PROJEKTNI ZADATAK IZ KONSTRUKCIJE KOMPAJLERA

KOMPAJLER ZA C#

Br. Indeksa: 90/21

Radio: David Sabljić

SADRŽAJ

UVOD

U ovom seminarskom radu opisan je projekat iz predmeta Konstrukcija kompilatora. Projekat se zasniva na “My First Language Frontend with LLVM” tutorijalu, koji je dostupan na https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html. Kompajler je pravljen za programski jezik C#. U nastavku seminarskog rada je opisana ideja LLVM-a, C# i njegova sintaksa, kao i ideja za pisanje kopmajlera za C#.

OSNOVNO O LLVM-u

LLVM je skup tehnologija kompajlera i alata koji se mogu koristiti za razvoj frontenda za bilo koji programski jezik i pozadine za bilo koju arhitekturu skupa instrukcija. LLVM je dizajniran oko jezika nezavisnog srednjeg predstavljanja (IR) koji služi kao prenosivi asemblerski jezik visokog nivoa koji se može optimizovati raznim transformacijama tokom višestrukih prolaza.

LLVM projekat je započeo 2000. godine na Univerzitetu Ilinois u Urbana-Šampaignu, pod rukovodstvom Vikrama Advea i Chrisa Lattnera. LLVM je prvobitno razvijen kao istraživačka infrastruktura za istraživanje tehnika dinamičke kompilacije za statičke i dinamičke programske jezike. Ime LLVM je prvobitno označavalo virtuelnu mašinu niskog nivoa, iako se projekat proširio i naziv više nije zvanično inicijalizam. Napisan je u c++ programskom jeziku.

OSNOVNO O C#

C# je jedan od mlađih programskih jezika. Nastao je 2000. godine kao sastavni dio Majkrosoftovog razvojnog okruženja .NET Framework 1.0. C# podržava više paradigmi (objektno orijentisanu, imperativnu, deklarativnu, generičku) kao i većina modernih viših programskih jezika. Jezik je opšte primjene i namenjen je izradi aplikacija za .NET Framework platformu.

C# ima široku primjenu - C# programski jezik omogućava da se kreiraju programi i aplikacije za Windows platformu i internet. U suštini, ovim jezikom možete praviti mnoštvo projekata: od mobilnih aplikacija, cloud usluga, kompanijskih softvera i video-igrica.

Razlika između C# i C++ - Kod u jeziku C# je malo razumljiviji od koda C++. Jezik C# je sigurniji po pitanju dodjele tipa promenljivima. To znači da ukoliko nekoj promenljivoj ili nekom podatku dodjelimo određeni tip, on ne može biti zamenjen drugim tipom. Postoje stroga pravila kojih se moramo pridržavati prilikom konverzija tipova, što u suštini znači da ćemo morati da napišemo više koda u C# nego u C++-u da bismo izvršili iste zadatke. Kod je robusniji i jednostavnije je otkloniti greške, odnosno .NET u svakom trenutku može odrediti kom tipu pripada traženi podatak. Za razliku od C++, C# ne podržava višestruko nasleđivanje, iako klasa može implementirati više interfejsa.

SINTAKSA C#

Sintaksa C# jezika temelji se na C i C++ jezicima. Neki osnovni elementi c# sintakse:

1. **Definicija klase i metode:**

   C# je objektno orijentirani jezik, pa je osnovna jedinica koda klasa. Svaka C# aplikacija ima barem jednu klasu.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

2. **Varijable i tipovi podataka:**

   C# je statički tipiziran jezik, što znači da tipovi varijabli moraju biti poznati prilikom kompajliranja. Neki osnovni tipovi podataka su:

   - int (cijeli brojevi)
   - double (brojevi s decimalama)
   - bool (logičke vrijednosti: true ili false)
   - string (tekstualni podaci)

```
int broj = 10;
double decimalniBroj = 10.5;
bool istina = true;
string poruka = "Pozdrav!";
```

3. **Petlje:**

   C# podržava uobičajene petlje poput for, while, i foreach.

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

4. **Kontrolne strukture:**

C# podržava uobičajene strukture poput if, else, i switch.

```
int broj = 10;
if (broj > 5)
{
    Console.WriteLine("Broj je veći od 5");
}
else
{
    Console.WriteLine("Broj je manji ili jednak 5");
}
```

5. **Funkcije (Metode):**

Metode u C#-u omogućuju organizaciju koda u ponovo iskoristive blokove.

```
int Zbroji(int a, int b)
{
    return a + b;
}

int rezultat = Zbroji(3, 5);
Console.WriteLine(rezultat);
```

6. **Objektno orijentisane karakteristike:**

```
class Zivotinja
{
    public void Glasaj() => Console.WriteLine("Životinja pravi zvuk.");
}

class Pas : Zivotinja
{
    public void Lajanje() => Console.WriteLine("Pas laje.");
}
```

IMPLEMENTACIJA KOMPAJLERA

```
///--------- TOKEN ---------
///-------------------------

enum Token{
    tok_eof = -1,

    tok_using = -2,
    tok_class = -3,
    tok_static = -4,
    tok_void = -5,
    tok_int = -7,
    tok_if = -8,
    tok_else = -9,

    tok_identifier = -10,
    tok_literal = -11,
    tok_operator = -12,
    tok_symb = -13, /// Punctuation and Delimiters
    tok_dot = -14,
    tok_number = -15,
};

static std::string IdentifierStr; /// Filled in if tok_identifier
static std::string LiteralVal;    /// Filled in if tok_literal
static std::string OperatorVal;   /// Filled in if tok_operator
static std::string SymbolVal;     /// Filled in if tok_symb or tok_dot
static int NumVal;                /// Filled in if tok_number
```

Token – enumerativni tip podatka, zadužen za lakše razlikovanje tokena koje dobijamo od lexera.

IdentifierStr – Čuva naziv identifikatora.

LiteralVal – Čuva vrijednost teksta izmedju navodnika (“ ”).

OperatorVal – Čuva vrijednost operatora.

SymbolVal – Čuva vrijednost simbola (npr. }).

NumVal – Čuva vrijednost broja.

Implementacija lexera:

```
///--------- LEXER ---------
///-------------------------

static int gettok(){

    static int LastChar = ' ';

    while (isspace(LastChar)){      /// Skip any whitespace
        LastChar = getNextChar();
    }
```

Lexer će nam biti predstavljen kao funkcija gettok(). Prva while petlja nam služi da ignorišemo sve bjeline.

Sledeće provjeravamo da li je token kljucna rijec ili identifkator. Ako je identifikator vratimo tok_idetifikator, a ako je ključna riječ vratimo odredjeni token za tu riječ.

```
if (isalpha(LastChar)){ /// identifier: [a-zA-Z][a-zA-Z0-9]*
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getNextChar()))){
        IdentifierStr += LastChar;
    }
    if (IdentifierStr == "using"){
        return tok_using;
    }
    else if (IdentifierStr == "class"){
        return tok_class;
    }
    else if (IdentifierStr == "static"){
    else if (IdentifierStr == "void"){
    else if (IdentifierStr == "int"){
    else if (IdentifierStr == "if"){
    else if (IdentifierStr == "else"){
    else if(IdentifierStr == "EOF"){
    return tok_identifier;
}
```

Ako ulaz nije identifikator onda, provjeravamo ostale mogućnosti kao što su brojevi, operatori, tekst i simboli.

```
if (isdigit(LastChar)) {   /// int: [0-9]+
    std::string NumStr;
    do {
        NumStr += LastChar;
        LastChar = getNextChar();
    } while (isdigit(LastChar));
    NumVal = std::stoi(NumStr);
    return tok_number;
}

if (LastChar == '=' || LastChar == '+' || LastChar == '-' || LastChar == '>' || LastChar == '<'){ /// operators
    OperatorVal = LastChar;
    LastChar = getNextChar();
    return tok_operator;
}

if (LastChar == ';' || LastChar == '.' || LastChar == '(' || LastChar == ')' || LastChar == '{' || LastChar == '}' ){ /// symbols
        SymbolVal = LastChar;
        if (LastChar =='.'){
            LastChar = getNextChar();
            return tok_dot;
        }
        LastChar = getNextChar();
        return tok_symb;
}
```

Implementacija Abstraktong sintaksong stabla:

Poslije leksera pravimo klase koje nam predstavljaju čvorove abstraktnog sintaksog stabla. Prvo kreiramo osntovni čvor koji će nasljedjivati svi ostali čvorovi.

```
///--------- Abstract Syntax Tree (aka Parse Tree) ---------
///---------------------------------------------------------

class ASTNode  {    /// ASTNode - Base class for all expression nodes
public:
  virtual ~ASTNode() = default;
  virtual Value *codegen() = 0;
};

using ASTNodePtr = std::shared_ptr<ASTNode>;
```

Dalje od ove bazne klase pravimo ostale potrebne čvoreve. Primjer takvih čvoreva u kodu su ClassDeclaration čvor i VariableReference čvor.

```
class ClassDeclaration : public ASTNode {   /// ClassDeclaration - Èvor za Deklaraciju Klase
public:
    std::string name;
    std::vector<ASTNodePtr> methods;

    ClassDeclaration(const std::string &name) : name(name) {}
    ClassDeclaration(const std::string &name, const std::vector<ASTNodePtr> &methods) : name(name), methods(methods) {}

    void addMethod(ASTNodePtr method) {
        methods.push_back(std::move(method));
    }

    llvm::Value *codegen() override;
};
```

```
class VariableReference : public ASTNode {
public:
    std::string name;

    VariableReference(const std::string& name) : name(name) {}
    llvm::Value *codegen() override;

};
```

Implementacija parsera:

Nakon što su napravljene klase za abstraktno sintaksno stablo, pravimo parser koji će kreirati to stablo.

```
///--------- PARSER ---------
///--------------------------
class Parser {
int CurTok;

int getNextToken() {
        return CurTok = gettok();
    };

public:

    ASTNodePtr Parse() {
        getNextToken();
        return ParseProgram();
    };
```

Parser vodi računa o tome koji je token trenutno na ulazu i na osnovu tih tokena vraća određeno abstraktno sintaksno stablo. Funkcija Parse() služi za pokretanje parsera. Klasa parser u sebi ima veći broj pomoćnih funkcija za parsiranje kao što su ParseBinExp(), ParseLeft() itd.

```
std::shared_ptr<BinaryExpression> ParseBinExp() {
    auto arg1 = ParseLeft();
    getNextToken(); // Eat name
    std::string op = SymbolVal;
    getNextToken(); // Eat op
    auto arg2 =  ParseLeft();
    getNextToken(); // Eat name
    auto temp = std::make_shared<BinaryExpression>(arg1,op,arg2);
    return temp;
}

std::shared_ptr<ASTNode> ParseLeft() {

    if(CurTok == tok_identifier){
        std::string name = IdentifierStr;
        auto left = std::make_shared<VariableReference>(name);
        getNextToken(); // Eat variable
        return left;
    }
    else{
        int number = NumVal;
        auto left = std::make_shared<NumberLiteral>(number);
        getNextToken(); // Eat number
        return left;
    }
}
```

Nakon prolaska kroz sve tokene parser vraća abstraktno sintaksno stablo, koje dalje se generiše pomoću llvm kod generatora.

Implementacija generatora koda:

Da bi generisali LLVM IR od dboijenog abstraktnog sintaksnog stabla, korsiticemo ugradjene generator koda. Prvo deklarišemo potrebne promjenljive.

```
///---------- CODE GENERATOR ----------
///------------------------------------


static std::unique_ptr<LLVMContext> TheContext;
static std::unique_ptr<Module> TheModule;
static std::unique_ptr<IRBuilder<>> Builder;
static std::map<std::string, Value *> NamedValues;
```

Zatim za svaki čvor abstraktnog sintaksnog stabla pravimo funkciju codegen(), koristeći metode iz llvm-a. Primjeri nekih od tih funkcija:

```
Value *BinaryExpression::codegen() {
  Value *L = left->codegen();
  Value *R = right->codegen();
  if (!L || !R) return nullptr;

  if (op == "+")
    return Builder->CreateFAdd(L, R, "addtmp");
  else if (op == "-")
    return Builder->CreateFSub(L, R, "subtmp");
  else if (op == "*")
    return Builder->CreateFMul(L, R, "multmp");
  else if (op == "<") {
    L = Builder->CreateFCmpULT(L, R, "cmptmp");
    return Builder->CreateUIToFP(L, Type::getDoubleTy(*TheContext), "booltmp");
  }
  return nullptr;
}
```

```
Value *VariableReference::codegen() {
  Value *V = NamedValues[name];
  if (!V)
    return LogErrorV("Unknown variable name");
  return Builder->CreateLoad(V, name.c_str());
}
```

Implementacija main funkcije:

```
 // 1. Inicijalizacija LLVM konteksta, modula i IR Builder-a
TheContext = std::make_unique<LLVMContext>();
TheModule = std::make_unique<Module>("my cool jit", *TheContext);
Builder = std::make_unique<IRBuilder<>>(*TheContext);

// 2. Otvorite fajl "test.txt"
std::ifstream infile("test.txt");
if (!infile.is_open()) {
    std::cerr << "Could not open file\n";
    return 1;
}

// 3. Inicijalizujte parser
Parser* parser = new Parser();
parser->Parse();

// 4. Ispisivanje generisanog LLVM IR koda
TheModule->print(llvm::errs(), nullptr);

// 5. Očistite resurse
infile.close();  // Zatvori fajl nakon što završiš
delete parser;  // Oslobodite memoriju za parser
```

U funkciji main prvo deklarišemo potrebne llvm promjenljive kao što su IRBuilder, LLVMContext i Module. Dalje otvaramo tekstualni fajl u kom se nalazi kod i taj kod obrađujemo.

Takođe u main funkciji se nalaze postavke za ciljanu mašinu.

```
InitializeAllTargetInfos();
InitializeAllTargets();
InitializeAllTargetMCs();
InitializeAllAsmParsers();
InitializeAllAsmPrinters();

auto TargetTriple = sys::getDefaultTargetTriple();
TheModule->setTargetTriple(TargetTriple);

std::string Error;
auto Target = TargetRegistry::lookupTarget(TargetTriple, Error);

if (!Target) {
  errs() << Error;
  return 1;
}

auto CPU = "generic";
auto Features = "";

TargetOptions opt;
auto TheTargetMachine = Target->createTargetMachine(
    TargetTriple, CPU, Features, opt, Reloc::PIC_);

TheModule->setDataLayout(TheTargetMachine->createDataLayout());
```

Zaključak:

Prolaskom kroz “My First Language Frontend with LLVM” tutorijal, se stiče osnovna ideja kako LLVM funkcioniše i kako pojednostavljuje izradu kompajlera. Kroz izradu projekta se proslo kroz neke koncepte LLVM-a kao što su generacija IR koda, jit optimizacija, odabir ciljne mašine i generacija object koda. Sa obzirom na složenost zadatka i poteškoće koje su se pojavile tokom implementacije, postoje određeni aspekti u kojima kompajler možda još ne radi potpuno ispravno ili ne generiše potpuno funkcionalan izvršni kod. U konkretnom primjeru test.txt kompajler dolazi do faze generisanja object fajla.

Za dalju izradu projekta neke od glavnih idjea (koraka) bi bile: 1) dodavanja petlji (cikličnih blokova) 2)poboljšavanje djela koda vezanog za generaciju koda 3) dodavanje optimizacije koda 4) preurediti projekat da output bude izvršavanje programa a ne object fajl