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 No5

Студент
Хуссейн, Авин
P4114

Преподаватель
Кореньков Ю.Д

Санкт-Петербург, 2025 г.

## Table of Contents

Санкт-Петербург, 2025 г.

## 1. Goals

To supplement the software package developed in previous assignments with support

for custom data types with the possibility of direct inheritance of members and additional

functionality according to the option.

**Variant**

- **Task 3: Dynamic typing, stack machine, common RAM**
- **Task 5: Overriding, Templating**

## 2. Subtasks

1. Extend the program developed in Task 3 by making the necessary changes in each of its modules to support polymorphism and type extension, according to variant a:

**a.** Define the necessary **data structures** to represent information about **user-defined types**, their **members** (fields and methods), and **relationships**.

**b.** Add support for **new syntactic constructs** in the **parser module** implemented in Task 1 (see additions to the syntactic model).

**c.** Add support for **new operations** related to working with type members in the **operation tree**, which is built by the module responsible for constructing the **control flow graph** from Task 2.

**d.** Support the **generation of appropriate instruction sequences** in the **linear code** for the newly added operations in the module developed in Task 3.

**e.** Using **data definition pseudo-instructions**, add information describing **user-defined types and their fields** (names and value types) to the **program structure information section**.

**f.** In the **console-based inspector application**, when displaying information about **argument values and local variables**, support the **output of values in the fields of user-defined types**.

2. Present the testing results in the form of a report, in accordance with the instructor's recommendations. The report should include:

**a.** A **description of changes** made to data structures and logic in the affected modules (items 1.a and 1.d), as well as the **internal organization** of the **program structure information section** (item 1.e).

**b. Examples** of the **new syntactic constructs and behaviors** added to the software system (items 1.b and 1.c).

**c. Examples of source code**, corresponding **assembly listings**, and **example outputs** of the test programs, as well as **output from the inspector program commands** at various stages of the test program's execution (items 1.b and 1.f).

Санкт-Петербург, 2025 г.

## 3. Solution description

The first changes were made to the file MyGrammar.g where we started by defining the language grammar to extend to custom types and templates as well as their related operations. After that the language extension was processed by extending the file ControlGraph.h where the required structures for classes and templates and new types (custom and generic) was added. Then, we processed these structures from the ast tree inside the file ControlGraph.c. After creating all the required classes and control flow graph, a second time traversal was done on the cfg in order to set the initialization of classes as well as the type substitution for generic and custom types, this was done by defining the needed structures in a new file "generic.h" and doing the graph traversal in the file "generic.c".

When the required typing was ready for templates, we moved to extend the unit responsible for generating the linear code "asm.c", we began by translating the new operators in the operation tree, related to custom types and their fields and functions. After that, we translated the needed structures for method tables. Finally, a solution was implemented to handle type casting in relation to dynamic typing with classes which was mainly done by extending both the architecture file "Architecture.pdsl" and the code generation unit "asm.c." A test example was constructed showing templates, overridden functions, casting and dynamically typed class function calls, as well as errors printed when the typing is handled incorrectly by the user.

### 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. Grammar extension

The grammar extension was done in the file MyGrammar.g. The added structures are:

```
classDef: 'class' ID ('(' custom ((','custom)*)? ')')? ('extends' ID ('(' typeRef ((',' typeRef)*)? ')')? )?
member* 'end' 'class' -> ^(ClassDef ID ^(Parameter custom+)? ^(Base  ^(ID typeRef*)?)? ^(Member
member*)) ;


member: modifier? (funcDef|field);
field: listIdentifier ('as' typeRef)? -> ^(Field listIdentifier typeRef?);
modifier: 'public'|'private';
funcDef: 'function' funcSignature statement* 'end' 'function' -> ^(FuncDef funcSignature statement*
'end');
```

In this section the classes were defined with the possibility to add generic types for templates.

The generic types themselves were introduced by extending the custom types defined previously

```
custom: ID ('<' typeRef (',' typeRef)* '>')? -> ^(ID typeRef*);
```

Finally the dot operator was added (".") to the grammar to allow for read and write operations from class fields

```
memberAccess
   : expr0 '.' expr3 expr4-> ^(expr4 ^('.' expr0 expr3 ) )
    | expr0 '.' ID expr4-> ^(expr4 ^('.' expr0 ID ) )
```

With that the grammar was ready to accept language classes.

### 3.1.2 Introduced class structures

The structures required to process the classes from the grammar were introduced in the module "ControlGraph.h". The first one being the class structure, which includes a list of parameters, functions and arguments, as well as the class name and it's parent which is noted as a Type structure in order for it to have the generic functionality for class templates.

```
typedef struct classDef {

    char* name;
    char** parameterNames;
    //classDef** parameters;

    int parametersCount;
   // bool isGenericParameter;

    Type* baseType;

    // Internal functions
    FunctionInfo** functions;
    int functionCount;

    // External functions
    ExternalFunctionInfo** externalFunctions;
    int externalFunctionCount;

    // Arguments
    ArgumentInfo** arguments;
    int argumentCount;

}classDef;
```

The second addition was introducing classInfo which serves as a class list used in returning the data to the main module

```
typedef struct classDefInfo {

    struct  classDef** classes;
    int classCount;
}classDefInfo;
```

And the different fields were processed by creating the required structures for the internal functions and arguments keeping track of their offset inside the class as well as their access modifiers.

```c
typedef struct FunctionInfo {
    enum AccessModifier modifier;
    int offset;
    struct Subroutine* subroutine;  // For internal functions
} FunctionInfo;


typedef struct ArgumentInfo {
    enum AccessModifier modifier;
    int offset;
    struct ArgumentDef* argument;
} ArgumentInfo;
```

And this list was returned with the result of the control flow graph building

```c
typedef struct CfgsInfo {
    controlFlowGraphBlock** cfgs;
    ErrorInfoCFG* errors;
    struct classDefInfo* classes;

} CfgsInfo;
```

A Generic type structure was also introduced to handle templates

```c
typedef struct GenericType {
    char* name;
    struct Type** parameters;
    int parametersCount;
} GenericType;
```

This structure allows for a generic type to have a list of parameters which themselves are Types, which allows for type nesting in a recursive manner.

This generic type was added to the original types created in previous labs which were only simple and array types

```c
typedef union {
    SimpleTypeStruct simpleType;
    ArrayType arrayType;
    GenericType genericType;
} TypeData;
```

The structure of the Type itself was also extended to include a reference to the class in case the type was custom or generic (user defined types)

```c
typedef struct Type {
    enum {
        TYPE_SIMPLE,
        TYPE_GENERIC,
        TYPE_ARRAY,
        TYPE_NONE
    } kind;               // To keep track of whether it's a simple type or
array
    TypeData data;        // Union to hold the type data
    struct classDef* def; // not NULL when references custom type
} Type;
```

A few functions were also added as helpers in the process of creating classes and defining their fields

```
// Creates a new class definition
classDef* createClassDef(const char* name);
classDefInfo* createClassDefInfo();
classDefInfo * addClassDefInfo(classDefInfo* classes, classDef* classdef);
```

### 3.1.3 Processing classes

Processing classes was implemented in "ControlGraph.c". The process started by parsing the classes from the ast into the class structures inside the function responsible interfacing between the control flow graph construction and ast parsing "cfgInterfacer"

```
    if (strcmp(tree->children[i]->token, "ClassDef") == 0) {
                char* baseClassName = NULL;
                classDef* class = createClassDef(tree->children[i]-
>children[0]->token);
                for (int x = 1; x < tree->children[i]->childrenCount; x++) {
                    if (strcmp(tree->children[i]->children[x]->token,
"Member") == 0) {
                        ParseTree* memberTree = tree->children[i]-
>children[x];
                        for (int j = 1; j < memberTree->childrenCount; j++)
{

                            AccessModifier modifire;
                            if (strcmp(memberTree->children[j - 1]->token,
"private") == 0) {
                                modifire = MODIFIER_PRIVATE;
                            }
                            else {
                                modifire = MODIFIER_PUBLIC;
                            }
                            if (strcmp(memberTree->children[j]->token,
"ExternFuncDef") == 0) {
                                ExternFuncDef* exter =
createExternFuncDef(memberTree->children[j]);
                                ExternalFunctionInfo* exterInfo =
createExternalFunctionInfo(modifire, class->externalFunctionCount, exter);
                                addExternalFunctionToClass(class,
exterInfo);
                            }
                            else {
                                if (strcmp(memberTree->children[j]->token,
"FuncDef") == 0) {
                                    Subroutine* func = createSubroutine();
                                    func->signatureDetails =
createSignatureDetails(memberTree->children[j]->children[0]);
                                    cfgsInfo->cfgs = processCfg(memberTree-
>children[j], cfgsInfo, fileName);
                                    func->cfg = cfgsInfo->cfgs[cfgsCount -
1];
                                    func->name = memberTree->children[j]-
>children[0]->children[0]->token;
                                    FunctionInfo* funcInfo =
createFunctionInfo(modifire, class->functionCount, func);
                                    addFunctionToClass(class, funcInfo);
                                }
                                else {
                                    if (strcmp(memberTree->children[j]-
>token, "Field") == 0) {
                                        ParseTree* argsTree = memberTree-
>children[j]->children[0];
                                        for (int k = 0; k < argsTree-
>childrenCount; k++) {
                                            ArgumentDef* argDef =
(ArgumentDef*)malloc(sizeof(ArgumentDef));
                                            argDef->name = argsTree-
>children[k]->token;
                                            if (memberTree->children[j]-
>childrenCount > 1) {

                                                argDef->type =
HandleType(memberTree->children[j]->children[1]);
                                            }
                                            else {
                                                argDef->type =
malloc(sizeof(Type));
                                                argDef->type->kind =
TYPE_NONE;

                                            }
                                            ArgumentInfo* argInfo =
createArgumentInfo(modifire, class->argumentCount, argDef);
                                            addArgumentToClass(class,
argInfo);
                                        }
                                    }
                                }
                            }
                        }
                    }
                    else {
                        if (strcmp(tree->children[i]->children[x]->token,
"Base") == 0) {
                            if (tree->children[i]->children[x]->children[0]-
>childrenCount > 1) {

                                Type* base = create_generic_type(tree-
>children[i]->children[x]->children[0]->token, tree->children[i]-
>children[x]->children[0]->childrenCount, tree->children[i]->children[x]-
>children[0]);
                                class->baseType = base;

                            }
                            else {

                                Type* base = create_simple_type(TYPE_CUSTOM,
tree->children[i]->children[x]->children[0]->token);
                                class->baseType = base;
                            }
                        }
                        else {
                            if (strcmp(tree->children[i]->children[x]-
>token, "Parameter") == 0) {
                                for (int y = 0; y < tree->children[i]-
>children[x]->childrenCount; y++) {
                                    addParameterName(class, tree-
>children[i]->children[x]->children[y]->token);
                                }
                            }
                        }
                    }
                }

                cfgsInfo->classes = addClassDefInfo(cfgsInfo->classes,
class);
            }
```

Several helper functions were implemented to help create a class or add a field to it, but will not be included in this report for redundancy.

A function was added to create generic types for templates, the function is called from another function called “**HandleType**” when a generic type is present and it recursively calles **HandleType** on all of its parameters to properly handle nested types in templates

```
Type* create_generic_type(char* class_name, int param_count, ParseTree*
base) {

    Type* type = (Type*)malloc(sizeof(Type));
    type->kind = TYPE_GENERIC;
    type->data.genericType.name = class_name;
    type->data.genericType.parametersCount = param_count;
    type->data.genericType.parameters = malloc(sizeof(Type) * param_count);
    for (int i = 0; i < param_count; i++) {
        type->data.genericType.parameters[i] = HandleType(base-
>children[i]);
        // set the parameters
    }
    type->def = NULL;
    return type;


}
```

Inside the function “HandleOperationTree”, an extension was performed to process the new memberAccess operator by creating a three operand structure for it and supplementing the operations “readFromField” and “writeInField”

```
OTNode* HandleOperationsTree(ParseTree* base) {
    OTNode* OT;
    if (strcmp(base->token, "CallOrIndexer") == 0) {

        OT = createOperatorNode(base->children[1]->token);


        insertCGToken(base->children[1]->token);

        base = base->children[0];
    }
    else {
        if (strcmp(base->token, "Unary") != 0) {

            if (base->childrenCount > 0) {
                if (strcmp(base->token, "=") == 0) {

                    if (strcmp(base->children[1]->token, ".") == 0) {
                        OT = createOperatorNode(".="); //Write in field
                        OT->cntOperands = 3;
                        OT->operands = malloc(sizeof(OTNode*) * 3);
                        OT->operands[0] = HandleOperationsTree(base-
>children[1]->children[0]);
                        OT->operands[1] = HandleOperationsTree(base-
>children[1]->children[1]);
                        OT->operands[2] = HandleOperationsTree(base-
>children[0]);
                        return OT;
                    }
                    if (strcmp(base->children[0]->token, ".") == 0) {
                        OT = createOperatorNode("=."); // read from field
                        OT->cntOperands = 3;
                        OT->operands = malloc(sizeof(OTNode*) * 3);
                        OT->operands[0] = HandleOperationsTree(base-
>children[1]);
                        OT->operands[1] = HandleOperationsTree(base-
>children[0]->children[0]);
                        OT->operands[2] = HandleOperationsTree(base-
>children[0]->children[1]);
                        return OT;

                    }
```

With that the previously constructed units “ControlGraph.c” and “ControlGraph.h” were now extended to support templates and user defined types.

### 3.1.4 User defined types (custom and generic)

Handling type substitution and class initialization was processed by a second traversal to the control flow graph after its construction. This process was defined in a module called “generic.c”.

Firstly, a function named “Set Type” works as an interface for calling “TraverseCfgType” on each functions’s cfg

It is important to note that this function does 2 things in the right order:

1- It sets the parent types for classes which are inherited (function “findClass” which will be discussed later)
2- It calls for cfg traversal for type substitution after the first step is successful

```
classSubrountineInfo* setTypes(Subroutine** subroutines, int cnt, char*
fileName, classDefInfo* classes) {


    for (int i = 0; i < classes->classCount; i++)
    {
        // for extends

        if (classes->classes[i]->baseType)
        {
            classes->classes[i]->baseType = findClass(classes, classes-
>classes[i]->baseType);
            classes->classes[i] = fixOffsetLikeFather(classes->classes[i]);

        }
    }

    for (int i = 0; i < cnt; i++) {

        subroutines[i]->cfg = traverseCfgType(subroutines[i]->cfg, NULL,
fileName, classes);

    }
    classSubrountineInfo* info = malloc(sizeof(classSubrountineInfo));
    info->cls = malloc(sizeof(classDefInfo));
    info->cls = classes;
    info->subs = malloc(sizeof(subroutineInfo*) * cnt);
    info->subs = subroutines;
    return info;
}
```

Traversing the cfg to set the type is a simple recursive process done on the different block types

```
controlFlowGraphBlock* traverseCfgType(controlFlowGraphBlock* cfg,
controlFlowGraphBlock* start, char* fileName, classDefInfo* classes) {

    switch (cfg->blocktype)
    {
    case IfBlock:
        ret = traverseCfgIfStatementType(cfg, start, fileName, classes);
        break;
    case WhileBlock:
        if (cfg->drawn == 0) {
            cfg->drawn = 1;
            ret = traverseCfgWhileStatementType(cfg, start, fileName,
classes);
        }
        else if (cfg->drawn == 1) {
            cfg->drawn = 2;
            return copyNode(cfg);
        }
        else {
            return copyNode(cfg);
        }
        break;
    case BreakBlock:


        return copyNode(cfg);
        break;
    case BaseBlock:
        if (cfg->outNodeCount <= 0) {


            return copyNode(cfg);
        }
        ret = traverseCfgBaseStatementType(cfg, start, fileName, classes);
        break;
    default:
        if (cfg->blocktype == IfExitBlock) {
            cfg->drawn++;
        }
        if (cfg->blocktype == IfExitBlock && cfg->drawn > 1) {
            cfg->drawn = 0;

            break;
        }

        if (cfg->outNodeCount <= 0) {


            return copyNode(cfg);
        }
        ret = traverseCfgBaseStatementType(cfg, start, fileName, classes);
        break;
    }
    return ret;

}
```

The important thing in this function is that a deep copy process happens inside each block traversal, by which type substitution happens if needed (generic and custom types) and if it's not needed then the nodes are deep copied as is.

```
controlFlowGraphBlock* copyNode(controlFlowGraphBlock* node) {
    controlFlowGraphBlock* copy = createCFGBlock(node->ast, node-
>blocktype);
    copy->instructions = copyInstructions(node);
    copy->drawn = node->drawn;
    copy->outNodeCount = node->outNodeCount;
    copy->called->cnt = node->called->cnt;
    copy->called->calledProcedures = realloc(copy->called->calledProcedures,
sizeof(controlFlowGraphBlock*) * node->called->cnt);
    for (int i = 0; i < node->called->cnt; i++) {
        copy->called->calledProcedures = node->called->calledProcedures[i];
        copy->called->calledTokens[i] = node->called->calledTokens[i];
    }
    copy->nodes = realloc(copy->nodes, sizeof(controlFlowGraphBlock) * node-
>outNodeCount);
    return copy;
}
```

And this is being handled when copying the instruction

```
cfgBlockContent* copyInstructionsVarStatement(cfgBlockContent* old) {
    // add a var instruction
    cfgBlockContent* content = malloc(sizeof(cfgBlockContent));
    char** ids = (char*)malloc(sizeof(char*) * old->varDec->cntId);
    for (int k = 0; k < old->varDec->cntId; k++) {

        ids[k] = old->varDec->Ids[k];
    }

    Type* instructionType = matchGenericType(old->varDec->type, current,
currentGeneric);
    varDeclaration* varDecl = CreateVarDeclaration(ids, old->varDec->cntId,
instructionType);
    content->type = TYPE_VARDECLARATION;
    content->varDec = varDecl;  // Assign var declaration to the union
    return content;
}
```

This process calls the function “matchGenericType” which is the one responsible for type substitution when needed, it takes as an input the initialized generic type as well as the class it’s initializing and the key that is being substituted for a type.

```
/// here I match the type key, from the parameters
///summary
/// let's take an example A<T> is later initialized as A<int>
///
/// T is key
/// class is A<T> (A as an unitialized class
///
/// generic type is A<int>  (as a type)
///
///
///
Type* matchGenericType(Type* key, classDef* class, Type* genericType) {
    if (key->kind == TYPE_SIMPLE) {
        if (key->data.simpleType.type == TYPE_CUSTOM) {

            for (int k = 0; k < class->parametersCount; k++) {

                if (strcmp(key->data.simpleType.custom_id, class-
>parameterNames[k]) == 0) {
                    //here we found an argument which has a generic type for
example T a;
                    return  genericType->data.genericType.parameters[k];

                }
                else {

                }

            }
            // it is not a param , must be another class
            return findClass(allClasses, key);
        }
    }
        if (key->kind == TYPE_GENERIC) {


            //// here I basically treat something like A<T, Z>, T and Z are
params, but the base of the type A must be a class

                return findClass(allClasses, key);

        }


}
```

And this function is calling the function findClass which is responsible for matching the new initialized class with a class from the previously defined classes, using the class's name

```
Type* findClass(classDefInfo* classes, Type * parent) {
    allClasses = classes;
    for (int j = 0; j < classes->classCount; j++) {
        char* parentName = get_type_name(parent);
        if (strcmp(parentName, classes->classes[j]->name) == 0) {
            classDef* found = classes->classes[j];

            if (found->parametersCount == 0) {
                //custom
                parent->def = found;
            }
            else {
                //generic


                parent->def= initClass(found, parent, classes);

            }
        }
    }
```

```
        return parent;}
```

This function has 2 possible outcomes

1- If the initialized class is not a template, then it finds it in our list of classes and returns it
2- If the initialized class is a template, then it initializes it with the required types for its parameters using the function “initClass”

```

classDef* initClass(classDef* found, Type* parent , classDefInfo * classes) {
  classDef* newClass = createClassDef(found->name);
  if (found->baseType) {
    newClass->baseType = found->baseType;

  }
     newClass->parametersCount = parent->data.genericType.parametersCount;
     newClass->parameterNames = malloc(sizeof(char*) * newClass->parametersCount);
     for (int i = 0; i < newClass->parametersCount; i++) {
       newClass->parameterNames[i] = found->parameterNames[i];
     }
     newClass->argumentCount = found->argumentCount;
     newClass->arguments = malloc(sizeof(ArgumentInfo*) * found->argumentCount);
     for (int i = 0; i < found->argumentCount; i++) {
       ArgumentDef* argDef = malloc(sizeof(ArgumentDef));
       argDef->name = found->arguments[i]->argument->name;
       argDef->type = matchGenericType(found->arguments[i]->argument->type, found, parent);

       newClass->arguments[i] = createArgumentInfo(found->arguments[i]->modifier, found->arguments[i]->offset, argDef);

     }
     // what to do with the newClasses functions? if they are found withing=

     for (int i = 0; i < found->functionCount; i++) {
       Subroutine* func = createSubroutine();
       SignatureDetails* details = malloc(sizeof(SignatureDetails));
       details->arguments = malloc(sizeof(ArgumentDef*) * found->functions[i]->subroutine->signatureDetails->cntArgs);
       for (int l = 0; l < found->functions[i]->subroutine->signatureDetails->cntArgs; l++) {
         ArgumentDef* argDef = malloc(sizeof(ArgumentDef));
         argDef->name = found->functions[i]->subroutine->signatureDetails->arguments[l]->name;
         argDef->type = matchGenericType(found->functions[i]->subroutine->signatureDetails->arguments[l]->type, found, parent);

         details->arguments[l] = argDef;
```

```


                }


                func->name = found->functions[i]->subroutine->name;
                func->signatureDetails = details;
                current = newClass;
                currentGeneric = parent;
                inClass = 1;
                ret = NULL;
                // should create the cfg and copy and substitute types
                func->cfg =traverseCfgType(found->functions[i]->subroutine->cfg, NULL, "fileName", classes, NULL);
                FunctionInfo* funcInfo = createFunctionInfo(found->functions[i]->modifier, i, func);
                addFunctionToClass(newClass, funcInfo);

            }
            return newClass;

}

```

This function deep copies the class nodes and substitutes the types wherever necessary, it also calls “match_generic_type ” on the parameters in order to recursively handle them which allows for type nests inside the templates.

A final note inside this module is that we also keep track of the offset of the functions and parameters inside the classes and make sure we have the correct order which aligns with the parent classes and this is important for translating the method tables in later stages (not all functions are provided here for redundancy).

### 3.1.5 Code generation

Translating the classes is done by extending the code generation module “asm.c”. We start by defining the method tables using a recursive function that looks through the classes and their parents in order to translate the tables of methods correctly for each.

```
int translate_vtable(classDef* class){

    // recursively get fathers and mark certain funcs as treated
    if (class->baseType)
    {
        translate_vtable(class->baseType->def);

    }
    for (int i = 0; i<class->functionCount; i++) {
        int isOverride = 0;
        if ((strcmp(class->name, curr->name) != 0)) {


        for (int j = 0; j < curr->functionCount; j++) {
            if (strcmp(curr->functions[j]->subroutine->name, class-
>functions[i]->subroutine->name) == 0) {
                isOverride = 1;
                curr->functions[j]->subroutine->isOverride = 1;
                class->functions[i]->subroutine->isOverride = 1;
                put_label_func_vtable(curr->name, class->functions[i]-
>subroutine->name);
                put_label_func_offset_vtable()
            }
        }
        }
        if (!class->functions[i]->subroutine->isOverride) {

        put_label_func_vtable(class->name, class->functions[i]->subroutine-
>name);
        put_label_func_offset_vtable()

        }
        class->functions[i]->subroutine->isOverride = 0;
    }


}
```

The structure of the vtables themselves would look like this in the listing for example:

```
vtable_ExampleClass:
  dq 3
  dd ExampleClass_DoThing
  dd 1
  dd Example_DoOtherThing
  dd 2
  dd ExampleClass_ShowMessage
  dd 3
vtable_Example:
  dq 2
  dd Example_DoThing
  dd 4
  dd Example_DoOtherThing
  dd 5
```

In this example the class ExampleClass inherits the class Example and overrides DoThing, it does not override DoOtherThing and it also has an extra function of its own “ShowMessage”

After that we also have a translate the functions inside the class just like we would translate any other function except that we put the name of the class write before the name of the function inside the listing (e.g ExampleClass_DoThing)

Translating the actual class happens when a user defined field is first initialized in the input text. In this case we create a heap structure inside the listing using our architecture by extending it using an extra register “heap_base” this register is used as a pointer inside the heap. As previously mentioned we defined a class definition inside the Type structure and this class is set to a specific class at the stage in which we did class initialization and type substitutions, so we know for sure that no type has a class def unless it is a user defined one so we basically do

```
                if (locals->type->def) {

                    push_hf()
                    pop_vtable(mystrcat("vtable_", locals->type->def->name))
                    for (int k = 0; k < locals->type->def->argumentCount;
k++) {
                        pop_hf();
                        pop_hf();
                    }



                    translate_type(locals->type);
                    add_s(locals->offset)
                    wide_store()
                    locals = locals->next;
                }
```

And what this does is it allocates a place on the heap for the classes arguments based on their order, as well as push the value of a pointer to the virtual table of the class itself to the stack, which we can now get from the stack and store inside the variable which has a custom or generic type. Here's a look into this instruction inside the architecture

```
  instruction pop_hf ={0010 0000} {
          heap_base = heap_base + 4;      // Increment stack pointer

    ip = ip + 1;      // Increment instruction pointer
};
  instruction pop_vtable ={0010 0001, imm16 as value} {

    ram:2[heap_base]=value;
          heap_base = heap_base + 4;      // Increment stack pointer

    ip = ip + 3;      // Increment instruction pointer
};
```

We also translate both “writeInField” and “readFromField”

```
      else if (strcmp(tree->value.operator,"=.") == 0) {
              // read from field

                translateOT(tree->operands[1], fileName, 0); // this puts
the value and type of the object on the stack
                Type* type = findVarType(tree->operands[1]->value.operand);
                if (tree->operands[2]->type == NODE_TYPE_OPERATOR) {
                    curr = type->def;
                    int funcOffset = get_func_offset(type->def, tree-
>operands[2]->value.operator)+1;
```

```
                    for (int i = 0; i < tree->operands[2]->cntOperands; i++)
{
                            translateOT(tree->operands[2]->operands[i],
fileName, 0);
                    }
                    load() // load from heap 0, as in get vtable ptr
                    add_mem_imm(funcOffset) // get the func in vtable
                    funcOffset = 0;
                    load()// get the func
                    pop()// pop filler type value
                    call_from_stack()
                    wide_store()


                }
                else{

                int arg_offset = translate_class(type->def, tree-
>operands[2]->value.operand)+1;
                add_mem_imm(arg_offset)
                load()
                translateOT(tree->operands[0], fileName, 1); // this puts
the value and type of the object's vtable (we stored a pointer to it in the
var) on the stack
                wide_store()

                }

            }
            else if (strcmp(tree->value.operator,".=") == 0) {
                translateOT(tree->operands[2], fileName, 1); // this puts
the value and type of the object on the stack

                translateOT(tree->operands[0], fileName, 0); // this puts
the value and type of the object on the stack
                Type* type = findVarType(tree->operands[0]->value.operand);
                int arg_offset = translate_class(type->def, tree-
>operands[1]->value.operand)+1;
                add_mem_imm(arg_offset)
                wide_store()

            }
```
This process basically checks if we are accessing a field or a function and getting the respective offset of the argument or the function and otherwise translate them as we would do a normal argument or a function.

The only problem that remains to be solved at this stage is how to do type casting when the types are dynamically defined and could change at runtime which would change the vtable of the variable. The solution proposed to this challenge was to keep track of a global offset for each function in the entire program and store this offset in the vtable itself. We stored in each vtable:

1- the number of the functions
2- pointers to the functions

3- the offset of each function next to it

Then inside the code generation unit we defined a custom function M(o, n) which as an input takes an object "o" and a number "n" which is the offset of the required function.

Upon translating M, we look inside the vtable of the object o and check if a certain function with that offset exists inside the vtable. If yes, we call it. If not, we print an error message "This method does not exist in this class object".

This happens inside the translation unit upon the call of M

```
    else if (strcmp(tree->value.operator,"M") == 0) {
                    translateOT(tree->operands[0], fileName, 0); // this
puts the value and type of the object on the stack

                    load() // load from heap 0, as in get vtable ptr
                        translateOT(tree->operands[1], fileName, 0); //
this puts the value and type of the object on the stack

                        check_vtable()
                        char* error_label = labelName();
                        char* call_label = labelName();
                        char* end_label = labelName();
                        jz(error_label)
                        jump(call_label)
                        // put a label inside which there is call from
stack
                        ret()
                        put_label(error_label)
                        call("error")
                        jump(end_label)

                        put_label(call_label)
                        //popOut()
                        call_from_stack()
                        jump(end_label)
                        put_label(end_label)


            }
```

And the checking process itself is done inside the architecture in the instruction "check_vtable" which loops through all the functions looking for this offset and returns either a pointer to said function having found it, or simply an error flag.

```
   instruction check_vtable = { 0010 0101 } {
         sp=sp + 2;
         let type=m_stack:2[sp];//pop type
         sp=sp + 2;
         let func = m_stack:2[sp];//pop value

         sp= sp + 2; // pop the pointer to the value in ram
         let ptr= m_stack:2[sp];
         let y = ram:2[ptr];
```

Санкт-Петербург, 2025 г.

```
        ptr = ptr + 4;
        let found = 0;
        while y > 0 do{
            ptr = ptr + 8;
            let value = ram:2[ptr];
            if value == func then
            {
            found = 1;
            m_stack[sp]= ram:2[ptr - 4];
            sp = sp - 2;// after this I need some for loop to check all
the funcs
            ip = ip + 1;
            break;
            }
            y = y - 1;
            }
        if found == 0 then {
            m_stack[sp]= 0;
            sp = sp - 2;// after this I need some for loop to check all
the funcs

        }
        ip = ip + 1;

  };
```

## 5. Results

As a result of the work done in this lab I was able to supplement the software package developed in previous assignments with support for custom data types with the possibility of direct inheritance of members and additional

functionality according to the option.

- **Task 3: Dynamic typing, stack machine, common RAM**
- **Task 5: Overriding, Templating**

We can see the architecture used in this lab in the file “architecture.pdsl”

```
architecture spo {
/*
    case                    18
    data word length        2
    code model              m_stack
    spaces                  common ram
*/
registers:

  storage sp [16];      // Stack pointer
  storage ip [16];      // Program counter (Instruction pointer)
  storage ebp [16];
  storage esp [16];
  storage inp [16];     // Input register (for input operations)
  storage outp [16];    // Output register (for output operations)
  storage heap_base[16];

memory:

  range ram [0x0000..0xffff] {   // Common RAM space for code/data
    cell = 16;     // 8-bit cells
    endianess = little-endian;  // Little endian
    granularity = 2;  // 2-byte chunks
  }

  range m_stack [0x0000..0xffff] {   // Stack space
    cell = 16;
    endianess = little-endian;
    granularity = 2;
  }

instructions:
    encode imm16 field = immediate [16];
    encode reg field = register {
        esp = 0,
        ebp = 1};

 // Push immediate value onto the stack
  instruction push = { 0000 0000, imm16 as value } {
    m_stack:2[sp] = value;
    sp = sp - 2;   // Decrement stack pointer
    ip = ip + 3;   // Increment instruction pointer
  };
  // Pop value from the stack into outp
  instruction pop = { 0000 0001 } {
    sp = sp + 2;        // Increment stack pointer
    ip = ip + 1;        // Increment instruction pointer
  };

  instruction popOut = { 0000 0010 } {
    sp = sp - 2;        // Increment stack pointer
    ip = ip + 1;        // Increment instruction pointer
  };



// Add the top two integers on the stack
  instruction iadd = {0000 0011 } {
    sp = sp + 2;                // Increment stack pointer
    let x = m_stack:2[sp];          // Pop first value into inp (top
value of the stack)

    sp = sp + 2;                // Increment stack pointer
    let y = m_stack:2[sp];         // Pop second value into outp (next
top value)

    let r = x + y;          // Perform addition

    m_stack:2[sp] = r;          // Push result back onto the stack
    sp = sp - 2;                    // Decrement stack pointer
    ip = ip + 1;                    // Increment instruction pointer
};
// Add the top two longs on the stack
 instruction ladd = { 0000 0100 } {
    sp = sp + 2;                    // Increment stack pointer
    let x = m_stack:2[sp];              // Pop first value into inp (top
value of the stack)

    sp = sp + 2;                    // Increment stack pointer
    let y = m_stack:2[sp];             // Pop second value into outp (next
top value)

    let r = x + y;              // Perform addition

    m_stack:2[sp] = r;            // Push result back onto the stack
    sp = sp - 2;                    // Decrement stack pointer
    ip = ip + 1;                    // Increment instruction pointer
};
// store the type and the value , maybe I should push the type and
value on the top of the stack
//must test store
instruction st = { 0000 0101, imm16 as ptr }
{
    sp=sp + 2;
    ram:2[ptr] = m_stack:2[sp];
    //ram:1[ptr+1] = from>>8;

    ip = ip + 3;
};
//instruction r_st = { 0000 1100,  reg as target}
//{
//  sp=sp + 2;
//  target = m_stack:2[sp];
//  ip = ip + 3;
//};
// I will use this to store the size of the array for array type
instruction a_st = { 0000 0110, imm16 as ptr }
{
    sp=sp + 2;
    ram:2[ptr+4] = m_stack:2[sp];// we save 8 bytes for each variable ,
store the type in the first 4 bytes (32 bits from the stack) and the
value in the next
    ip = ip + 3;
};
// to create load I have to have different loads (iload, lload..) I
also need a general funciton to dispatch to them, but I also need a to
recieve the type
// and a way to extract it, and a way to handle the labels, in short I
need to know how I figured out how to store it in metadata section
// were I keep all my variables and their types, or simple just how I
am storing so that I can load
  instruction jmp = { 0000 0111, imm16 as target } {
    ip = target;
  };
  instruction jz = { 0000 1000, imm16 as target } {
    sp = sp + 2;                // Increment stack pointer
    let x = m_stack:2[sp];          // Pop first value into inp (top
value of the stack)

    if x == 0x0 then
      ip = target;
    else
      ip = ip + 3;
  };
  // for now, everywhere I pushed the value and then pushed the type to
the stack
instruction wide_add ={0000 1001} {
    // Increment stack pointer to "pop" the first operand's pointer
    sp = sp + 2;
    let type1 = m_stack:2[sp];         // First operand type (address
of operand)
    sp = sp + 2;
    let value1 = m_stack:2[sp];         // Type of first operand (e.g.,
2 for number)

    // Increment stack pointer to "pop" the second operand's pointer
    sp = sp + 2;
    let type2 = m_stack:2[sp];         // Second operand pointer
    sp = sp + 2;
    let value2 = m_stack:2[sp];          // Type of second operand

    // Check if both operands are of type "number" (type == 2)
    if (type1 == 2) && (type2 == 2) then {

        let r = value1 + value2;                // Perform addition

        // Push the result back onto the stack
        m_stack:2[sp] = r;
        sp = sp - 2;                  // Decrement stack pointer
        m_stack:2[sp]= 2; // push the type
        sp= sp -2;
    } else {
        // Push error code (-1) onto the stack if types don't match
        m_stack:2[sp] = -1;
        sp = sp - 2;                  // Decrement stack pointer
        m_stack:2[sp] = -1;
        sp=sp -2; // push type error
    }

    // Increment instruction pointer to the next instruction
    ip = ip + 1;
};
instruction wide_store = { 0000 1010 }
{   sp=sp + 2;
    let ptr = m_stack:2[sp];// TO

    sp=sp + 2;
    let type=m_stack:2[sp];//pop type
    sp=sp + 2;
    let value=m_stack:2[sp];//pop value
    ram:2[ptr] = type;// store type
    ram:2[ptr+4] = value;// store value
    ip = ip + 1;
};
instruction wide_sub ={0000 1011} {
    // Increment stack pointer to "pop" the first operand's pointer
    sp = sp + 2;
    let type1 = m_stack:2[sp];         // First operand type (address
of operand)
    sp = sp + 2;
    let value1 = m_stack:2[sp];         // Type of first operand (e.g.,
2 for number)

    // Increment stack pointer to "pop" the second operand's pointer
    sp = sp + 2;
    let type2 = m_stack:2[sp];         // Second operand pointer
    sp = sp + 2;
    let value2 = m_stack:2[sp];           // Type of second operand

    // Check if both operands are of type "number" (type == 2)
    if (type1 == 2) && (type2 == 2) then {

        let r = value1 - value2;                 // Perform addition

        // Push the result back onto the stack
        m_stack:2[sp] = r;
        sp = sp - 2;                     // Decrement stack pointer
        m_stack:2[sp]= 2; // push the type
        sp= sp -2;
    } else {
        // Push error code (-1) onto the stack if types don't match
        m_stack:2[sp] = -1;
        sp = sp - 2;                     // Decrement stack pointer
        m_stack:2[sp] = -1;
        sp=sp -2; // push type error
    }

    // Increment instruction pointer to the next instruction

    ip = ip + 1;
};
instruction wide_mult ={0000 1100} {
    // Increment stack pointer to "pop" the first operand's pointer
    sp = sp + 2;
    let type1 = m_stack:2[sp];         // First operand type (address
of operand)
    sp = sp + 2;
    let value1 = m_stack:2[sp];         // Type of first operand (e.g.,
2 for number)

    // Increment stack pointer to "pop" the second operand's pointer
    sp = sp + 2;
    let type2 = m_stack:2[sp];         // Second operand pointer
    sp = sp + 2;
    let value2 = m_stack:2[sp];           // Type of second operand

    // Check if both operands are of type "number" (type == 2)
    if (type1 == 2) && (type2 == 2) then {

        let r = value1 * value2;                // Perform addition

        // Push the result back onto the stack
        m_stack:2[sp] = r;
        sp = sp - 2;                      // Decrement stack pointer
        m_stack:2[sp]= 2; // push the type
        sp= sp -2;
    } else {
        // Push error code (-1) onto the stack if types don't match
        m_stack:2[sp] = -1;
        sp = sp - 2;                      // Decrement stack pointer
        m_stack:2[sp] = -1;
        sp=sp -2; // push type error
    }

    // Increment instruction pointer to the next instruction
    ip = ip + 1;
};
instruction wide_div ={0000 1101} {
     // Increment stack pointer to "pop" the first operand's pointer
    sp = sp + 2;
    let type1 = m_stack:2[sp];         // First operand type (address
of operand)
    sp = sp + 2;
    let value1 = m_stack:2[sp];         // Type of first operand (e.g.,
2 for number)

    // Increment stack pointer to "pop" the second operand's pointer
    sp = sp + 2;
    let type2 = m_stack:2[sp];         // Second operand pointer
    sp = sp + 2;
    let value2 = m_stack:2[sp];          // Type of second operand

    // Check if both operands are of type "number" (type == 2)
    if (type1 == 2) && (type2 == 2) then {

        let r = value1 / value2;                // Perform addition

        // Push the result back onto the stack
        m_stack:2[sp] = r;
        sp = sp - 2;                      // Decrement stack pointer
        m_stack:2[sp]= 2; // push the type
        sp= sp -2;
    } else {
        // Push error code (-1) onto the stack if types don't match
        m_stack:2[sp] = -1;
        sp = sp - 2;                      // Decrement stack pointer
        m_stack:2[sp] = -1;
        sp=sp -2; // push type error
    }

    // Increment instruction pointer to the next instruction
    ip = ip + 1;
};
  instruction jgt = { 0000 1111, imm16 as target } {
    sp = sp + 2;                      // Increment stack pointer
    let x = m_stack:2[sp];            // Pop first value into inp (top 
value of the stack)

    if (x >> 15 == 0x0) && (x != 0x0) then
      ip = target;
    else
      ip = ip + 3;
  };
    instruction jge = { 0001 0000, imm16 as target } {
    sp = sp + 2;                      // Increment stack pointer
    let x = m_stack:2[sp];            // Pop first value into inp (top 
value of the stack)

    if (x >> 15 == 0x0)  then
      ip = target;
    else
      ip = ip + 3;
  };

      instruction jlt = { 0001 0001, imm16 as target } {
    sp = sp + 2;                      // Increment stack pointer
    let x = m_stack:2[sp];            // Pop first value into inp (top 
value of the stack)

    if (x >> 15 == 0x1)  then
      ip = target;
    else
      ip = ip + 3;
  };

        instruction jle = { 0001 0010, imm16 as target } {
    sp = sp + 2;                      // Increment stack pointer
    let x = m_stack:2[sp];            // Pop first value into inp (top 
value of the stack)

    if (x >> 15 == 0x1) || (x == 0x0) then
      ip = target;
    else
      ip = ip + 3;
  };
  // Halt execution
  instruction hlt = { 1111 1111 } {
    // Execution halts
  };

    instruction write = { 0001 0011 } {
    sp = sp +2 ;
    let x = m_stack:2[sp];            // Pop first value into inp (top 
value of the stack)

    outp = x; // Retrieve value from stack to standard output
    ip = ip + 1;          // Increment instruction pointer
  };
    instruction read = { 0001 0100  } {
    sp = sp + 2 ;
    let ptr= m_stack:2[sp];
    ram:2[ptr+4] = inp;
    ip = ip + 1;    // Increment instruction pointer
  };
    instruction jmp_emp = { 0001 0101, imm16 as ptr } {
    sp = sp + 2;                // Increment stack pointer
    let target = ram:2[ptr];
    ip = target;
  };
  instruction wide_mod ={0001 0110} {
     sp = sp + 2;
    let type1 = m_stack:2[sp];         // First operand type (address
of operand)
    sp = sp + 2;
    let value1 = m_stack:2[sp];         // Type of first operand (e.g.,
2 for number)

    // Increment stack pointer to "pop" the second operand's pointer
    sp = sp + 2;
    let type2 = m_stack:2[sp];         // Second operand pointer
    sp = sp + 2;
    let value2 = m_stack:2[sp];           // Type of second operand

    // Check if both operands are of type "number" (type == 2)
    if (type1 == 2) && (type2 == 2) then {

        let r = value1 % value2;                // Perform addition

        // Push the result back onto the stack
        m_stack:2[sp] = r;
        sp = sp - 2;                // Decrement stack pointer
        m_stack:2[sp]= 2; // push the type
        sp= sp -2;
    } else {
        // Push error code (-1) onto the stack if types don't match
        m_stack:2[sp] = -1;
        sp = sp - 2;                // Decrement stack pointer
        m_stack:2[sp] = -1;
        sp=sp -2; // push type error
    }

    // Increment instruction pointer to the next instruction
    ip = ip + 1;
};

  instruction push_sf ={0001 0111 , imm16 as value} {
    ram[esp]= value;
    esp = esp - 4;   // Decrement stack pointer
    ip = ip + 3;    // Increment instruction pointer
  };
  instruction pop_sf ={0001 1000} {
    esp = esp + 4;         // Increment stack pointer

    ip = ip + 1;        // Increment instruction pointer
};

  instruction call = { 0001 1010, imm16 as ptr, 0000 0000 } {
    esp = esp + 4;
    ram:2[esp] = ip + 4; // save return point
    esp=esp + 4;
    ram:2[esp]= ebp; // save the old ebp


    ebp = esp;
    ip = ptr;
  };
  instruction ret = { 0001 1011  } {
        //if ebp != 0 then {
            esp = ebp;
            ebp = ram:2[esp] ;
            esp = esp - 4;
            ip = ram:2[esp];
            esp = esp - 4;
            //  }
        //ip = ip + 4;
    //  ip=ip + 1;
};
  instruction load = { 0001 1100 } {
        sp= sp + 2; // pop the pointer to the value in ram
        let ptr= m_stack:2[sp];
        let type = ram:2[ptr];
        let value = ram:2[ptr + 4];
        m_stack[sp]= value;
        sp =sp -2; //push the value
        m_stack[sp]= type;
        sp =sp -2; //push the type
        ip=ip+1;
};
instruction add_reg_imm = { 0001 1101, imm16 as value } {
        if value ==1 then {
        m_stack:2[sp] = ebp + 4 ;// the size of the cell is 2 bytes (16
bits) and I need 4 for value and 4 for type
        }
        else {
        m_stack:2[sp] = ebp +8 *(value - 1) +4 ;
        }
        sp = sp - 2;   // Decrement stack pointer
        ip=ip+3;
};

instruction sub_reg_imm = { 0001 1110, imm16 as value } {
        if value ==1 then {
        m_stack:2[sp] = ebp -4 * value +8;
        }
        else {
        m_stack:2[sp] = ebp - (8 * (value - 1)) + 4;
        }
        sp = sp - 2;   // Decrement stack pointer

        ip=ip+3;
};
instruction add_mem_imm = { 0010 0011, imm16 as value } {
        sp =sp + 2;
        let x= m_stack:2[sp]; // our pointer
        sp =sp + 2;
        let z= m_stack:2[sp];// filler value
        let y = x + value*8  ;

        m_stack[sp]= y;
        sp = sp - 2;
        ip = ip + 3;

};

instruction init ={ 0001 1111,imm16 as target }{
    ebp=target;
    esp=ebp;
    heap_base=0x7A00;
    ip=ip + 3;
};
  instruction pop_hf ={0010 0000} {
    heap_base = heap_base + 4;         // Increment stack pointer

    ip = ip + 1;         // Increment instruction pointer
};
  instruction pop_vtable ={0010 0001, imm16 as value} {

    ram:2[heap_base]=value;
    heap_base = heap_base + 4;         // Increment stack pointer

    ip = ip + 3;         // Increment instruction pointer
};

 instruction push_hf = { 0010 0010} {
    heap_base = heap_base + 4;
    m_stack:2[sp] = heap_base;
    sp = sp - 2;    // Decrement stack pointer
    ip = ip + 1;    // Increment instruction pointer
  };

  instruction call_from_stack = { 0010 0100 } {

    let ptr= m_stack:2[sp]; // our pointer
    esp = esp + 4;
    ram:2[esp] = ip + 1; // save return point
    esp=esp + 4;
    ram:2[esp]= ebp; // save the old ebp


    ebp = esp;
    ip = ptr;
  };
    instruction check_vtable = { 0010 0101 } {
        sp=sp + 2;
        let type=m_stack:2[sp];//pop type
        sp=sp + 2;
        let func = m_stack:2[sp];//pop value

        sp= sp + 2; // pop the pointer to the value in ram
        let ptr= m_stack:2[sp];
        let y = ram:2[ptr];
        ptr = ptr + 4;
        let found = 0;
        while y > 0 do{
            ptr = ptr + 8;
            let value = ram:2[ptr];
            if value == func then
            {
            found = 1;
            m_stack[sp]= ram:2[ptr - 4];
            sp = sp - 2;// after this I need some for loop to check all
the funcs
            ip = ip + 1;
            break;
            }
            y = y - 1;
            }
        if found == 0 then {
            m_stack[sp]= 0;
            sp = sp - 2;// after this I need some for loop to check all
the funcs

        }
        ip = ip + 1;

  };
mnemonics:
  mnemonic init for init(target) "{1}";
  mnemonic push (value) "{1}";
  mnemonic pop() ;
  mnemonic popOut();
  mnemonic iadd();
  mnemonic ladd() ;
  mnemonic jump for jmp(target) "{1}";
  mnemonic jz(target) "{1}";
  mnemonic jgt(target) "{1}";
  mnemonic jge(target) "{1}";
  mnemonic jlt(target) "{1}";
  mnemonic jle(target) "{1}";
  mnemonic add_mem_imm(value) "{1}";
  mnemonic store for st(ptr) "{1}";
  mnemonic astore for a_st(ptr) "{1}";
  mnemonic wide_add();
  mnemonic wide_store();
  mnemonic wide_sub();
  mnemonic wide_mult();
  mnemonic wide_div();
  mnemonic wide_mod();
  mnemonic read();
  mnemonic write();
  mnemonic jmp_emp(ptr) "{1}";
  mnemonic check_vtable () ;
  mnemonic push_sf (value) "{1}" ;
  mnemonic pop_vtable (value) "{1}" ;
  mnemonic pop_sf () ;
  mnemonic push_hf () ;
  mnemonic pop_hf () ;
  mnemonic call ( ptr) "{1}" ;
  mnemonic ret() ;
  mnemonic call_from_stack() ;
  mnemonic load() ;
  mnemonic add_s for add_reg_imm(value) "{1}" ;
  mnemonic sub_s for sub_reg_imm(value) "{1}" ;



  mnemonic hlt();
}
```

Architecture.pdsl

## 5.1 Dynamic type casting

As an example of the work done in this lab I created an input file in my language that has the following features

1- A generic class Example
2- A generic class ExampleClass which inherits Example
3- Example has functions : DoThing, DoOtherThing
4- ExampleClass overrides Dothing, does not override DoOtherThing and also introduces its own function ShowMessage
5- Function M(o, n) which checks for correct type casting
6- Function error which is responsible for handling errors in type casting
7- Function printf which prints to console
8- Function helper which calls M on an object of type Example
9- Function helper2 which calls M on an object of type ExampleClass
10- Function helper3 which calls M on an object from type Example casted into an type ExampleClass
11- Main function which for each helper function checks M on classes with global offsets from 0 to 7 and prints out the results

```
class ExampleClass (X, Y) extends Example (int , int)


    public z as int

    public function ShowMessage() as int

       printf(3);
    end function
    public function DoThing() as int
        printf(1);
    end function
end class
class Example (A, B)

    public x as A
    public y as B
    public function DoThing() as int
       printf(4);

    end function
     public function DoOtherThing() as int
       printf(5);

    end function


end class
function helper(n as int)
  dim ex as Example <int, int>
  dim ex_cl as ExampleClass <int, int>
  dim x as int
  M(ex, n);
  print(10);
  x;
end function
function helper2(n as int)
  dim ex as Example <int, int>
  dim ex_cl as ExampleClass <int, int>
  dim x as int
  M(ex_cl, n);
  print(10);
  x;
end function
function helper3(n as int)
  dim ex as Example <int, int>
  dim ex_cl as ExampleClass <int, int>
  dim x as int
  ex_cl = ex;
  M(ex_cl, n);
  print(10);
  x;
end function
function fib(n as int)
  dim ex as Example <int, int>
  dim ex_cl as ExampleClass <int, int>
  dim x as int
  ex.x = 5;
  x = ex_cl.ShowMessage();
  x = ex.x;
  M(ex, n);
  x;
end function
function M(O, n as int)
 printf(7);
end function
function error()
    print(109);
    print(101);
    print(116);
    print(104);
    print(111);
    print(100);
    print(32);
    print(100);
    print(111);
```
Санкт-Петербург, 2025 г.

```
    print(101);
    print(115);
    print(32);
    print(110);
    print(111);
    print(116);
    print(32);
    print(101);
    print(120);
    print(105);
    print(115);
    print(116);
    print(32);
    print(105);
    print(110);
    print(32);
    print(116);
    print(104);
    print(105);
    print(115);
    print(32);
    print(99);
    print(108);
    print(97);
    print(115);
    print(115);
    print(10);
end function

function printf(res as int)
    dim nextLine, revertedNum, tmp as int
    nextLine = 10;
    revertedNum = 0;
    while res != 0
        revertedNum = revertedNum * 10   ;
        tmp = res % 10;
        revertedNum= revertedNum + tmp;
        res = res / 10;
    wend
    while revertedNum != 0
        tmp = revertedNum % 10;
        tmp=tmp + 48;
        print(tmp);
        revertedNum = revertedNum / 10;
    wend
    print(10);
end function

function main()
  dim i as int
  i = 0;
 while i < 7
     helper(i);
     i = i + 1;
   wend
     i = 0;
 while i < 7
```

```
            helper2(i);
            i = i + 1;
        wend   i = 0;
    while i < 7
            helper3(i);
            i = i + 1;
        wend

    end function

```

Input.txt

As a result of executing the translation module we get the following listing.

```
[section ram, code]
        init CodeEnd
        jump main
vtable_ExampleClass:
  dq 3
  dd ExampleClass_DoThing
  dd 1
  dd Example_DoOtherThing
  dd 2
  dd ExampleClass_ShowMessage
  dd 3
vtable_Example:
  dq 2
  dd Example_DoThing
  dd 4
  dd Example_DoOtherThing
  dd 5
ExampleClass_ShowMessage:
        push 3
        push 2
        pop_sf
        pop_sf
        call printf
        ret
ExampleClass_DoThing:
        push 1
        push 2
        pop_sf
        pop_sf
        call printf
        ret
Example_DoThing:
        push 4
        push 2
        pop_sf
        pop_sf
        call printf
        ret
Example_DoOtherThing:
        push 5
        push 2
        pop_sf
        pop_sf
        call printf
        ret
helper:
        sub_s 1
        wide_store
        pop_sf
        pop_sf
        pop_sf
        pop_sf
        pop_sf
        pop_sf
        push_hf
        pop_vtable vtable_Example
        pop_hf
        pop_hf
        pop_hf
        pop_hf
        add_s 1
        wide_store
        push_hf
        pop_vtable vtable_ExampleClass
        pop_hf
        pop_hf
        add_s 2
        wide_store
        push 0
        push 2
        add_s 3
        wide_store
        add_s 1
        load
        load
        sub_s 1
        load
        check_vtable
        jz label_8
        jump label_9
        ret
label_8:
        call error
        jump label_10
label_9:
		call_from_stack
		jump label_10
	label_10:
		push 10
		push 2
		pop
		write
		add_s 3
		load
		ret
	helper2:
		sub_s 1
		wide_store
		pop_sf
		pop_sf
		pop_sf
		pop_sf
		pop_sf
		pop_sf
		push_hf
		pop_vtable vtable_Example
		pop_hf
		pop_hf
		pop_hf
		pop_hf
		add_s 1
		wide_store
		push_hf
		pop_vtable vtable_ExampleClass
		pop_hf
		pop_hf
		add_s 2
		wide_store
		push 0
		push 2
		add_s 3
		wide_store
		add_s 2
		load
		load
		sub_s 1
		load
		check_vtable
		jz label_12
		jump label_13
		ret
	label_12:
		call error
```

```
        jump label_14
label_13:
        call_from_stack
        jump label_14
label_14:
        push 10
        push 2
        pop
        write
        add_s 3
        load
        ret
helper3:
        sub_s 1
        wide_store
        pop_sf
        pop_sf
        pop_sf
        pop_sf
        pop_sf
        pop_sf
        push_hf
        pop_vtable vtable_Example
        pop_hf
        pop_hf
        pop_hf
        pop_hf
        add_s 1
        wide_store
        push_hf
        pop_vtable vtable_ExampleClass
        pop_hf
        pop_hf
        add_s 2
        wide_store
        push 0
        push 2
        add_s 3
        wide_store
        add_s 1
        load
        add_s 2
        wide_store
        add_s 2
        load
        load
        sub_s 1
        load
```

```
            check_vtable
            jz label_16
            jump label_17
            ret
    label_16:
            call error
            jump label_18
    label_17:
            call_from_stack
            jump label_18
    label_18:
            push 10
            push 2
            pop
            write
            add_s 3
            load
            ret
    fib:
            sub_s 1
            wide_store
            pop_sf
            pop_sf
            pop_sf
            pop_sf
            pop_sf
            pop_sf
            push_hf
            pop_vtable vtable_Example
            pop_hf
            pop_hf
            pop_hf
            pop_hf
            add_s 1
            wide_store
            push_hf
            pop_vtable vtable_ExampleClass
            pop_hf
            pop_hf
            add_s 2
            wide_store
            push 0
            push 2
            add_s 3
            wide_store
            push 5
            push 2
            add_s 1
            load
            add_mem_imm 1
            wide_store
            add_s 2
            load
            load
            add_mem_imm 3
            load
            pop
            call_from_stack
            wide_store
            add_s 1
            load
            add_mem_imm 1
            load
            add_s 3
            wide_store
            add_s 1
            load
            load
            sub_s 1
            load
            check_vtable
            jz label_21
            jump label_22
            ret
label_21:
            call error
            jump label_23
label_22:
            call_from_stack
            jump label_23
label_23:
            add_s 3
            load
            ret
M:
            sub_s 1
            wide_store
            sub_s 2
            wide_store
            push 7
            push 2
            pop_sf
            pop_sf
            call printf
            ret
error:
```

```
push 109
push 2
pop
write
push 101
push 2
pop
write
push 116
push 2
pop
write
push 104
push 2
pop
write
push 111
push 2
pop
write
push 100
push 2
pop
write
push 32
push 2
pop
write
push 100
push 2
pop
write
push 111
push 2
pop
write
push 101
push 2
pop
write
push 115
push 2
pop
write
push 32
push 2
pop
write
```

```
                push 110
                push 2
                pop
                write
                push 111
                push 2
                pop
                write
                push 116
                push 2
                pop
                write
                push 32
                push 2
                pop
                write
                push 101
                push 2
                pop
                write
                push 120
                push 2
                pop
                write
                push 105
                push 2
                pop
                write
                push 115
                push 2
                pop
                write
                push 116
                push 2
                pop
                write
                push 32
                push 2
                pop
                write
                push 105
                push 2
                pop
                write
                push 110
                push 2
                pop
                write
```

```
push 32
push 2
pop
write
push 116
push 2
pop
write
push 104
push 2
pop
write
push 105
push 2
pop
write
push 115
push 2
pop
write
push 32
push 2
pop
write
push 99
push 2
pop
write
push 108
push 2
pop
write
push 97
push 2
pop
write
push 115
push 2
pop
write
push 115
push 2
pop
write
push 10
push 2
pop
write
```

```
        ret
printf:
        sub_s 1
        wide_store
        pop_sf
        pop_sf
        pop_sf
        pop_sf
        pop_sf
        pop_sf
        push 0
        push 2
        add_s 1
        wide_store
        push 0
        push 2
        add_s 2
        wide_store
        push 0
        push 2
        add_s 3
        wide_store
        push 10
        push 2
        add_s 1
        wide_store
        push 0
        push 2
        add_s 2
        wide_store
                ;while
label_64:
        push 0
        push 2
        sub_s 1
        load
        wide_sub
        pop
        jz label_66
        push 1
                ;false branch
        jump label_67
label_66:
        push 0
                ;true branch
label_67:
        jz label_68
                ;while body
```

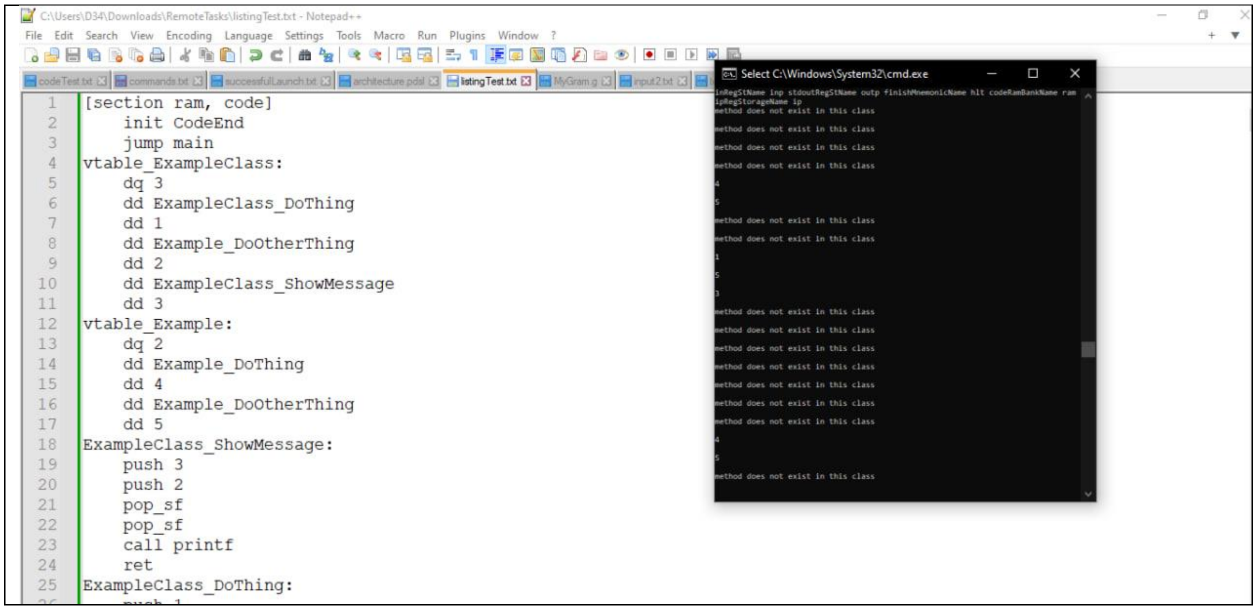
```
            push 10
            push 2
            add_s 2
            load
            wide_mult
            add_s 2
            wide_store
            push 10
            push 2
            sub_s 1
            load
            wide_mod
            add_s 3
            wide_store
            add_s 3
            load
            add_s 2
            load
            wide_add
            add_s 2
            wide_store
            push 10
            push 2
            sub_s 1
            load
            wide_div
            sub_s 1
            wide_store
            jump label_64
    label_68:
                    ;end while
                    ;while
    label_72:
            push 0
            push 2
            add_s 2
            load
            wide_sub
            pop
            jz label_74
            push 1
                    ;false branch
            jump label_75
    label_74:
            push 0
                    ;true branch
    label_75:
            jz label_76
```

```
                        ;while body
                push 10
                push 2
                add_s 2
                load
                wide_mod
                add_s 3
                wide_store
                push 48
                push 2
                add_s 3
                load
                wide_add
                add_s 3
                wide_store
                add_s 3
                load
                pop
                write
                push 10
                push 2
                add_s 2
                load
                wide_div
                add_s 2
                wide_store
                jump label_72
        label_76:
                        ;end while
                push 10
                push 2
                pop
                write
                ret
        main:
                pop_sf
                pop_sf
                push 0
                push 2
                add_s 1
                wide_store
                push 0
                push 2
                add_s 1
                wide_store
                        ;while
        label_82:
                push 7
                push 2
                add_s 1
                load
                wide_sub
                pop
                jlt label_84
                push 0
                                ;false branch
                jump label_85
label_84:
                push 1
                                ;true branch
label_85:
                jz label_86
                                ;while body
                add_s 1
                load
                pop_sf
                pop_sf
                call helper
                push 1
                push 2
                add_s 1
                load
                wide_add
                add_s 1
                wide_store
                jump label_82
label_86:
                                ;end while
                push 0
                push 2
                add_s 1
                wide_store
                                ;while
label_90:
                push 7
                push 2
                add_s 1
                load
                wide_sub
                pop
                jlt label_92
                push 0
                                ;false branch
                jump label_93
label_92:
                push 1
```

```
                        ;true branch
    label_93:
            jz label_94
                        ;while body
            add_s 1
            load
            pop_sf
            pop_sf
            call helper2
            push 1
            push 2
            add_s 1
            load
            wide_add
            add_s 1
            wide_store
            jump label_90
    label_94:
                        ;end while
            push 0
            push 2
            add_s 1
            wide_store
                        ;while
    label_98:
            push 7
            push 2
            add_s 1
            load
            wide_sub
            pop
            jlt label_100
            push 0
                        ;false branch
            jump label_101
    label_100:
            push 1
                        ;true branch
    label_101:
            jz label_102
                        ;while body
            add_s 1
            load
            pop_sf
            pop_sf
            call helper3
            push 1
            push 2
            add_s 1
            load
            wide_add
            add_s 1
            wide_store
            jump label_98
    label_102:
                    ;end while
            jump halt
    halt:
            hlt
    CodeEnd:
```

listingTest.txt

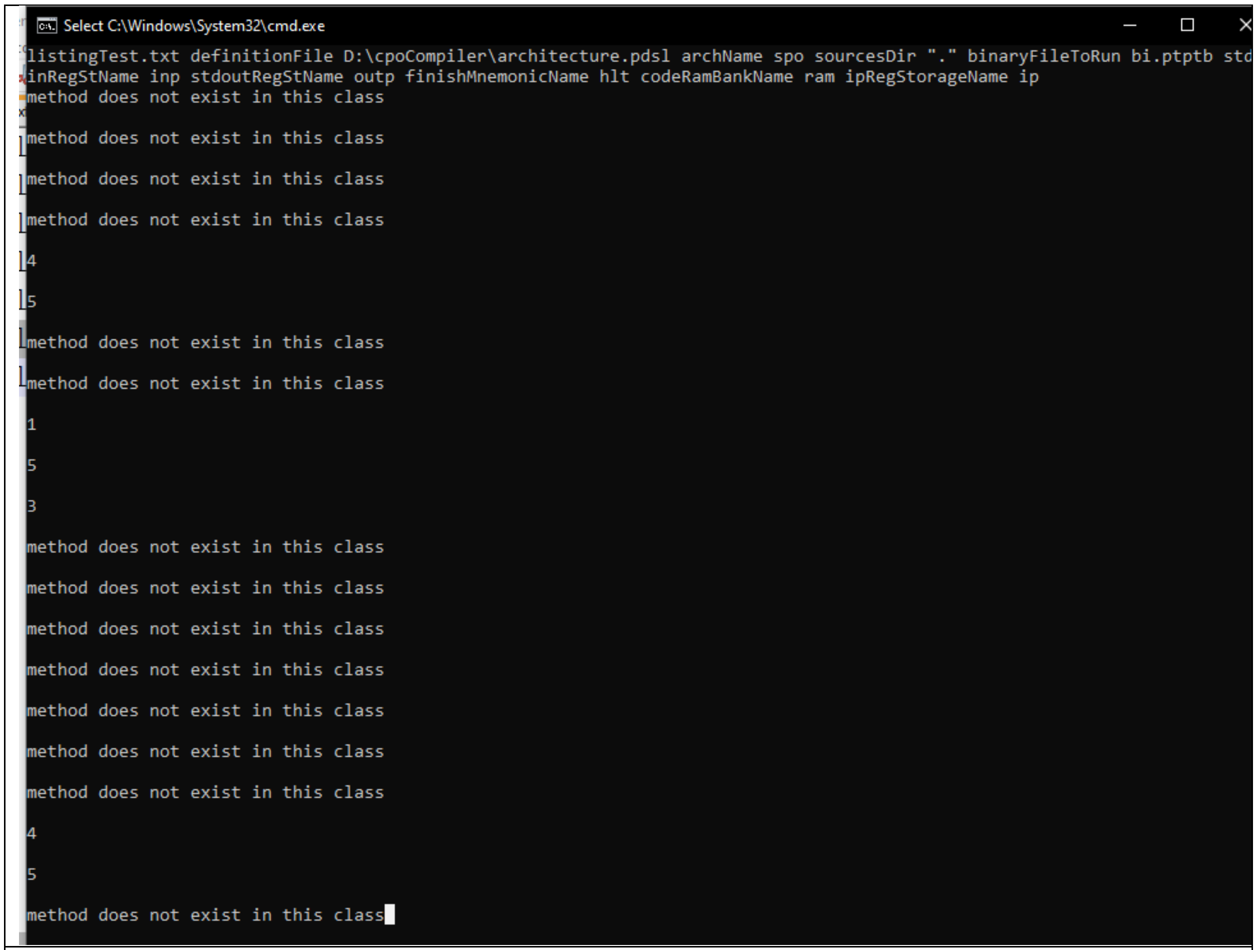
And as a result of assembling and executing this listing according to our architecture we get the following results


C:\Users\D34\Downloads\RemoteTasks\listingTest.txt - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
codeTest.txt
commands.txt
successfulLaunch.txt
architecture.pdsl
listingTest.txt
MyGram.g
input2.txt
Select C:\Windows\System32\cmd.exe

```
[section ram, code]
    init CodeEnd
    jump main
vtable_ExampleClass:
    dq 3
    dd ExampleClass_DoThing
    dd 1
    dd Example_DoOtherThing
    dd 2
    dd ExampleClass_ShowMessage
    dd 3
vtable_Example:
    dq 2
    dd Example_DoThing
    dd 4
    dd Example_DoOtherThing
    dd 5
ExampleClass_ShowMessage:
    push 3
    push 2
    pop_sf
    pop_sf
    call printf
    ret
ExampleClass_DoThing:
```

```
inRegStName inp stdoutRegStName outp finishMnemonicName hlt codeRamBankName ram
ipRegStorageName ip
method does not exist in this class

method does not exist in this class

method does not exist in this class

method does not exist in this class

4

5

method does not exist in this class

method does not exist in this class

1

5

3

method does not exist in this class

method does not exist in this class

method does not exist in this class

method does not exist in this class

method does not exist in this class

method does not exist in this class

method does not exist in this class

4

5

method does not exist in this class
```

Санкт-Петербург, 2025 г.



```
Select C:\Windows\System32\cmd.exe
listingTest.txt definitionFile D:\cpoCompiler\architecture.pdsl archName spo sourcesDir "." binaryFileToRun bi.ptptb std
inRegStName inp stdoutRegStName outp finishMnemonicName hlt codeRamBankName ram ipRegStorageName ip
method does not exist in this class

method does not exist in this class

method does not exist in this class

method does not exist in this class

4

5

method does not exist in this class

method does not exist in this class

1

5

3

method does not exist in this class

method does not exist in this class

method does not exist in this class

method does not exist in this class

method does not exist in this class

method does not exist in this class

method does not exist in this class

4

5

method does not exist in this class
```

Execution output in console

As we can see the function DoThing inside ExampleClass got the offset 1, DoOtherThing got the offset 5 globally and since it does not exist inside ExampleClass it also got the offset 5 locally, while the function showMessage got the offset 3. Inside Example, the function DoThing has the offset 4, while DoOtherThing has the offset 5. Upon calling the checker function M on the object with the offsets from 0 to 7, we were able to call the functions number 4 and 5 from inside an object with the type example (for clarification I made the functions print out their offset upon call), while an object of type Example class printed out 1, 5, 3. Finally an object which is casted into the type Example printed out 4 and 5 as well. All other offsets printed out the error message “method does not exist in this class”.

## 6. Conclusions

In summary, this laboratory work set out to enrich the existing software package with full-fledged support for user-defined types, inheritance, overriding, and templating within a dynamically typed, stack-based machine architecture. Each module—from the parser and control-flow graph builder to the code generator, metadata emitter, and console inspector—was carefully extended: new data structures now encode type hierarchies and members; the grammar accepts the added constructs; operation trees model field/method access; linear code emits the requisite instruction sequences; and the inspector transparently renders nested field values. The accompanying report documents these structural changes, showcases representative source programs and their assembly listings, and demonstrates correct runtime behavior. Taken together, these results confirm that every objective outlined in the assignment has been met, and I believe I have successfully implemented all required steps.

During the process of completing this lab assignment I learned what a virtual table is, and I got a deep understanding of templating and generic types that I did not previously have. I learned what happens under the hood when a custom or a generic type is defined, when a casting happens and when a user defines a type and accesses its members.

## 7. Link to the project on gitlab

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

Санкт-Петербург, 2025 г.