УНИВЕРСИТЕТ ИТМО

факультет программной инженерии и компьютерной техники

Направление подготовки 09.04.04 Системное и прикладное программное обеспечение

Системное програмное обеспечение Семестр 1

Лабораторная работа No2

Студент
Хуссейн, Авин
P4114

Преподаватель
Кореньков Ю.Д

Санкт-Петербург, 2024 г.

Table of Contents

Санкт-Петербург, 2024 г.

1. Goals

Implement control-flow graph (CFG) construction by syntax tree analysis for a set of input files. Process the information and generate representation of the results data format which supports graphical visualization.

2. Subtasks

- Write down a number of data structures intended to represent a certain information about the set of files, the set of subprograms and control-flow graphs
- Implement a module responsible for the control-flow graph construction for a given subprogram based on the syntax structure of its text.
- Generate a call graph describing for all the subprograms being analyzed.
- Implement test to demonstrate the completeness of the solution

3. Solution description

modules this stage of the solution is included in 2 files, “ControlGraph.c” and “controlGraph.h”

In the header file, we define different structures to represent the set of different files, the corresponding subprograms in each file, the control flow graphs for those subprograms, and finally the call graph that connects all of those files. In more detail we have the following structures

3.1 Modules Overview

The codebase is organized into several interrelated modules, each responsible for specific functionality. Below is an explanation of the primary modules and their purposes:

3.1.1. Type System Module

This module defines and manages types used in the program.

- **Structures and Enumerations**:
  - `SimpleType`: Enumeration of basic types (e.g., `TYPE_BOOL`, `TYPE_INT`, etc.).
  - `Type`: Encapsulation of types (simple or array) using a union for flexible representation.
  - `ArgumentDef`: Represents function arguments, including their name and type.
- **Responsibilities**:
  - Provide a robust system to handle simple and array types.
  - Enable type associations for variables and function arguments.

3.1.2. Control Flow Graph (CFG) Module

This module is central to constructing and managing the program's control flow graph (CFG).

- **Structures**:
  - `controlFlowGraphBlock`: Represents a single block in the control flow graph with associated metadata.

Санкт-Петербург, 2024 г.

  - `CfgsInfo`: Contains multiple CFGs and error information.
  - `Instructions`: Encapsulates instructions associated with a CFG block.
- **Responsibilities**:
  - Build CFGs for subprograms using the abstract syntax tree (AST) as input.
  - Define control flow block types (e.g., `BaseBlock`, `WhileBlock`, `IfBlock`).
  - Support integration with tools for visual representation of CFGs (e.g., DOT format).
- **Key Functions**:
  - `ConstructCFG*`: Functions to build CFG blocks for `if`, `while`, and base statements.
  - `writeDotGraph*`: Functions to export CFGs into DOT format for visualization.

3.1.3. Operation Tree (OT) Module

This module manages the representation of operation trees, which are essential for evaluating and optimizing expressions.

- **Structures**:
  - `OTNode`: Represents a node in the operation tree, which can be either an operator or an operand.
- **Responsibilities**:
  - Create and manage the tree structure for expressions.
  - Enable easy traversal and printing of operation trees for debugging and analysis.
- **Key Functions**:
  - `createOperatorNode`, `createOperandNode`: Create nodes for operators and operands, respectively.
  - `printTree`: Generate a string representation of the operation tree.

3.1.4. Subroutine Management Module

This module handles function and subroutine definitions within the program.

- **Structures**:
  - `Subroutine`: Represents a subroutine with its name, CFG, and signature details.
- **Responsibilities**:
  - Manage the definition and metadata of subroutines.
  - Link subroutines to their associated CFGs and signatures.
- **Key Functions**:
  - `DefineSubprogram`: Define subroutines from the given CFGs and parse tree.

3.1.5. Error Handling Module

This module collects and manages errors encountered during CFG construction or other stages.

- **Structures**:
  - `ErrorInfoCFG`: Represents individual errors with details like message, line, and position.
- **Responsibilities**:
  - Capture errors related to CFG construction or invalid syntax in the AST.

    - Provide a linked list-based mechanism to manage multiple errors.
- **Key Functions**:
    - `createErrorInfoNodeCFG`: Create error nodes.
    - `addErrorCFG`: Append errors to the error list.

3.1.6. Call Graph Module

This module manages the relationships between procedures and their call sites.

- **Structures**:
    - `callGraph`: Represents the call relationships among CFG blocks.
- **Responsibilities**:
    - Track procedure invocations within the control flow.
    - Enable analysis and visualization of call dependencies.
- **Key Functions**:
    - `insertCG`: Insert a call graph node for a procedure call.
    - `printCallGraph`: Print the call graph for debugging.

3.1.7. Utility and Helper Functions Module

This module provides various auxiliary functions used across other modules.

- **Responsibilities**:
    - Manage memory and string operations (e.g., `mystrcat`, `stringLen`).
    - Implement stack operations for managing open nodes in CFG construction.
- **Key Functions**:
    - `createStack`, `push`, `pop`: Manage stacks of parse tree nodes.
    - `estimatedSize`: Estimate the size of an operation tree.

3.1.8. AST Interfacing Module

This module interacts with the abstract syntax tree (AST) to build CFGs, operation trees, and other program structures.

- **Responsibilities**:
    - Interface between the AST and other modules like CFG and OT.
    - Extract relevant information from the AST for further processing.
- **Key Functions**:
    - `HandleType`: Extract type information from AST nodes.
    - `InsertInstruction`: Insert parsed instructions into the CFG.

3.1.9. DOT Export Module

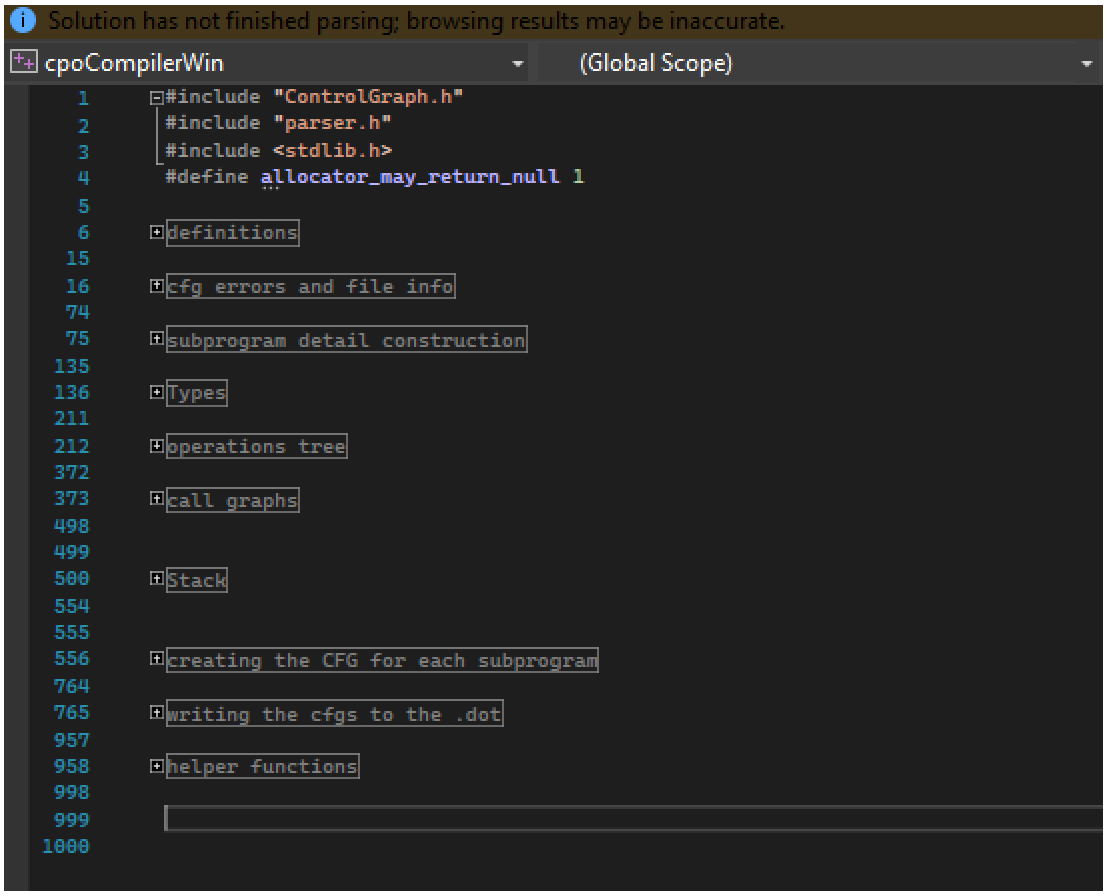
This module focuses on exporting CFGs and other graph structures into DOT format for visualization.

- **Responsibilities**:
    - Generate DOT files for control flow graphs and operation trees.
    - Provide tools for visual debugging.
- **Key Functions**:
    - `CFGToDotFile`: Export a CFG to a DOT file.
    - `writeDotGraph*`: Write specific block types into the DOT representation.

This modular design ensures clear separation of concerns, making the codebase maintainable, extensible, and easier to understand. Each module is designed to work cohesively, enabling the construction, analysis, and visualization of complex program structures like CFGs and call graphs.

4. Implementation details and source code examples

- **Exterior structure of the code** the code is devided into the following regions



```
Solution has not finished parsing; browsing results may be inaccurate.
cpoCompilerWin                    (Global Scope)
#include "ControlGraph.h"
#include "parser.h"
#include <stdlib.h>
#define allocator_may_return_null 1

definitions

cfg errors and file info

subprogram detail construction

Types

operations tree

call graphs


Stack


creating the CFG for each subprogram

writing the cfgs to the .dot

helper functions
```

Figure 1 exterior structure of ControlGraph.c

The given structure will be explained in further detail.

- **Creating the CFG for each subprogram** The first step in the solution is to create the control flow graphs that represent all the subprograms for a given file and do that for all the files and save this information about each file along with its name and errors should the processing module find any, in the struct that is defined as cfgFile.

- **Implementation** is done through the functions:
  - **cfgInterfacer:** which works as an interface between the file and its CFGs
  - **ConstructCFG (and variations like ConstructCFGWhileStatement, ConstructCFGIfStatement, ConstructCFGBaseStatement)**
    Which are the main functions that are responsible for creating the control flow graphs.

And many other helper fun

ctions for creating the cfg nodes which have the following structure

```
typedef struct controlFlowGraphBlock {
    struct ParseTree* ast;                          //< the corresponding AST
    enum BlockType blocktype;                     ///< The type of the control flow
graph block.
    struct controlFlowGraphBlock** nodes;   ///< Pointer to the outgoing nodes
(following blocks).
    //struct controlFlowGraphBlock** inNodes;   ///< Pointer to the outgoing nodes
(following blocks).
    //int inNodeCount;                          ///< Number of incoming nodes
(preceding blocks).
    int outNodeCount;                       ///< Number of outgoing nodes
(following blocks).
    Instructions* instructions;             // CFG Block Content
    //char** calledIndex;
    struct callGraph* called;
    //int calledFoundCnt;
    //int calledCnt;
    int drawn;
} controlFlowGraphBlock, * pControlFlowGraphBlock;
```

- **Input and output data** And the construction of a cfg is done with the help of a stack of ast nodes that was implemented in C so as to keep the order in which I must visit the abstract syntax tree's nodes to produce the correct structure. The construction functions are also given a start cfg node to continue from when creating the new node.
- **Example** to create cfgs for a given file in the main function just call

```
files[i]->cfgs = CFGInterfacer(files[i]->name, files[i]->ast);
```

Where you get back the cfgs for the file of index i, by inputting its name and ast

during the cfg construction process, I keep track of the instruction inside each block, these instructions can be var statements or expressions

- **Types (VarStatements)** for such nodes I keep track of the variable types, should they be simple or complex such as array types. I also create a var declaration for the given statement by holding information on the IDs of the vars and their types.
- **Operations tree construction(Expression)** in such cases I create the operations tree for the given expression by holding information of 2 types of nodes which are operators and operands. (I handle the type of the variable but not the type of the expressions yet).

- **Subprogram detail construction** The function DefineSubprogram processes a parse tree (ParseTree) to define subroutines within a program. It associates subroutines with their corresponding control flow graphs (CFGs) and captures details like the function name, signature, arguments, and return type.

```
Subroutine** DefineSubprogram(char* fileName, controlFlowGraphBlock** cfgs,
ParseTree* tree);
```

- **Input and output data**

1. **`fileName`** (`char*`):
    - Name of the file from which the parse tree originates.
    - Useful for context or error reporting but not directly utilized in the logic.
2. **`cfgs`** (`controlFlowGraphBlock**`):
    - An array of CFGs corresponding to each subroutine.
    - Each CFG represents the control flow logic of a subroutine.
3. **`tree`** (`ParseTree*`):
    - The abstract syntax tree (AST) for the program.
    - Expected to contain function definitions (`funcDefs`) and their details like function signature and body.

Returns an array of **`Subroutine*`** pointers:

- Each `Subroutine` represents one function defined in the input AST.
- A `Subroutine` includes:
    - The subroutine's name.
    - Its signature (name, arguments, return type).
    - A pointer to its CFG.

- **Call Graphs** the work on generating the call graphs is done in 2 steps.
    - The first step is storing the names of the called functions inside every cfg node, this one is simply done when handling the operations tree for a given expression in a subprogram.
    - The second step is done after all the cfgs are constructed for all the subprograms, and it is checking if the called procedure actually exists in which case it is added to the called functions parameter for the calling procedure, and if not, then I produce and error that there was a call for an undefined function.
    - When all the subprograms are processed I then print the results to a dot file, and then transform it to an image that shows the call graph for all the subprograms.

The way of using this functionality is by calling

```
    HandleCallGraphs(files, numberOfFiles);
```

which takes as an input the files (cfgFile) being processed as well as the number of them.

- **Stack** the stack is a typical stack that I implemented for ast nodes to keep track of the order in which I must get back to certain nodes in the tree so that the cfg is constructed correctly. This implementation holds the typical functionality of Push(), Pop(), isEmpty(), Peek() as well as create and delete stack ofcourse.
- **Writing the cfgs to a dot file** The provided functionality serializes a **Control Flow Graph (CFG)** into a **DOT file** format. The DOT file can then be visualized using tools like **Graphviz** to understand the program's execution paths, especially in control structures like **if**, **while**, and **base blocks**.

- **Core Functions**

    1. **`CFGToDotFile`**:
        - Entry point for generating the DOT file.
        - Initializes the file with graph settings and calls `writeDotGraph`.
    2. **`writeDotGraph`**:
        - Dispatches CFG blocks to appropriate handlers based on their type (e.g., `IfBlock`, `WhileBlock`).
    3. **`writeDotGraphIfStatement`**:
        - Handles `if` statements, writing "true" and "false" edges for branches.
    4. **`writeDotGraphWhileStatement`**:
        - Handles `while` loops, including edges for loop continuation (`True`) and exit (`False`).
    5. **`writeDotGraphBaseStatement`**:
        - Writes general statements not part of complex control structures.
    6. **`writeDotGraphOperationsTree`**:
        - Handles the operations within a CFG block, appending them to the DOT graph.
    7. **`printBlockToFile`**:
        - Writes a single block's label and instructions to the DOT file.

- **Input Data**

    1. **`cfg` (Root CFG Block)**:
        - Represents the control flow graph of a program or function.
        - Contains nodes (blocks) that include instructions and links to other nodes.
    2. **`fileName`**:
        - The name of the DOT file to be generated.
    3. **Additional Data**:
        - Each block in the CFG (`controlFlowGraphBlock`) contains:
            - `blocktype`: Specifies the type of block (e.g., `IfBlock`, `WhileBlock`).
            - `instructions`: Contains operations or statements.
            - `nodes`: Links to successor blocks.

- **Output Data** The output DOT file contains which is then transformed to a .png file.

- **Cfg errors and file info** the errors corresponding to each file were kept as a linked list that is printed by the main module.

5. Results

As a result of the work done in this lab I was able to implement control-flow graph (CFG) construction by syntax tree analysis for a set of input files. Process the information and generate representation of the results data format which supports graphical visualization.
The achievement of the tasks of the lab work were supported by testing the work on a set of 2 files, input.txt and input2.txt which have the following structure

```
function b()
   a();
   processIf();
   h();
   d();
end function
function d()
   dim arr as int (,)
   dim i as int
   sum = 2;
   i = 0;
   b();
   c();
   processAnotherArray(arr);
    while i>0
        if i>=sum then
             i=i-1;
           else
              i=i+1;
              break
        end if
     wend
end function
function processAnotherArray(arr as int(,)) as int
   dim sum as int
   dim i as int
   sum = 2;
   i = 0;
   processWhile(1,2);
   if i < 9 then
      sum = sum + i;
      i = i + 1;
   else
     sum=sum-i;
     while i>0
        if i>=sum then
             i=i-1;
           else
              i=i+1;
              break
```

```
                    end if
                  wend
               end if


               if i > 0 then
                      sum = sum + i;
                      i = i + 1;
               end if
               if i > 0 then
                      sum = sum + i;
                      i = i + 1;
               end if
end function
```

Input.txt

And the second input file has the following structure

```
function a()
dim arr as int (,)
dim i as int
sum = 2;
i = 0;
b();
c();
processArray(arr);
end function
function c()
 b();
 processWhile(1,2);
end function
function processWhile(i as int,j as int)
end function
function processArray(arr as int(,)) as int
   dim sum as int
   dim i as int
   sum = 2;
   i = 0;
   processWhile(1,2);
   if i < 9 then
      sum = sum + i;
      i = i + 1;
   else
     sum=sum-i;
     while i>0
     break
        if i>=sum then
              i=i-1;
           else
```

```
                    i=i+1;
                end if
              wend
            end if


            if i > 0 then
                sum = sum + i;
                i = i + 1;
            end if
            if i > 0 then
                sum = sum + i;
                i = i + 1;
            end if
end function
```

Input2.txt

And as a result, a subroutine structure was defined for every subprogram in the given set of files which contains information about the file name and the corresponding cfg. Some examples of the cfgs that were constructed are as follows

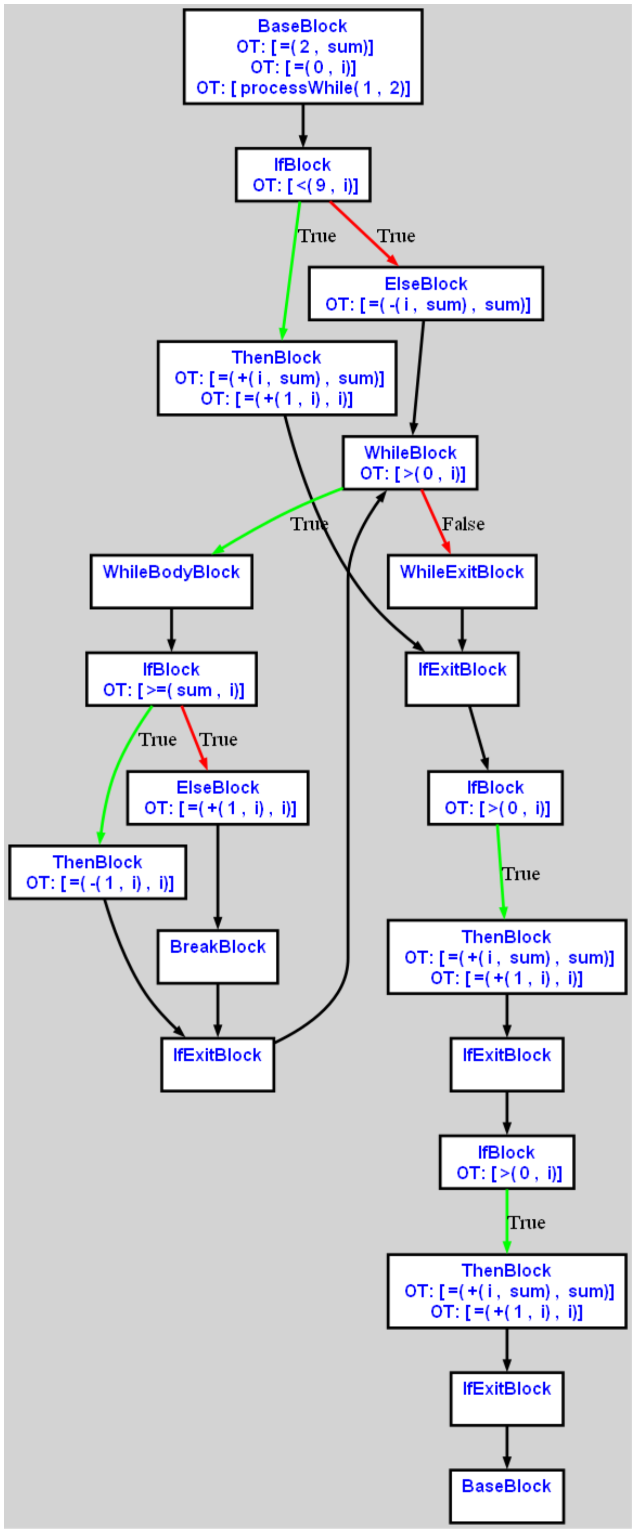


Figure 2 control flow graph of the function ProcessAnotherArray

Санкт-Петербург, 2024 г.

Another example is the cfg for the function d.

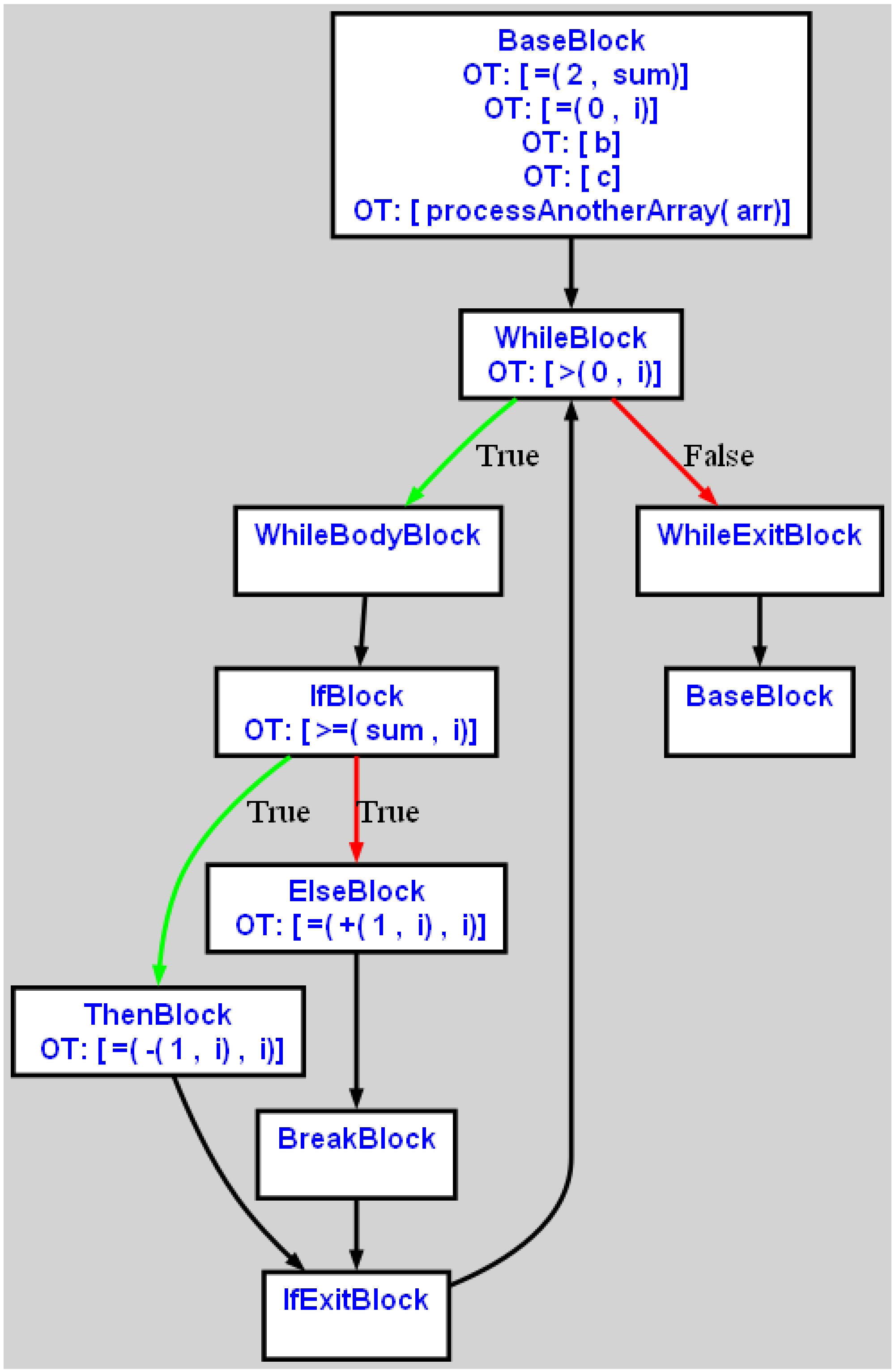


Figure 3 control flow graph of the function d

Санкт-Петербург, 2024 г.

And for the call graph we have the following result

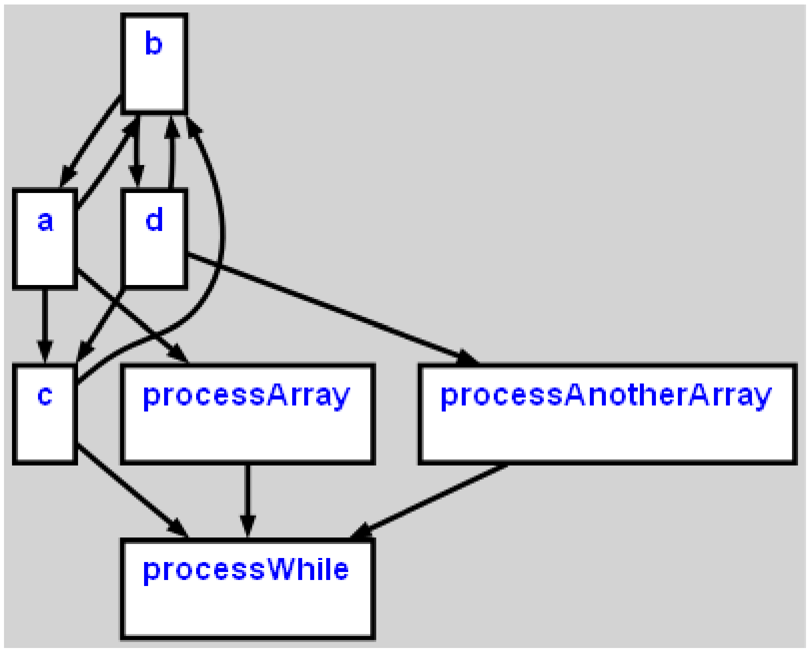
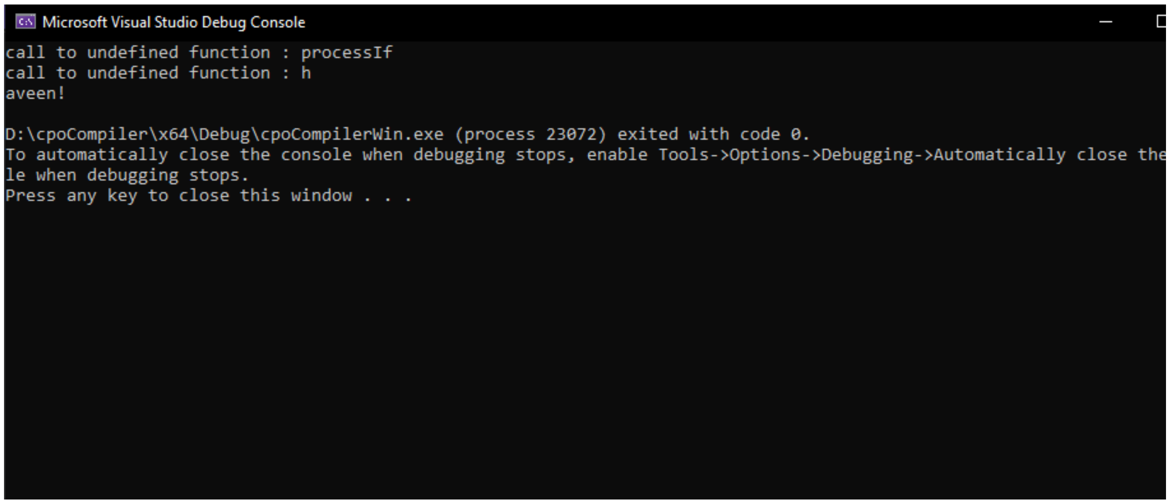


Figure 4 call graph

And for an example of the errors captured

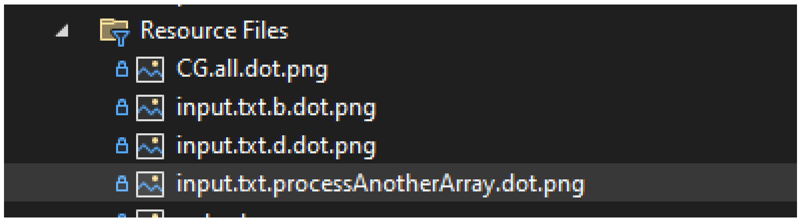


```
call to undefined function : processIf
call to undefined function : h
aveen!

D:\cpoCompiler\x64\Debug\cpoCompilerWin.exe (process 23072) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the
le when debugging stops.
Press any key to close this window . . .
```

Figure 5 an example of the errors captured by the module

The files in the folder have the following structure



```
Resource Files
    CG.all.dot.png
    input.txt.b.dot.png
    input.txt.d.dot.png
    input.txt.processAnotherArray.dot.png
```

Figure 6 the structure of the result files in the folder

Санкт-Петербург, 2024 г.

6. Conclusions

I believe the results of this lab work align with the goals. The data structures intended to represent the information about the set of files, the set of subprograms and control-flow graphs was implemented successfully. The control flow graphs were handled correctly based on the results of the tests and given the information received about the syntax tree. The call graph was also generated successfully, and all these results have been tested to show any errors found during the whole process.

During the process of completing this lab assignment I learned what a control graph is and I learned more about the natural flow of a program in a given language, I learned about operations trees and about the typing of variable. This process led to a deeper understanding of the underlying structure of compilers.

7. Link to the project on gitlab

https://gitlab.se.ifmo.ru/Aveena/cpoCompiler

Санкт-Петербург, 2024 г.