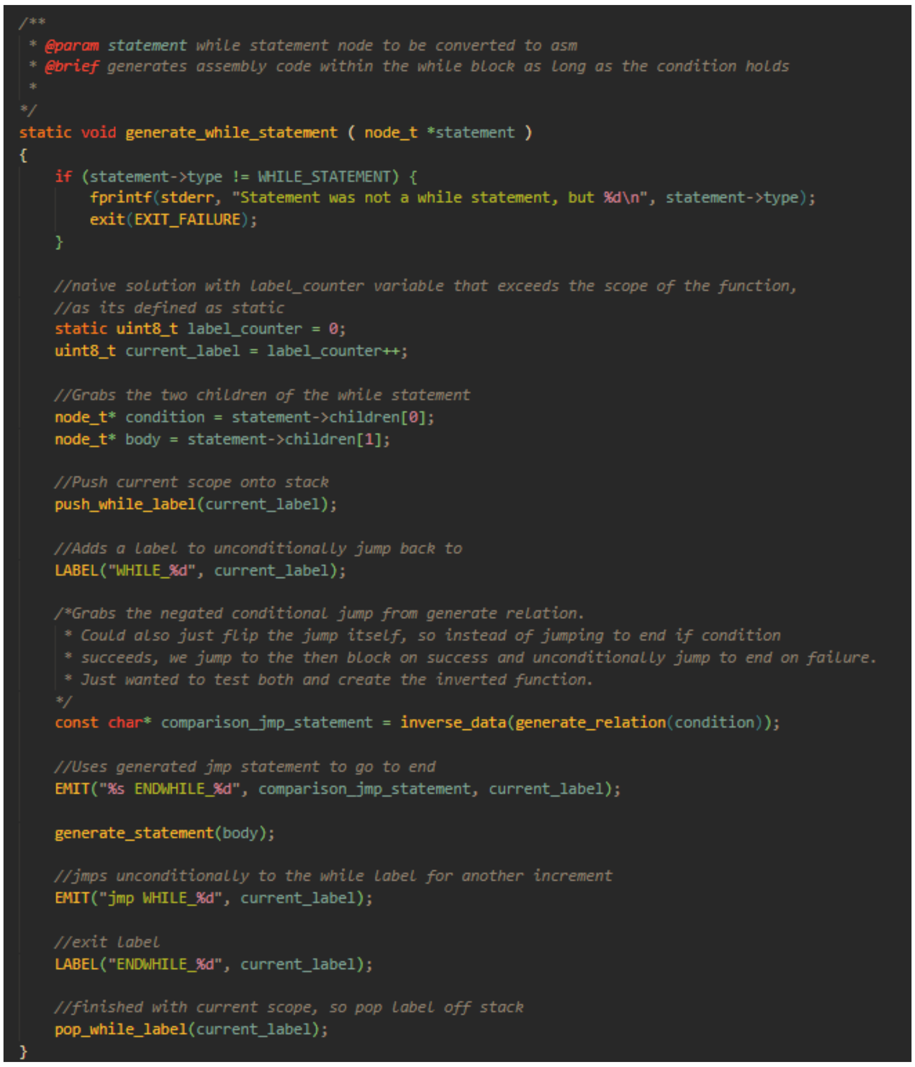
```c
    //generates else block if its defined
    if (else_block != NULL) {
        generate_statement(else_block);
    }

    //If we had an else block we just jump to the end.
    EMIT("jmp IF_%d_end", current_label);

    //We arrive here if the condition was correct
    LABEL("IF_%d_then", current_label);
    generate_statement(RHS);

    LABEL("IF_%d_end", current_label);
}
```

Figure 12: If Statement Part II

## 2.2 WHILE_STATEMENT nodes (20%)

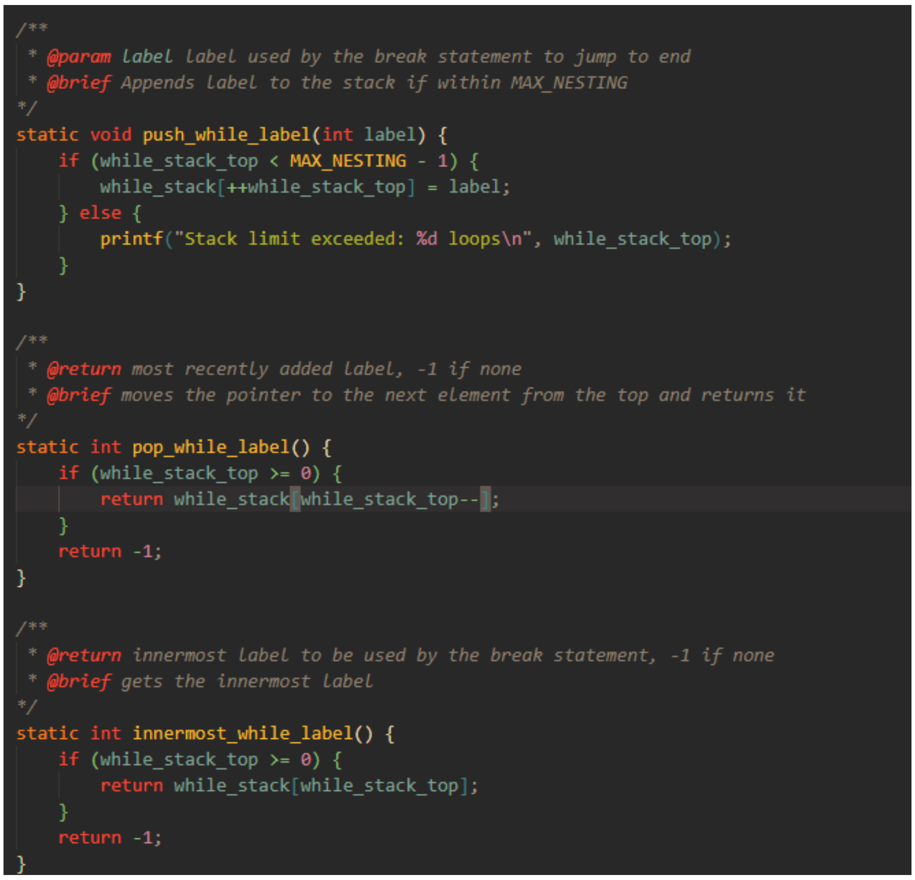
**Q5:** Implement generate_while_statement() to emit code for while statements.

**A:**



```c
/**
 * @param statement while statement node to be converted to asm
 * @brief generates assembly code within the while block as long as the condition holds
 *
*/
static void generate_while_statement ( node_t *statement )
{
    if (statement->type != WHILE_STATEMENT) {
        fprintf(stderr, "Statement was not a while statement, but %d\n", statement->type);
        exit(EXIT_FAILURE);
    }

    //naive solution with label_counter variable that exceeds the scope of the function,
    //as its defined as static
    static uint8_t label_counter = 0;
    uint8_t current_label = label_counter++;

    //Grabs the two children of the while statement
    node_t* condition = statement->children[0];
    node_t* body = statement->children[1];

    //Push current scope onto stack
    push_while_label(current_label);

    //Adds a label to unconditionally jump back to
    LABEL("WHILE_%d", current_label);

    /*Grabs the negated conditional jump from generate relation.
     * Could also just flip the jump itself, so instead of jumping to end if condition
     * succeeds, we jump to the then block on success and unconditionally jump to end on failure.
     * Just wanted to test both and create the inverted function.
    */
    const char* comparison_jmp_statement = inverse_data(generate_relation(condition));

    //Uses generated jmp statement to go to end
    EMIT("%s ENDWHILE_%d", comparison_jmp_statement, current_label);

    generate_statement(body);

    //jmps unconditionally to the while label for another increment
    EMIT("jmp WHILE_%d", current_label);

    //exit label
    LABEL("ENDWHILE_%d", current_label);

    //finished with current scope, so pop label off stack
    pop_while_label(current_label);
}
```

Figure 13: While Statement Generator



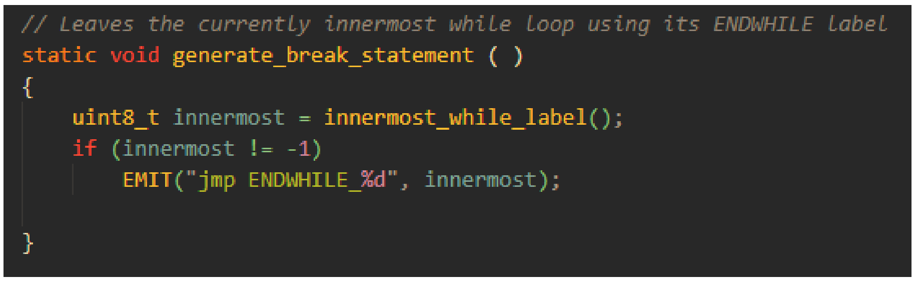
```c
/**
 * @param label label used by the break statement to jump to end
 * @brief Appends label to the stack if within MAX_NESTING
*/
static void push_while_label(int label) {
    if (while_stack_top < MAX_NESTING - 1) {
        while_stack[++while_stack_top] = label;
    } else {
        printf("Stack limit exceeded: %d loops\n", while_stack_top);
    }
}

/**
 * @return most recently added label, -1 if none
 * @brief moves the pointer to the next element from the top and returns it
*/
static int pop_while_label() {
    if (while_stack_top >= 0) {
        return while_stack[while_stack_top--];
    }
    return -1;
}

/**
 * @return innermost label to be used by the break statement, -1 if none
 * @brief gets the innermost label
*/
static int innermost_while_label() {
    if (while_stack_top >= 0) {
        return while_stack[while_stack_top];
    }
    return -1;
}
```

Figure 14: While Statement Scope Handlers

## 2.3 BREAK_STATEMENT nodes (10%)

**Q6:** Implement generate_break_statement() to emit code for leaving the currently innermost while-statement, and continuing execution beyond it.

**A:**



```c
// Leaves the currently innermost while loop using its ENDWHILE label
static void generate_break_statement ( )
{
    uint8_t innermost = innermost_while_label();
    if (innermost != -1)
        EMIT("jmp ENDWHILE_%d", innermost);

}
```

Figure 15: Break Statement Generator

## References

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