


LLVM

Conținut

1. [LLVM. Prezentare](#)
2. [Dezvoltare back-end](#)
3. [clang](#)
 1. [Exemple de compilare clang + LLVM](#)

1. Prezentare LLVM

Conform definiției date de dezvoltatorii LLVM-ului acesta este

 Proiectul LLVM are mai multe componente. Nucleul proiectului se numește „LLVM”. Acesta conține toate instrumentele, bibliotecile și fișierele de antet necesare pentru procesarea reprezentărilor intermediare și conversia acestora în fișiere obiect. Instrumentele includ un asamblor, un dezasamblor, un analizor de coduri de biți și un optimizator de coduri de biți. Acesta conține, de asemenea, teste de regresie de bază.


2. Generarea Codului LLVM IR

Prerequisites

☐ LLVM

```
sudo apt install llvm-dev
```

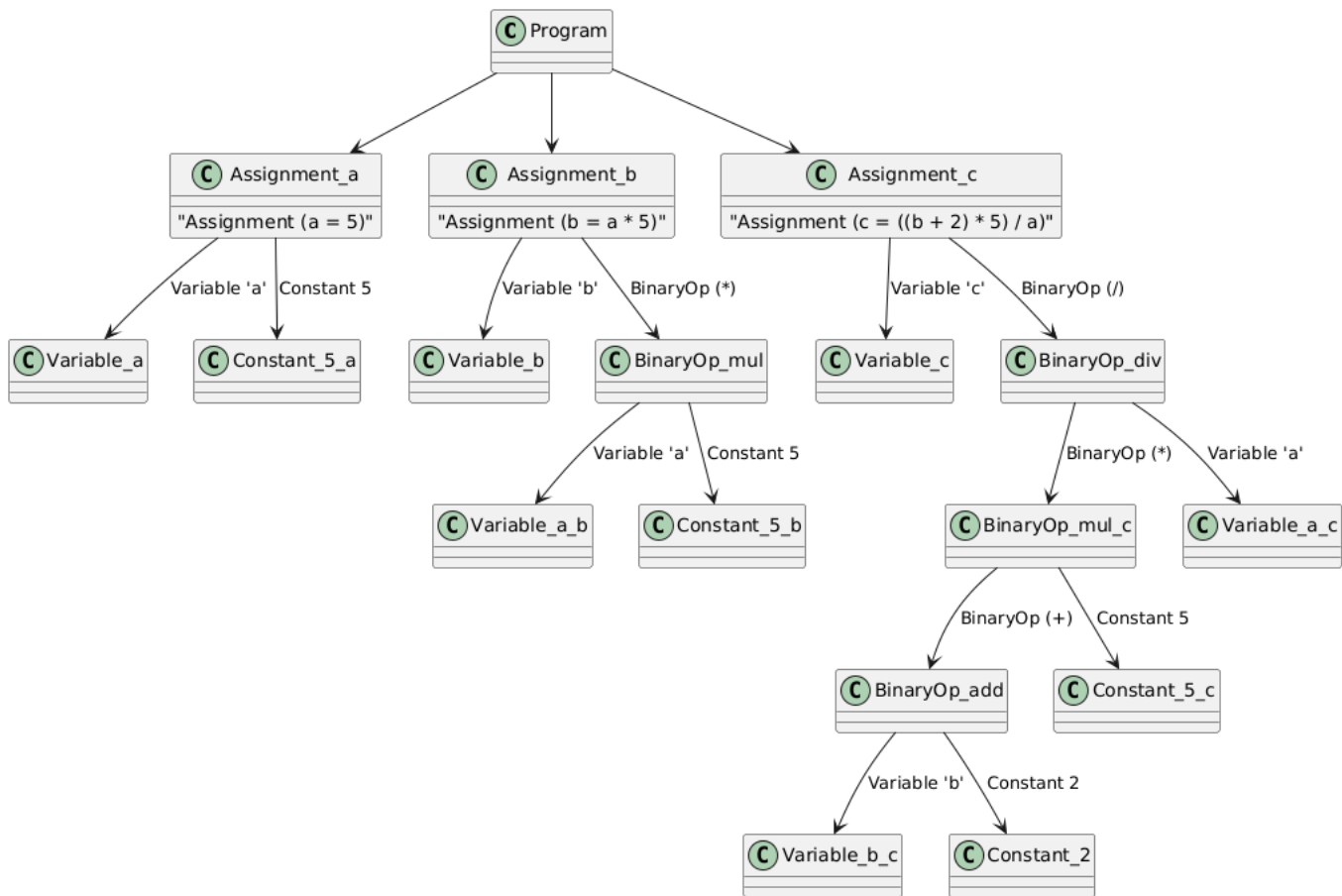
AST

 Reprezentarea AST (Abstract Syntax Tree) este un pas esențial în procesul de compilare atunci când folosim LLVM (sau orice alt framework de generare de cod intermediar). AST-ul servește drept punte între analiza inițială a codului sursă și generarea de cod executabil. Un AST (Abstract Syntax Tree) este o reprezentare ierarhică a structurii logice a unui program. Spre deosebire de codul sursă brut, AST-ul elimină detalii de suprafață, cum ar fi paranteze sau caractere de spațiu, păstrând doar structura semantică a programului.

Pseudocod:

```
Integer a = 5;  
Integer b = a * 5;  
Integer c = (b + 2) * 5 / a;
```

AST este o structură arborescentă care reprezintă operațiunile și expresiile. Iată cum arată AST pentru acest cod:



Clasele de reprezentare a nodurilor AST

```

#include <memory>
#include <vector>
#include <string>

// Base class for all AST nodes
class ASTNode {
public:
    virtual ~ASTNode() = default;
};

// Represents a variable
class Variable : public ASTNode {
public:
    std::string name;
    explicit Variable(const std::string& name) : name(name) {}
};

// Represents a constant (integer value)
class Constant : public ASTNode {
public:
    int value;
    explicit Constant(int value) : value(value) {}
};
  
```

```

};

// Represents a binary operation (e.g., +, -, *, /)
class BinaryOp : public ASTNode {
public:
    char op;
    std::unique_ptr<ASTNode> left;
    std::unique_ptr<ASTNode> right;

    BinaryOp(char op, std::unique_ptr<ASTNode> left, std::unique_ptr<ASTNode>
right)
        : op(op), left(std::move(left)), right(std::move(right)) {}
};

// Represents an assignment (e.g., a = 5)
class Assignment : public ASTNode {
public:
    std::string variableName;
    std::unique_ptr<ASTNode> expression;

    Assignment(const std::string& variableName, std::unique_ptr<ASTNode>
expression)
        : variableName(variableName), expression(std::move(expression)) {}
};

// Represents the program root
class Program : public ASTNode {
public:
    std::vector<std::unique_ptr<ASTNode>> statements;

    void addStatement(std::unique_ptr<ASTNode> statement) {
        statements.push_back(std::move(statement));
    }
};

```

Pentru a crea IR (Reprezentare intermediara) mai intai trebuie sa construim arborele AST, in cazul nostru, o sa simplificăm implementare frontend-ului (parserul, analiza sintactica si semantica) si o sa trecem direct la crearea arborelui intr-o maniera hardcodată.

```

std::unique_ptr<Program> buildAST() {
    auto program = std::make_unique<Program>();

    // a = 5
    program->addStatement(std::make_unique<Assignment>(
        "a",
        std::make_unique<Constant>(5)
    ));

    // b = a * 5
    program->addStatement(std::make_unique<Assignment>(

```

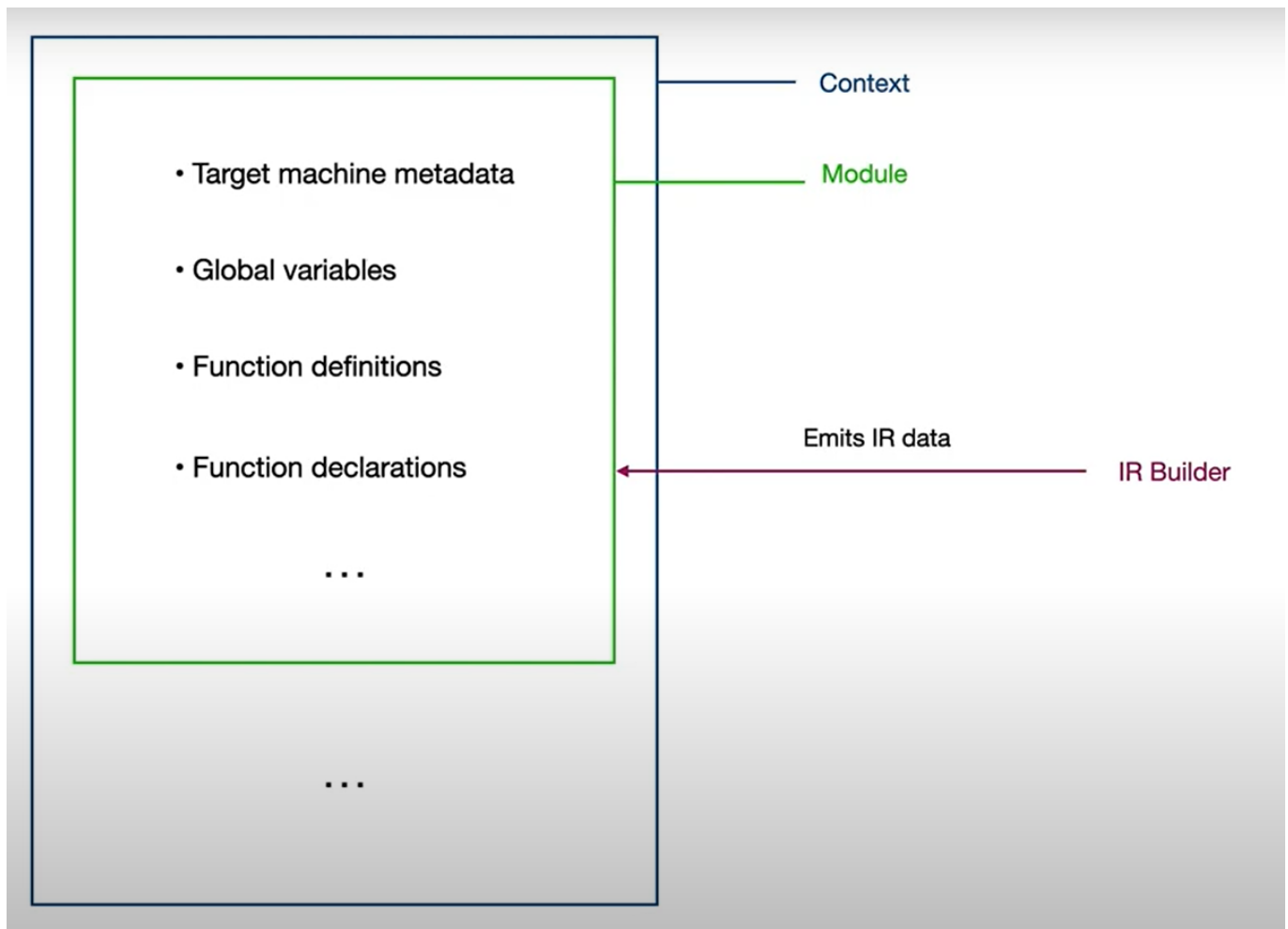
```
        "b",
        std::make_unique<BinaryOp>(<
            '*',
            std::make_unique<Variable>("a"),
            std::make_unique<Constant>(5)
        )
    ));

// c = (b + 2) * 5 / a
program->addStatement(std::make_unique<Assignment>(<
    "c",
    std::make_unique<BinaryOp>(<
        '/',
        std::make_unique<BinaryOp>(<
            '*',
            std::make_unique<BinaryOp>(<
                '+',
                std::make_unique<Variable>("b"),
                std::make_unique<Constant>(2)
            ),
            std::make_unique<Constant>(5)
        ),
        std::make_unique<Variable>("a")
    )
));

return program;
}
```



LLVM este o infrastructură modulară și flexibilă pentru compilatoare, iar în acest context, concepte precum module, context, funcții, și alte elemente joacă un rol esențial. Aceste componente definesc modul în care se organizează și se generează codul intermediar (IR). Să analizăm detaliat fiecare concept:



1. LLVM Context (`llvm::LLVMContext`)

Contextul reprezintă spațiul de lucru în care LLVM păstrează toate datele legate de tipuri, constante și alte obiecte globale. Este o componentă crucială pentru a preveni conflictele și pentru a gestiona resursele.

Roluri principale:

Gestionarea unică a obiectelor: Tipurile și constantele sunt create și gestionate într-un context pentru a evita duplicarea.

Izolare între module: Poți avea mai multe contexte dacă generezi mai multe module separate.

Performanță: Contextul permite reutilizarea obiectelor comune (de exemplu, tipuri de date) pentru a reduce consumul de memorie.

2. Module (`llvm::Module`)

Un modul reprezintă un program sau o unitate de traducere în LLVM. Este echivalent cu un fișier sursă în alte compilatoare și conține toate definițiile globale ale unui program.

Conținutul unui modul:

Funcții: Definiții și declarații de funcții.

Variabile globale: Date globale accesibile din toate funcțiile modulului.

Metadata: Informații adiționale, cum ar fi debug info sau optimizări.

Tipuri și constante: Obiecte definite pentru utilizare globală.

Roluri principale:

Organizare: Reunește toate componentele necesare unui program.

Compilare: Modulul este unitatea de bază transmisă backend-ului pentru generarea de cod nativ.

Exemplu:

```
llvm::Module module("MyModule", context);
```

3. Funcție (`llvm::Function`)

O funcție este o secvență de instrucțiuni IR care îndeplinește o anumită sarcină. În LLVM, funcțiile sunt componente de bază în cadrul unui modul.

Atribute ale unei funcții:

Tipul funcției (`FunctionType`): Descrie tipul valorii returnate și tipurile parametrilor.

Corpul funcției: Instrucțiunile efective, organizate în blocuri de bază.

Linkage: Modificatorii de acces (e.g., internal, external).

Blocuri de bază: Fiecare funcție este compusă din unul sau mai multe blocuri de bază (`BasicBlock`). fjdls

Exemplu

```
auto* funcType = llvm::FunctionType::get(builder.getInt32Ty(), false);
auto* mainFunc = llvm::Function::Create(
    funcType, llvm::Function::ExternalLinkage, "main", module);
```

4. Bloc de bază (`llvm::BasicBlock`)

Un bloc de bază este o secvență de instrucțiuni consecutive, fără ramificații interne. Este unitatea fundamentală a controlului fluxului în LLVM.

Atribute ale unui bloc:

Instrucțiuni: Secvența efectivă de operații. Terminator: Fiecare bloc trebuie să se termine cu o instrucțiune de control al fluxului (e.g., ret, br). Exemplu:

```
auto* entryBlock = llvm::BasicBlock::Create(context, "entry", mainFunc);
builder.SetInsertPoint(entryBlock);
```

5. Instrucțiuni (`llvm::Instruction`)

Instrucțiunile reprezintă operațiile efective din IR. Fiecare instrucțiune aparține unui bloc de bază și este generată folosind `IRBuilder`.

Tipuri de instrucțiuni:

Aritmetice: add, sub, mul, sdiv.

Memorie: alloca, load, store.

Control al fluxului: br, ret, call.

Exemplu:

```
llvm::Value* a = builder.CreateAlloca(builder.getInt32Ty(), nullptr, "a");
llvm::Value* five = llvm::ConstantInt::get(context, llvm::APInt(32, 5));
builder.CreateStore(five, a);
```

6. Tipuri (llvm::Type)

Tipurile definesc structura datelor utilizate în instrucțiunile IR. LLVM este strict tipizat, deci toate operațiile trebuie să aibă tipuri compatibile.

Tipuri comune:

Scalar: i32 (integer pe 32 de biți), float, double.

Pointeri: i32*.

Structuri: Agregate complexe.

Exemplu:

```
llvm::Type* int32Type = llvm::Type::getInt32Ty(context);
```

7. Builder (llvm::IRBuilder)

IRBuilder este un utilitar pentru generarea de instrucțiuni în LLVM IR. Este folosit pentru a simplifica procesul de creare a instrucțiunilor și pentru a seta punctul curent de inserție.

Roluri:

Generarea instrucțiunilor: Permite crearea de instrucțiuni cu metode intuitive (e.g., CreateAdd, CreateLoad).

Gestionarea punctului de inserție: Definește unde sunt adăugate instrucțiunile în IR.

Exemplu:

```
llvm::IRBuilder<> builder(entryBlock);
auto* sum = builder.CreateAdd(aValue, five, "addtmp");
```

8. Memorie (alloca, load, store)

LLVM gestionează memoria explicit pentru variabile utilizând instrucțiuni IR:

alloca: Alocă spațiu pentru o variabilă pe stiva.

```
auto* a = builder.CreateAlloca(builder.getInt32Ty(), nullptr, "a");
```

store: Salvează o valoare în memoria alocată.

```
builder.CreateStore(five, a);
```

load: Încarcă valoarea din memorie.

```
auto* aValue = builder.CreateLoad(a);
```

9. Relația între concepte

Context: Gestionează totul (tipuri, constante, etc.).

Module: Containere pentru funcții și variabile globale.

Funcții: Blocuri organizate de instrucțiuni, construite cu ajutorul builder-ului.

Blocuri de bază: Structuri pentru controlul fluxului.

Instrucțiuni: Operații individuale din cod.

Codul pentru generare LLVM IR

```
#include <llvm/IR/IRBuilder.h>
#include <llvm/IR/LLVMContext.h>
#include <llvm/IR/Module.h>
#include <llvm/IR/Verifier.h>
#include <iostream>

// Function to generate LLVM IR from the AST
llvm::Value* generateIR(ASTNode* node, llvm::IRBuilder<>& builder,
    llvm::LLVMContext& context, llvm::Module& module, std::map<std::string,
    llvm::Value*>& namedValues) {
    if (auto* var = dynamic_cast<Variable*>(node)) {
        // Load a variable
        return builder.CreateLoad(builder.getInt32Ty(), namedValues[var->name]);
    } else if (auto* constant = dynamic_cast<Constant*>(node)) {
        // Return a constant value
        return llvm::ConstantInt::get(builder.getInt32Ty(), constant->value);
    } else if (auto* binOp = dynamic_cast<BinaryOp*>(node)) {
        // Process binary operation
        llvm::Value* left = generateIR(binOp->left.get(), builder, context,
module, namedValues);
        llvm::Value* right = generateIR(binOp->right.get(), builder, context,
module, namedValues);
        switch (binOp->op) {
            case '+': return builder.CreateAdd(left, right, "addtmp");
            case '*': return builder.CreateMul(left, right, "multmp");
            case '/': return builder.CreateSDiv(left, right, "divtmp");
            default: throw std::runtime_error("Unknown binary operator");
        }
    } else if (auto* assignment = dynamic_cast<Assignment*>(node)) {
```



```

        // Process assignment
        llvm::Value* value = generateIR(assignment->expression.get(), builder,
context, module, namedValues);
        builder.CreateStore(value, namedValues[assignment->variableName]);
        return value;
    }
    throw std::runtime_error("Unknown AST node type");
}

int main() {
    // Setup LLVM components
    llvm::LLVMContext context;
    llvm::Module module("SimpleModule", context);
    llvm::IRBuilder<> builder(context);

    // Define variables map
    std::map<std::string, llvm::Value*> namedValues;

    // Create the main function
    llvm::FunctionType* funcType = llvm::FunctionType::get(builder.getInt32Ty(),
false);
    llvm::Function* mainFunc = llvm::Function::Create(funcType,
llvm::Function::ExternalLinkage, "main", module);

    // Create the entry basic block
    llvm::BasicBlock* entryBlock = llvm::BasicBlock::Create(context, "entry",
mainFunc);
    builder.SetInsertPoint(entryBlock);

    // Allocate variables
    namedValues["a"] = builder.CreateAlloca(builder.getInt32Ty(), nullptr, "a");
    namedValues["b"] = builder.CreateAlloca(builder.getInt32Ty(), nullptr, "b");
    namedValues["c"] = builder.CreateAlloca(builder.getInt32Ty(), nullptr, "c");

    // Build the AST
    auto program = buildAST();

    // Traverse the AST and generate IR
    for (auto& stmt : program->statements) {
        generateIR(stmt.get(), builder, context, module, namedValues);
    }

    // Return 0
    builder.CreateRet(llvm::ConstantInt::get(builder.getInt32Ty(), 0));

    // Verify the module and print IR
    if (llvm::verifyModule(module, &llvm::errs())) {
        std::cerr << "Error: Module verification failed\n";
        return 1;
    }

    module.print(llvm::outs(), nullptr);
    return 0;
}

```

3. Optimizarea IR
