Programming Lab Project Report

Prof. Dr. Peter Faber

Rashed Al-Lahaseh – 00821573

Table of Contents

Task

Introduce a different type (suggested: void type) that can be used at least for function declarations and definitions (for internal functions, i.e., functions defined in the compilation unit to be processed, as well as for external functions, i.e., functions that are only declared in the compilation unit to be processed)
If an internal function is declared void, a return statement within the body of this declaration should not take any expression
Note that this may also mean that you will have to declare a return type of void for LLVM; you can obtain an LLVM type for void in a similar way as for the int32 type using the function llvm::Type *llvm::Type::getVoidTy(LLVMContext &context)
This project is appropriate for a single person (or max 2)

Overview

As we all know, the output of a function that usually returns but does not pass the result value to the caller is called a Void Type. Often, certain tasks are named for their side effects, including running tasks or writing their performance parameters.

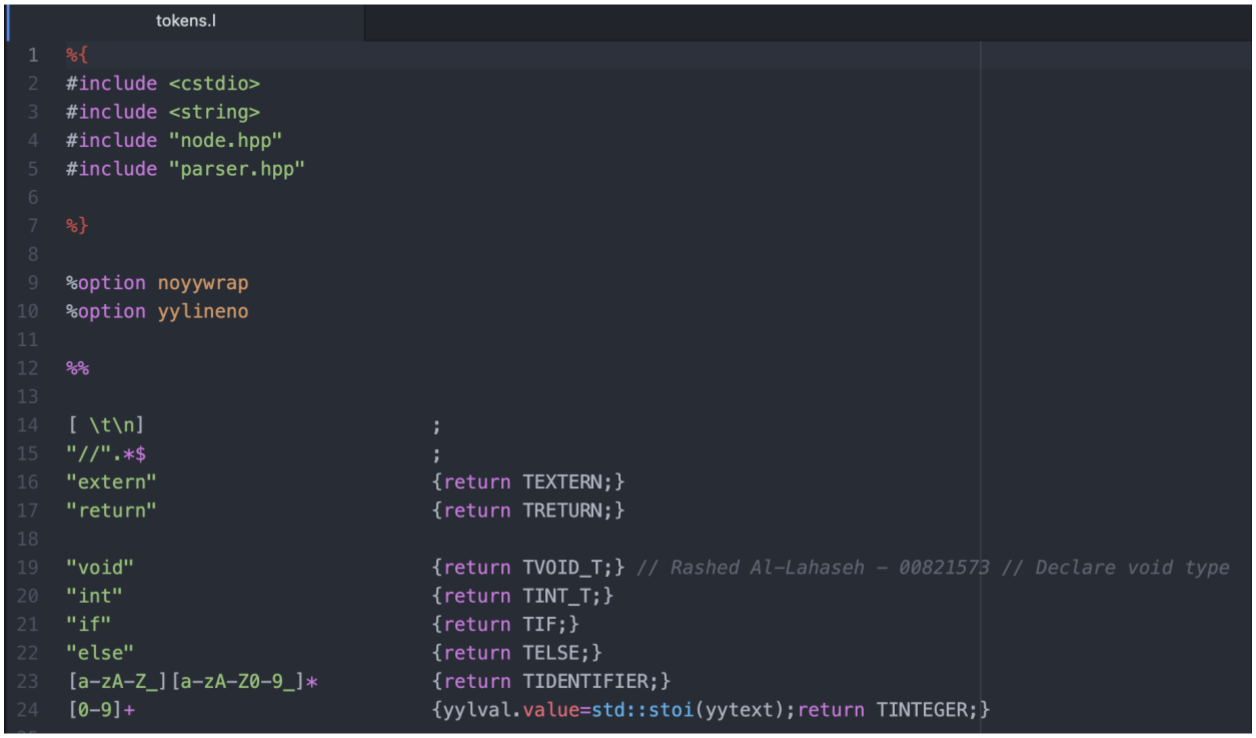
And since we already have int type declared we need to add also the void type in a way we do not replace it where both cases shall be working fine, where we will be need to know each function type call for each implemented method to handle it.

GitLab Repository

Implementation Strategy

Lexical Analysis with Flex

Therefore, I assigned a symbolic name to the token. In our grammar, this (TVOID_T) symbol becomes the “void”. We just put it back, but never described it. Then, we will get the grammatical definition of bison. Bison will create the parser.hpp file we included, and all tokens contained in this file will be generated and available for use.



tokens.l

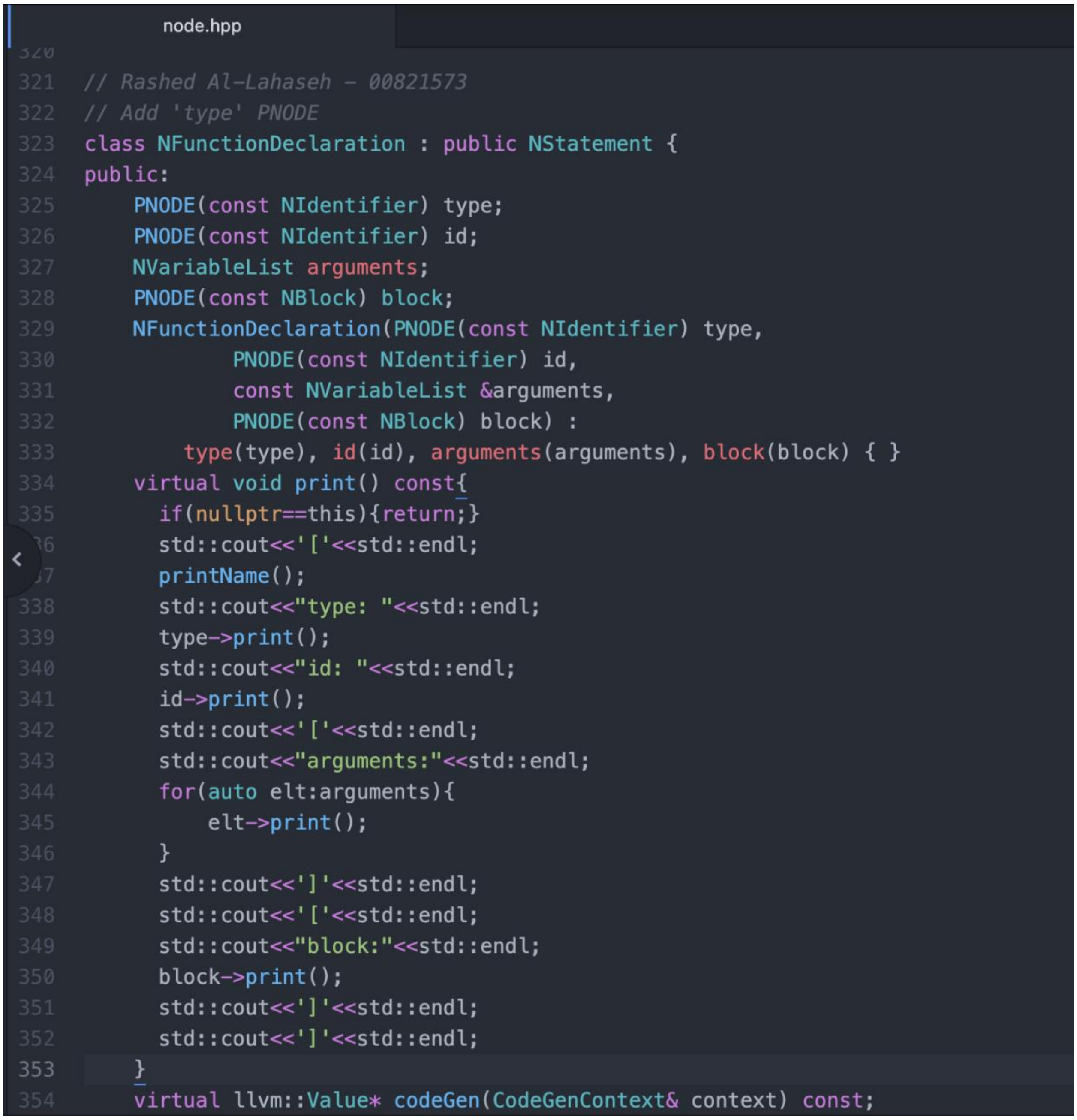
```
%{
#include <cstdio>
#include <string>
#include "node.hpp"
#include "parser.hpp"

%}

%option noyywrap
%option yylineno

%%

[ \t\n]                         ;
"//".*$                         ;
"extern"                        {return TEXTERN;}
"return"                        {return TRETURN;}

"void"                          {return TVOID_T;} // Rashed Al-Lahaseh - 00821573 // Declare void type
"int"                           {return TINT_T;}
"if"                            {return TIF;}
"else"                          {return TELSE;}
[a-zA-Z_][a-zA-Z0-9_]*          {return TIDENTIFIER;}
[0-9]+                          {yylval.value=std::stoi(yytext);return TINTEGER;}
```

Figure 1: token.l file

Using Bison for Semantic Parsing

Definition

Our AST is what our language expresses in memory, just as a string like “void x” represents our language in text form (before putting them together).
This process is very simple, because we are essentially building a structure for every semantic that our language can convey. They are all candidates for AST nodes: function calls, method declarations, vector declarations and references.



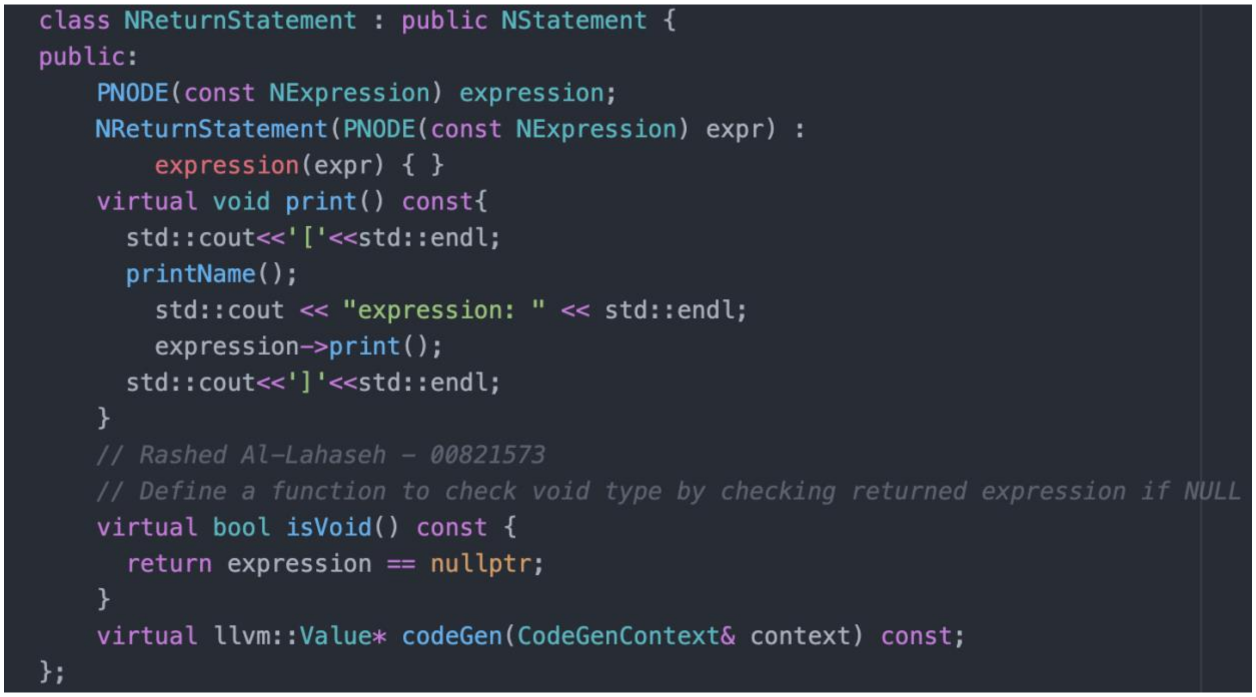
node.hpp

```
// Rashed Al-Lahaseh - 00821573
// Add 'type' PNODE
class NFunctionDeclaration : public NStatement {
public:
    PNODE(const NIdentifier) type;
    PNODE(const NIdentifier) id;
    NVariableList arguments;
    PNODE(const NBlock) block;
    NFunctionDeclaration(PNODE(const NIdentifier) type,
            PNODE(const NIdentifier) id,
            const NVariableList &arguments,
            PNODE(const NBlock) block) :
        type(type), id(id), arguments(arguments), block(block) { }
    virtual void print() const{
      if(nullptr==this){return;}
      std::cout<<'['<<std::endl;
      printName();
      std::cout<<"type: "<<std::endl;
      type->print();
      std::cout<<"id: "<<std::endl;
      id->print();
      std::cout<<'['<<std::endl;
      std::cout<<"arguments:"<<std::endl;
      for(auto elt:arguments){
          elt->print();
      }
      std::cout<<']'<<std::endl;
      std::cout<<'['<<std::endl;
      std::cout<<"block:"<<std::endl;
      block->print();
      std::cout<<']'<<std::endl;
      std::cout<<']'<<std::endl;
    }
    virtual llvm::Value* codeGen(CodeGenContext& context) const;
```

Figure 2: node.hpp file

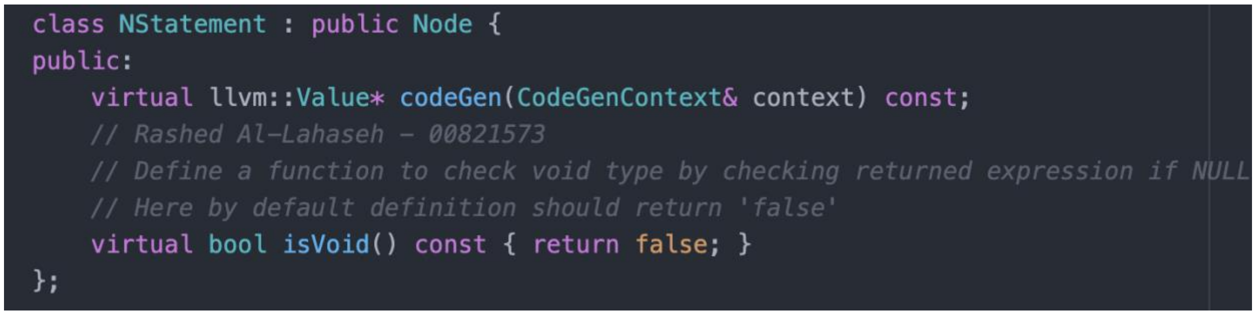
Validation

Now to cover error detection case, where need to validate if the statement neither the block of the function does not return empty statement incase we are defining an int type function or return statement in void function type.



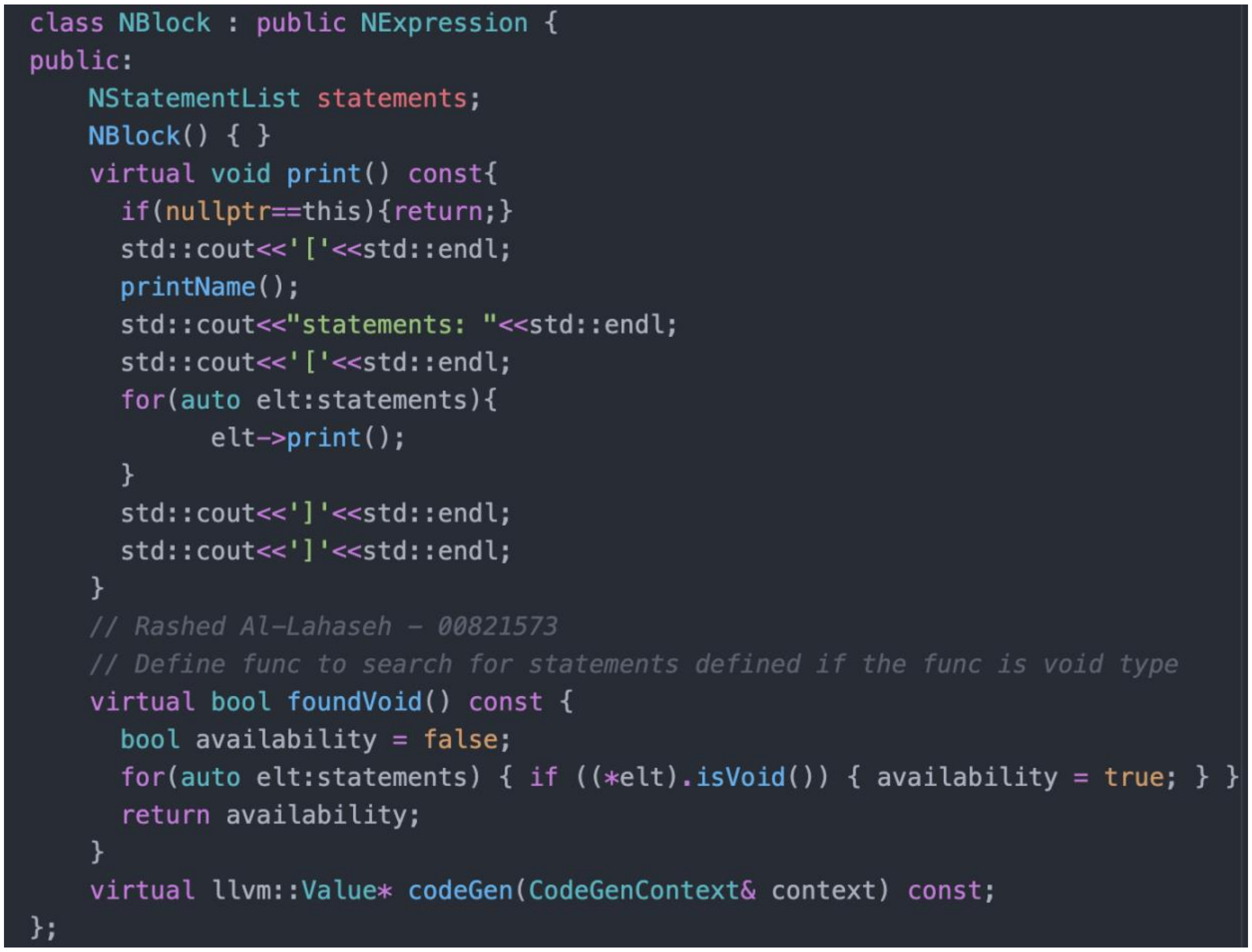
```
class NReturnStatement : public NStatement {
public:
    PNODE(const NExpression) expression;
    NReturnStatement(PNODE(const NExpression) expr) :
        expression(expr) { }
    virtual void print() const{
      std::cout<<'['<<std::endl;
      printName();
        std::cout << "expression: " << std::endl;
        expression->print();
      std::cout<<']'<<std::endl;
    }
    // Rashed Al-Lahaseh - 00821573
    // Define a function to check void type by checking returned expression if NULL
    virtual bool isVoid() const {
      return expression == nullptr;
    }
    virtual llvm::Value* codeGen(CodeGenContext& context) const;
};
```

Figure 3: node.hpp file, covering validating void type expressions



```
class NStatement : public Node {
public:
    virtual llvm::Value* codeGen(CodeGenContext& context) const;
    // Rashed Al-Lahaseh - 00821573
    // Define a function to check void type by checking returned expression if NULL
    // Here by default definition should return 'false'
    virtual bool isVoid() const { return false; }
};
```

Figure 4: node.hpp file, covering statments case

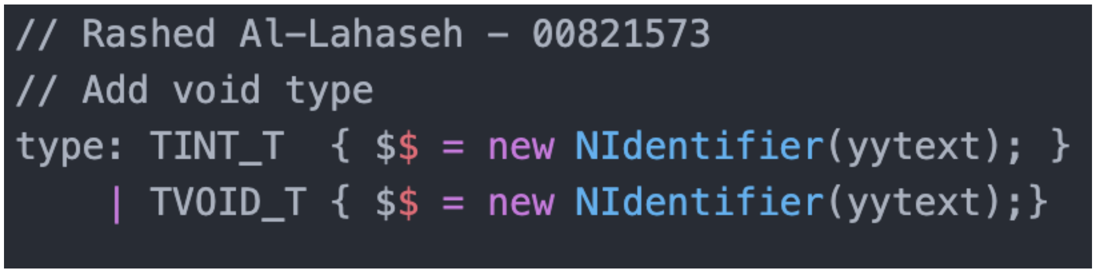


```
class NBlock : public NExpression {
public:
    NStatementList statements;
    NBlock() { }
    virtual void print() const{
      if(nullptr==this){return;}
      std::cout<<'['<<std::endl;
      printName();
      std::cout<<"statements: "<<std::endl;
      std::cout<<'['<<std::endl;
      for(auto elt:statements){
            elt->print();
      }
      std::cout<<']'<<std::endl;
      std::cout<<']'<<std::endl;
    }
    // Rashed Al-Lahaseh - 00821573
    // Define func to search for statements defined if the func is void type
    virtual bool foundVoid() const {
      bool availability = false;
      for(auto elt:statements) { if ((*elt).isVoid()) { availability = true; } }
      return availability;
    }
    virtual llvm::Value* codeGen(CodeGenContext& context) const;
};
```

Figure 5: node.hpp file, defining functionality to search for void statments

Defining Grammars

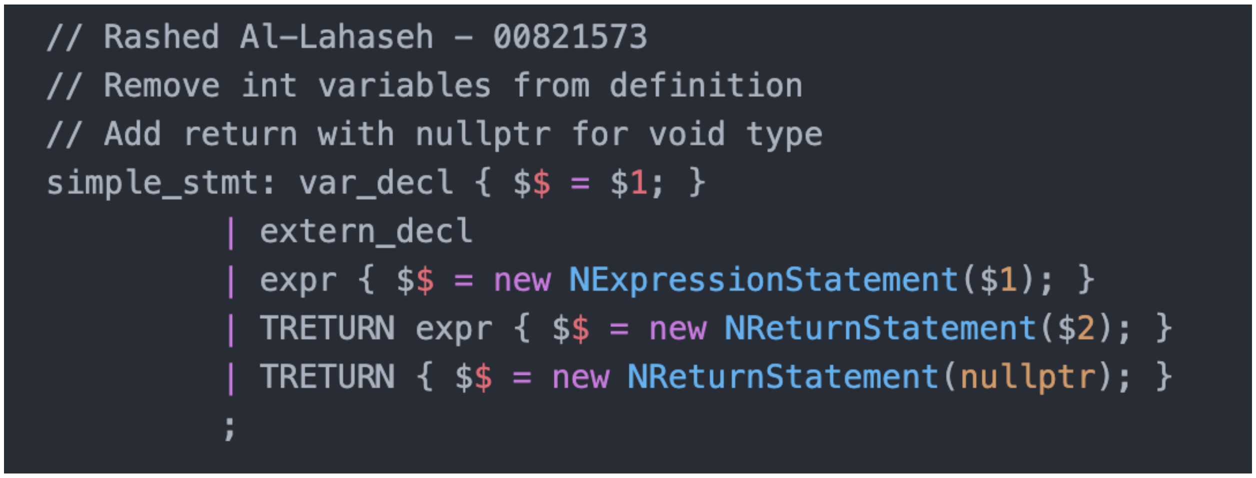
Then we are going to redefine some grammars to include the new added void type.



```
// Rashed Al-Lahaseh - 00821573
// Add void type
type: TINT_T  { $$ = new NIdentifier(yytext); }
    | TVOID_T { $$ = new NIdentifier(yytext);}
```

Figure 6: parser.y file, adding the new defined void type

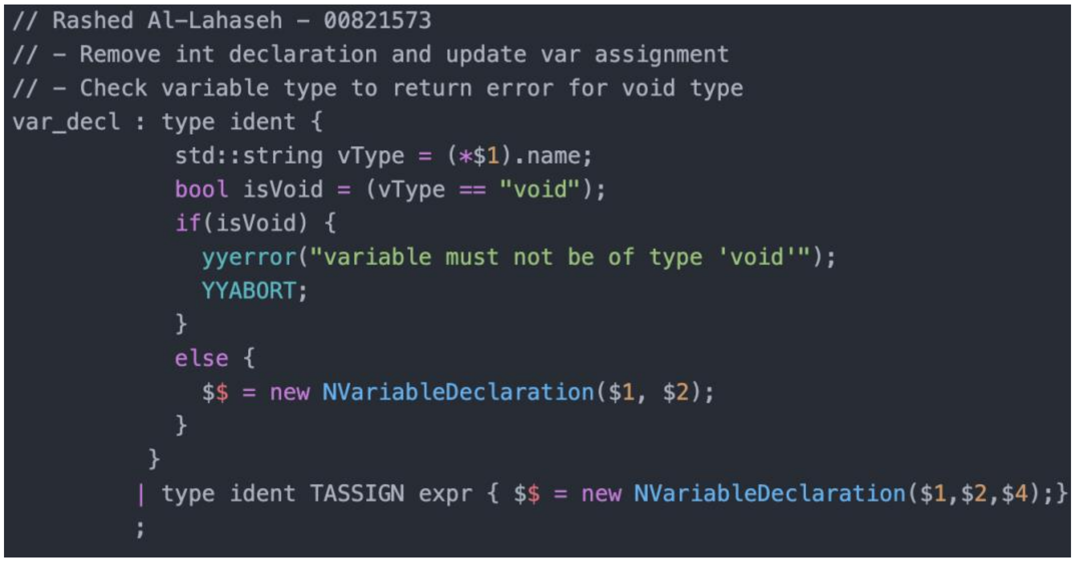
Also, we need to add a new rule for statement definition to accept returning null pointer for void types.



```
// Rashed Al-Lahaseh - 00821573
// Remove int variables from definition
// Add return with nullptr for void type
simple_stmt: var_decl { $$ = $1; }
           | extern_decl
           | expr { $$ = new NExpressionStatement($1); }
           | TRETURN expr { $$ = new NReturnStatement($2); }
           | TRETURN { $$ = new NReturnStatement(nullptr); }
           ;
```

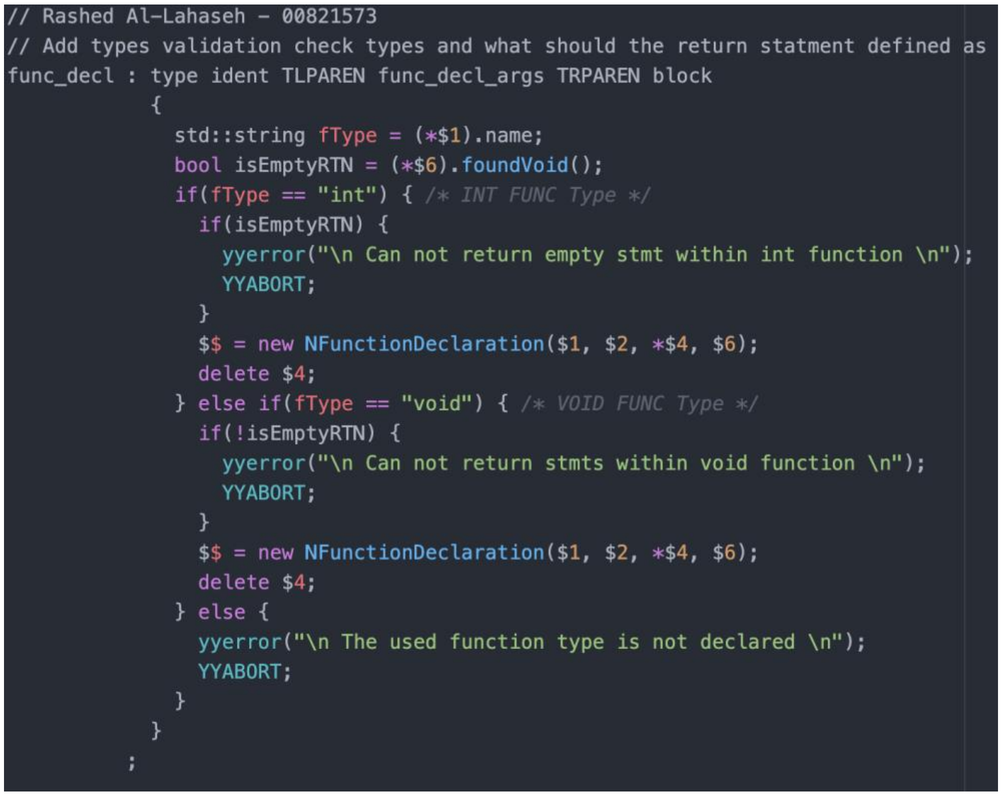
Figure 7: parser.y file, add nullptr return type

And finally, we are going to add the validation for checking void type so we make sure its only valid for void defied type.



```
// Rashed Al-Lahaseh - 00821573
// - Remove int declaration and update var assignment
// - Check variable type to return error for void type
var_decl : type ident {
            std::string vType = (*$1).name;
            bool isVoid = (vType == "void");
            if(isVoid) {
              yyerror("variable must not be of type 'void'");
              YYABORT;
            }
            else {
              $$ = new NVariableDeclaration($1, $2);
            }
          }
         | type ident TASSIGN expr { $$ = new NVariableDeclaration($1,$2,$4);}
         ;
```

Figure 8: parser.y file, add variable validation for check types



```
// Rashed Al-Lahaseh - 00821573
// Add types validation check types and what should the return statment defined as
func_decl : type ident TLPAREN func_decl_args TRPAREN block
            {
              std::string fType = (*$1).name;
              bool isEmptyRTN = (*$6).foundVoid();
              if(fType == "int") { /* INT FUNC Type */
                if(isEmptyRTN) {
                  yyerror("\n Can not return empty stmt within int function \n");
                  YYABORT;
                }
                $$ = new NFunctionDeclaration($1, $2, *$4, $6);
                delete $4;
              } else if(fType == "void") { /* VOID FUNC Type */
                if(!isEmptyRTN) {
                  yyerror("\n Can not return stmts within void function \n");
                  YYABORT;
                }
                $$ = new NFunctionDeclaration($1, $2, *$4, $6);
                delete $4;
              } else {
                yyerror("\n The used function type is not declared \n");
                YYABORT;
              }
            }
          ;
```

Figure 9: parser.y file, add variable validation for check defined function type

Assembling the AST using LLVM

The subsequent step in a compilers' system is to transform this AST to machine code. This includes translating every semantic node right into a machine coaching equivalent.

LLVM simplifies this for us with the aid of using abstracting the character commands into whatever that resembles an AST. That is, what we're doing is changing from one AST to the subsequent.

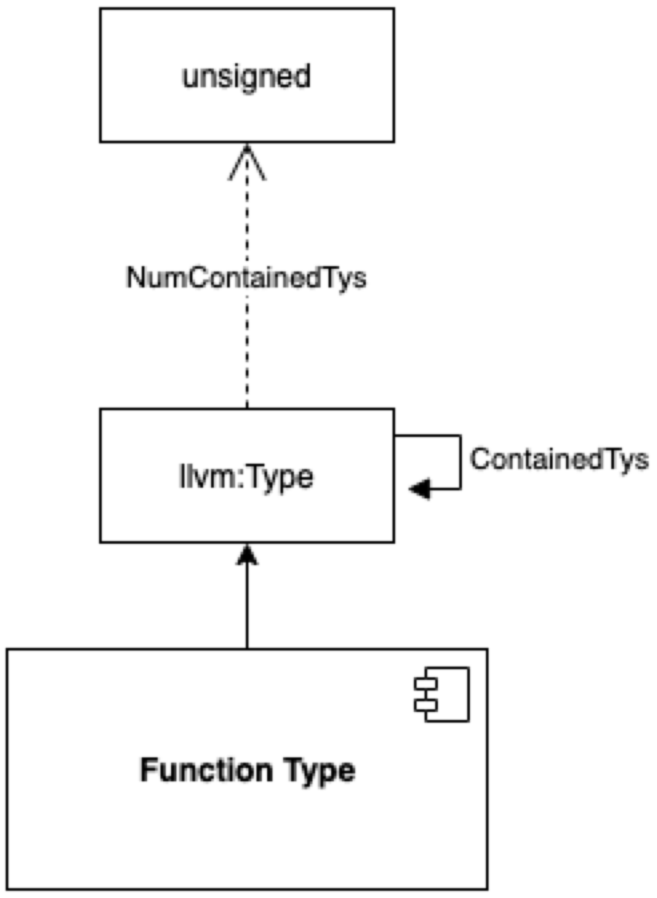
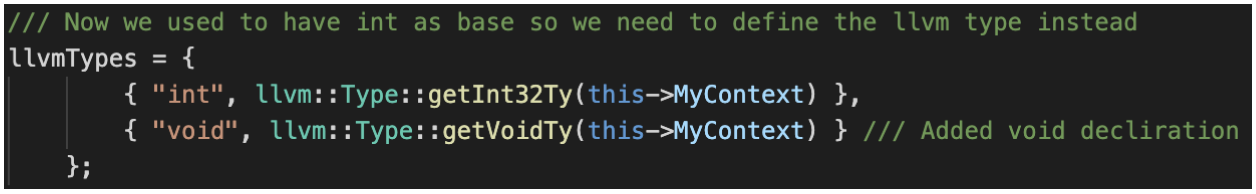


Figure 10: LLVM-Function-Type Functionality Structure

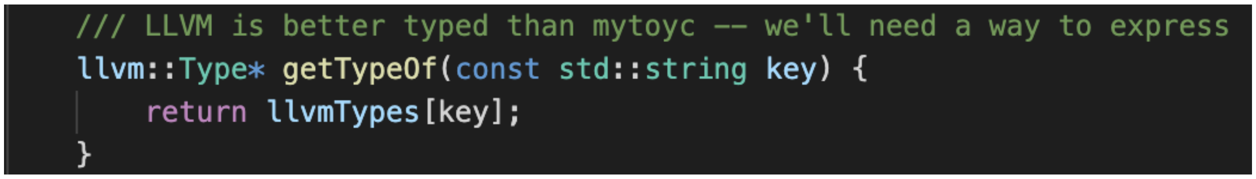
Definition

So, as we said we are going to define the void type now



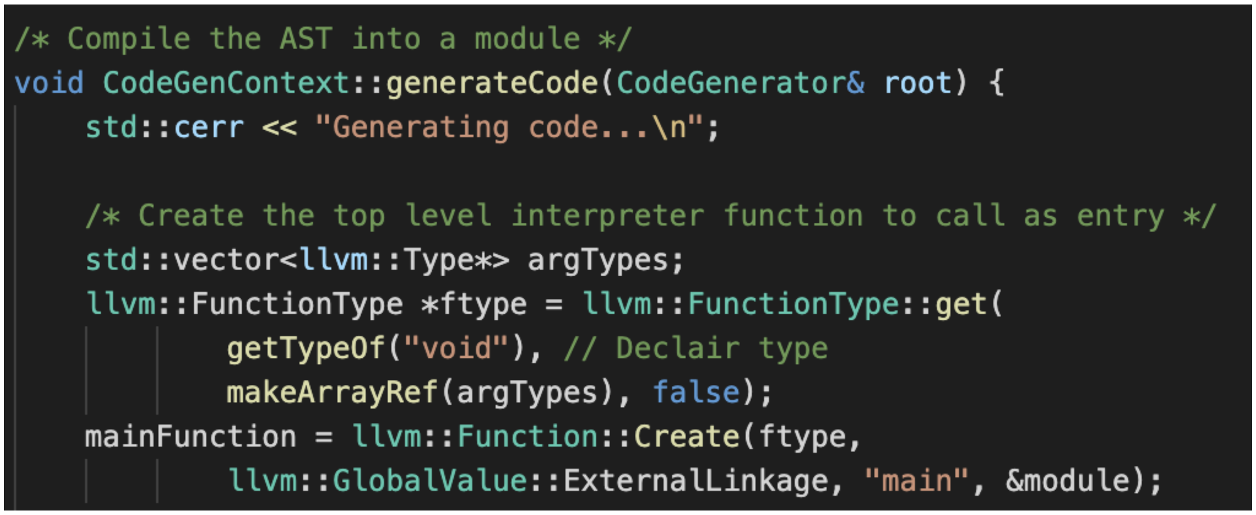
```
/// Now we used to have int as base so we need to define the llvm type instead
llvmTypes = {
        { "int", llvm::Type::getInt32Ty(this->MyContext) },
        { "void", llvm::Type::getVoidTy(this->MyContext) } /// Added void decliration
    };
```

Figure 11: codegen.cpp file, generating new void type



```
    /// LLVM is better typed than mytoyc -- we'll need a way to express
    llvm::Type* getTypeOf(const std::string key) {
        return llvmTypes[key];
    }
```

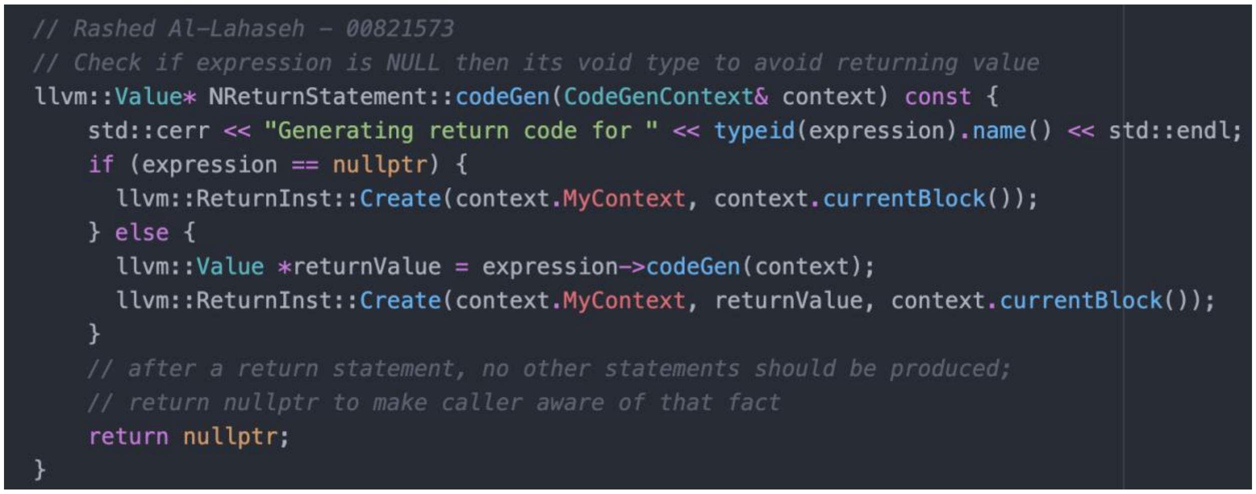
Figure 12: codegen.cpp file, add new functionality to get the current used llvm functionality type



```
/* Compile the AST into a module */
void CodeGenContext::generateCode(CodeGenerator& root) {
    std::cerr << "Generating code...\n";

    /* Create the top level interpreter function to call as entry */
    std::vector<llvm::Type*> argTypes;
    llvm::FunctionType *ftype = llvm::FunctionType::get(
            getTypeOf("void"), // Declair type
            makeArrayRef(argTypes), false);
    mainFunction = llvm::Function::Create(ftype,
            llvm::GlobalValue::ExternalLinkage, "main", &module);
```

Figure 13: codegen.cpp file, void function type definition

And also, to check the definition of return statement, so in case of void function no need to include the return value within it.



```
// Rashed Al-Lahaseh - 00821573
// Check if expression is NULL then its void type to avoid returning value
llvm::Value* NReturnStatement::codeGen(CodeGenContext& context) const {
    std::cerr << "Generating return code for " << typeid(expression).name() << std::endl;
    if (expression == nullptr) {
      llvm::ReturnInst::Create(context.MyContext, context.currentBlock());
    } else {
      llvm::Value *returnValue = expression->codeGen(context);
      llvm::ReturnInst::Create(context.MyContext, returnValue, context.currentBlock());
    }
    // after a return statement, no other statements should be produced;
    // return nullptr to make caller aware of that fact
    return nullptr;
}
```

Figure 14: codegen.cpp add nullptr definition for return statment

References

- https://llvm.org/doxygen
- http://www.gnu.org/software/bison/manual/bison.html
- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/
- https://github.com/DoctorWkt/acwj
- https://mygit.th-deg.de/prgai/mytoyc
- https://gnuu.org/2009/09/18/writing-your-own-toy-compiler/

Conclusion

In this semester, we learned a lot by interacting with four different interrelated tasks. In fact, this project consists of different tasks where each task is a stepping stone for the next task. In the first task, we will start with the Lex (lexical analysis) where we had to split our input data into a set of tokens such as (identifiers, keywords, brackets, etc.). This helped us understand how Flex works.

In the second task, we deal with the semantic parsing, where we use Bison to generate an AST while parsing the tokens and what's nice here is that Bison is going to do most of the legwork here as we will notice we are only going to define our needed AST.

In the third task, we further interact with our assembly using LLVM. This part is responsible for generating the machine code for each node where it walks over our AST.

In general, this workshop (project) is a good experience for learning new things and solving problems, and understanding how programming languages works as much as you get deeper on it as much as you start realizing that not all the parts hard to understand.

So, in the end, I would like to thank you Prof. Dr. Ing. **Peter Faber** for the material was provided through the course and for your help during our classes. Because we believe that this course gave us the good knowledge to dig in more on this field because as I believe this field is very important and still actively in the market with the new languages being provided in our days such as (Kotlin, Dark, Elm, etc.).