KONSTRUKCIJA KOMPILATORA

- LLVM -

Matematički fakultet

Zorana Gajić
400/2016

Beograd 2018.

Sadržaj

1 Lexer

2 Parser. AST

Prva dva poglavlja LLVM tutorijala su upoznavanje sa jezikom kaleidoskop, njegov lexer, parser i apstraktno sintaksno stablo. To propuštam.

3 Generisanje koda – metod codegen

3.1 Uvod

Ovo poglavlje će nam pokazati kako da transformišemo apstraktno sintaksno stablo izgrađeno u prva dva poglavlja u LLVM IR.

3.2 Codegen setup

Da bismo generisali LLVM IR prvo treba da generišemo metod codegen u svakoj AST klasi.
Metod **codegen()** kaže da se generiše IR za taj AST čvor zajedno sa svim stvarima od kojih zavisi, a svi oni vraćaju LLVM Value objekat.

```
1  class ExprAST {
2  public:
3      virtual ~ExprAST() {}
4      virtual Value* codegen() const = 0;
5  };
6
7  class NumberExprAST: public ExprAST {
8  public:
9      NumberExprAST(double d)
10     : _d(d) {}
11     Value* codegen() const;
12 private:
13     double _d;
14 };
15 ...
```

Value je klasa koja se koristi za prikazivanje SSA (*engl. Static Single Assignment register*).
Najvažnije kod SSA je što se njihova vrednost izračunava kako se izvršava neka instrukcija, ali ona ne dobija novu vrednost sve dok se ne izvrši instrukcija. Odnosno, ne postoji način da se "promeni" SSA vrednost.
Statičke varijable će se koristiti tokom generisanja koda.

```
1  // U ast.hpp:
2  #include "llvm/IR/Module.h"
3  #include "llvm/IR/Constants.h"
4  #include "llvm/IR/IRBuilder.h"
5  #include "llvm/IR/Verifier.h"
6  using namespace llvm;
7
```

```
// U ast.cpp:
LLVMContext TheContext;
Module* TheModule;
IRBuilder<> Builder(TheContext);
map<string, Value*> NamedValues;
```

TheContext je objekat koji poseduje mnoštvo LLVM struktura podataka (*na primer informacije o tipovima*), ali je nećemo detaljno razmatrati.

Builder objekat je pomoćni objekat za olakšavanje generisanja LLVM instrukcija. Instance ovog objekta prate trenutnu lokaciju i zato mogu da ubace i kreiraju nove instrukcije.

TheModule je LLVM konstrukt koji sadrzi funkcije i globalne varijable. Ovo je struktura najvišeg nivoa koju LLVM IR koristi da čuva kod. On ce posedovati svu memoriju za sve IR koje generišemo.

NamedValues mapa vodi računa o tome koje su vrednosti definisane u trenutnom opsegu i koja je njihova LLVM reprezentacija. (*drugim rečima, to je tablica simbola za kod*).

Pošto imamo pokazivač na TheModule trebamo ga kreirati. U main-u unutar parser.ypp:

```
extern llvm::Module* TheModule;
extern llvm::LLVMContext TheContext;

int main() {
    TheModule = new llvm::Module("Moj modul", TheContext);
    yyparse();

    TheModule->dump();
    delete TheModule;
    return 0;
}
```

Metod **dump()** će ispisati sve što se nalazi u TheModule, odnosno sve što je izgenerisano sa codegen().

3.3 Expression Code Generation

Počnimo od klase NumberExprAST.

```
Value* NumberExprAST::codegen() const {
    return ConstantFP::get(TheContext, APFloat(_d));
}
```

U LLVM IR, numeričke konstante su predstavljene ConstantFP klasom. Ona interno sadrži numeričku vrednost u APFloat. Ovaj kod kreira i vraća ConstantFP.
Treba imati na umu da su u LLVM IR konstante sve jedinstvene i zajedničke. Zato nigde nećemo imati "new foo(...)" ili "foo::Create(...)", vec "foo::get(...)"

```
Value* VariableExprAST::codegen() const {
    Value *v = NamedValues[_v];
    if(v == NULL) {
        cerr << "Nepostojeca promenljiva " << _v << endl;
        exit(EXIT_FAILURE);
    }
    return v;
}
```

U jednostavnoj verziji Kaleidoskopa pretpostavljamo da je varijabla već negde emitovana i da je njena vrednost dostupna (*u nasem slučaju da je već u mapi NamedValues*).

Ovaj kod proverava da li je navedeno ime u mapi. Ako nije prijavljuje grešku, inače vraćamo njenu vrednost.

Što se tiče binarnih operatora, osnovna ideja je da rekurzivno pozovemo codegen() za levu stranu izraza, a zatim za desnu i onda izracunamo rezultat binarnog izraza.
Ovde ćemo koristiti LLVM klasu Builder, jer će on sam znati gde da ubaci novo kreiranu instrukciju i sve što treba da uradi je da odredi koju naredbu treba da kreira.
Naredbe koje ćemo mi koristiti:

```
// Pozivi funkcija
 Builder.CreateFAdd(l, d, "addtmp");
 Builder.CreateFSub(l, d, "subtmp");
 Builder.CreateFMul(l, d, "multmp");
 Builder.CreateFDiv(l, d, "divtmp");
// Gde prvi argument u pozivu funkcija predstavlja levu stranu
   izraza, drugi desnu, a treci ime pomocne promenljive u koje ce
   smestati rezultati.
```

F u imenu funkcije Builder.Create**F**.. je za FLOAT!

Napomena: LLVM neće sve pomoćne promenljive nazvati isto *(što bi bila greška jer sve promenljive moraju imati različita imena)*. Ako se u nekom bloku dva puta pojavljuje addtmp, LLVM će automatski svakoj sledećoj novoj pomoćnoj promenljivoj obezbediti sufiks na kraju, koji će se uvećavati kako se kreira nova promenljiva *(addtmp, addtmp1, addtmp2...)*.

Evo kako to izgleda u klasi AddExprAST. Slično je i za ostale.

```
Value* AddExprAST::codegen() const {
    Value *l = _v[0]->codegen();
    Value *d = _v[1]->codegen();
    if(l == NULL || d == NULL)
        return NULL;
    return Builder.CreateFAdd(l, d, "addtmp");
}
```

3.4 Function Code Generation

Generisanje koda za prototipe funkcija i funkcije sadrži brojne detalje, pa će kod biti manje čitljiv nego za izraze.
Počnimo od prototipa funkcija. Prototip funkcije koristimo i za telo funkcija i za deklarisanje eksterne funkcije. Kod bi počeo ovako:

```
1 Function* PrototypeAST::codegen() const {
2     vector<Type*> d(_args.size(), Type::getDoubleTy(TheContext));
3     FunctionType* FT = FunctionType::get(Type::getDoubleTy(
          TheContext), d, false);
4     Function *F = Function::Create(FT, Function::ExternalLinkage,
          _f, TheModule);
5     ...
6 }
```

Prvo što treba da zapazimo je da funkcija vraća *Function** umesto *Value**, zato što prototip funkcije zaista govori o spoljnom interfejsu za funkciju, a ne o vrednosti koju izračunava izraz.
Prvo kreiramo vektor tipova duzine iste dužini niza argumenata, koji će sadržati informaciju kog je tipa svaki argument. U ovom slučaju double.

FunctionType mora da zna kog su mu tipa argumenti i kog mu je tipa povratna vrednost.
Poziv FunctionType::get(...): Prvi argument je tip povratne vrednosti funkcije, drugi je vektor tipova argumenata i treći je tipa bool koji kaže da li argumenti mogu ili ne mogu biti promenjeni.

I na kraju želimo da kreiramo IR funkciju koja odgovara našem prototipu.
Poziv funkcije Function::Create(...): Argumenti su redom: tip funkcije, koje linkovanje želimo, ime funkcije, modul u koji želimo ubaciti funkciju.
"ExternalLinkage" znači da se funkcija može definisati izvan trenutnog modula i/ili da se ona može pozivati iz funkcija izvan modula.

Treba još postaviti imena argumenata.

```
1  Function* PrototypeAST::codegen() const {
2      vector<Type*> d(_args.size(), Type::getDoubleTy(TheContext));
3      FunctionType* FT = FunctionType::get(Type::getDoubleTy(
           TheContext), d, false);
4      Function *F = Function::Create(FT, Function::ExternalLinkage,
           _f, TheModule);
5
6      unsigned i=0;
7      for(auto &Arg: F->args())
8          Arg.setName(_args[i++]);
9
10     return F;
11 }
```

Konačno, postavili smo ime svakog od argumenata funkcije prema imenima datim u prototipu. U ovom trenutku imamo funkciju prototipa bez tela.

Generisanje koda za **FunctionAST**:
TheModule sadrži funkcije i globalne promenljive. Zato ćemo početi sa pretragom TheModule-ove tablice simbola za postojeću verziju ove funkcije. U slučaju da je već kreiran pomoću izraza "extern". Ako TheModule::getFunction(...) vrati NULL onda ne postoji prethodna verzija, tako da cemo pozvati metod codegen().

```
1  Function* FunctionAST::codegen() const {
2      Function* F = TheModule->getFunction(_proto.getName());
3      if(F == NULL)
4          F = _proto.codegen();
5
6      if(F == NULL) {
7          cerr << "Nemoguce generisanje koda za funkciju " <<
               _proto.getName() << endl;
8          exit(EXIT_FAILURE);
9      }
10     if (!F->empty()) {
11       cerr << "Nemoguce je predefinisati fju " << _proto.getName
           () << endl;
12       exit(EXIT_FAILURE);
13     }
14 }
```

Metod **getName()** koji se poziva nad prototipom je naš metod definisan u klasi PrototypeAST.

```
1  class PrototypeAST {
2  public:
3      ...
4      string getName() const {
5        return _f;
6      }
7  private:
8      string _f;
9      ...
10 };
```

Sada smo došli do tačke gde je Builder već postavljen. Linija 3 u sledećem kodu će kreirati novi osnovni blok sa labelom **entry**. Linija 4 kaže Builder-u da nove instrukcije ubaci na kraj ovog osnovnog bloka. Zatim dodajemo argumente funkcije u mapu NamedValues.

```
1  Function* FunctionAST::codegen() const {
2      ...
3      BasicBlock* BB = BasicBlock::Create(TheContext, "entry", F);
4      Builder.SetInsertPoint(BB);
5
6      NamedValues.clear();
7      for(auto &Arg: F->args())
8          NamedValues[Arg.getName()] = &Arg;
9  }
```

Sada želimo da pozovemo metod codegen() za korenski izraz funkcije. Ako pretpostavimo da nema greške, onda ćemo kreirati LLVM ret instrukciju iz LLVM-a, koja upotpunjuje našu funkciju. Kada je funkcija kreirana, možemo pozvati funkciju **verifyFunction()**. Ona vrši niz provera generisanog koda kako bi utvrdila da li naš kompajler čini sve kako treba. Korišcenje ovoga je važno jer može uhvatiti puno grešaka. Kada je funkcija završena i potvrđena, vratimo je.

```
Function* FunctionAST::codegen() const {
    ...
    Value* RetVal;
    if((RetVal = _e->codegen())) {
        Builder.CreateRet(RetVal);

        verifyFunction(*F);
        return F;
    }
    else {
        F->eraseFromParent();
        return NULL;
    }
}
```

Treba pokriti i slučaj šta se dešava u slučaju greške.
Zbog jednostavnosti, sa greškama se borimo tako što ćemo celu funkciju u kojoj postoji neka greška izbrisati u potpunosti korišćenjem **eraseFromParent()**. Ovo ce dozvoliti korisniku da predefinise funkciju koju je prethodno možda netačno uneo. Da je nismo obrisali ona bi i dalje bila u tablici simbola sa telom i time sprečavala njeno moguće predefinisanje.
Ostalo nam je da implementiramo metod codegen() u klasi CallExprAST.

```
Value* CallExprAST::codegen() const {
    Function *F = TheModule->getFunction(_f);
    if(F == NULL) {
        cerr << "Poziv funkcije " << _f
             << "koja nije definisana." << endl;
        exit(EXIT_FAILURE);
    }

    if(F->arg_size() != _v.size()) {
        cerr << "Funkcija " << _f << " ocekuje "
             << F->arg_size() << " argumenata" ;
        exit(EXIT_FAILURE);
    }

    vector<Value *> a;
    for(unsigned i = 0; i<_v.size(); i++)
        a.push_back(_v[i]->codegen());

    return Builder.CreateCall(F, a, "calltmp");
}
```

Pri pozivu funkcije prvo pronalazimo tu funkciju u našem modulu. Ukoliko ne postoji to bi značilo da funkciju nismo ni definisali, prijavićemo gresku. Ukoliko je različit broj prosleđenih argumenata sa onim sačuvanim pri deklarisanju, takođe prijavljujemo grešku. Kreiramo instrukciju tako što pozovemo funkciju **Builder.CreateCall(..)**.

Takođe ne zaboravimo na izraze u test primerima oblika "23 + 43" i slično. Da bismo to omogućili ideja je da kreiramo neimenovane funkcije koje će vraćati te vrednosti izraza.

```
// U parseru dodati

Naredba
    : ...
    | E {
        PrototypeAST p("neimenovana_fja_"+to_string(counter++),
           vector<string>());
        FunctionAST f(p, $1);
        f.codegen();
    }
    ;
```

4 Podrška za optimizator

4.1 Trivial Constant Folding

Napravite sledeći test primer.

```
def f(x)
    1 + 2 + x;
```

Pri pokretanju kaleidoskopa na ovom test primeru dobijamo sledeće.

```
; ModuleID = 'Moj modul'
define double @f(double %x) {
entry:
  %addtmp = fadd double 3.000000e+00, %x
  ret double %addtmp
}
```

Primećujemo da je umesto $1 + 2$ već sračunato 3.
Constant folding na ovom primeru je veoma jednostavan jer je operator + levo rekurzivan. Dovoljno je proveriti da su u levom i desnom podstablu konstante i onda primeniti taj operator, a izračunati rezultat zameniti umesto čvora tog operatora. Pa je zato u mnogim jezicima podrazumevano ugrađen ako je u ovakvom obliku.
Sa LLVM-om ova podrška nam nije potrebna u AST-u. Pošto svi pozivi prolaze kroz LLVM IR Builder, on je zadužen da proveri postoji li mogućnost preklapanja konstanti i ako postoji vrati konstantu umesto kreiranja instrukcije! Dok za $x+1+2$ ovo neće biti slučaj, jer 1 i 2 nisu podstabla istog operatora plus.

Ali ako napravimo sledeći test primer, vidimo da Builder ima ograničene mogućnosti i da neće prepoznati da je $3 + x$ i $x + 3$ isto.

```
def h(x)
    (1 + 2 + x) * (x + (1 + 2));
```

```
; ModuleID = 'Moj modul'
define double @h(double %x) {
entry:
  %addtmp = fadd double 3.000000e+00, %x
  %addtmp1 = fadd double %x, 3.000000e+00
  %multmp = fmul double %addtmp, %addtmp1
  ret double %multmp
}
```

Zato LLVM pruža širok spektar optimizacija koje možemo koristiti u obliku "prolaza" *(engl. passes)*.

4.2 Optimizacioni prolazi

LLVM dozvoljava programeru kompajlera da donese potpune odluke o tome koje optimizacije treba koristiti, u kojem redosledu i kojoj situaciji.

Kao konkretan primer, LLVM podrzava 2 tipa prolaza: globalni (u okviru celog modula) i na nivou jedne funkcije. U našem slučaju mi želimo da pokrenemo nekoliko optimizacija po funkciji kada korisnik unese tu funkciju.

Da bi se omogućile optimizacije po funkcijama, potrebno je kreirati **FunctionPassManager** koji će da čuva i organizuje željene optimizacije koje želimo pokrenuti. Jednom kada to imamo, možemo da dodamo skup optimizacija za pokretanje. Trebaće nam novi FunctionPassManager za svaki modul koji želimo da optimizujemo, tako da ćemo napisati funkciju u kojoj ćemo inicijalizovati i modul i pass manager-a.

```
FunctionPassManager *TheFPM;

void InitializeModuleAndPassManager() {
    TheModule = new llvm::Module("Moj Modul", TheContext);
    TheFPM = new FunctionPassManager(TheModule);

    // Do simple "peephole" optimizations and bit-twiddling
       optzns.
    TheFPM->add(createInstructionCombiningPass());
    // Reassociate expressions.
    TheFPM->add(createReassociatePass());
    // Eliminate Common SubExpressions.
    TheFPM->add(createGVNPass());
    // Simplify the control flow graph (deleting unreachable
       blocks, etc).
    TheFPM->add(createCFGSimplificationPass());

    TheFPM->doInitialization();
}
```

Ovaj kod inicijalizuje globalni modul TheModule i FunctionPassManager-a za taj modul. Kada ga kreiramo, kositimo niz poziva "add" sa LLVM optimizacionim prolazima.

Naravno neophodno je uključiti i sledeće:

```
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Transforms/Scalar.h"
using namespace llvm::legacy;
```

Sada kada je sve spremno moramo pozvati optimizator metodom **run()** nad kreiranom funkcijom u FunctionAST::codegen() ali pre nego što je vratimo.

```
Function* FunctionAST::codegen() const {
  ...
    Value* RetVal;
    if((RetVal = _e->codegen())) {
        Builder.CreateRet(RetVal);
        verifyFunction(*F);

        TheFPM->run(*F);

        return F;
    }
 ...
}
```

Takođe ne smemo zaboraviti u main funkciji parser.ypp zameniti liniju sa kreiranjem modula na poziv funkcije InitializeModuleAndPassManager.

```
// U parseru

int main() {
    InitializeModuleAndPassManager();
    yyparse();

    TheModule->dump();
    delete TheModule;
    return 0;
}
```

I sada kada pokrenemo kaleidoskop nad onim test primerom dobijamo sledeće.

```
; ModuleID = 'Moj Modul'
define double @h(double %x) {
entry:
  %addtmp = fadd double %x, 3.000000e+00
  %multmp = fmul double %addtmp, %addtmp
  ret double %multmp
}
```

5 Proširenje jezika

5.1 LLVM IR za If/Then/Else

Motivacija: Želimo da nas kaleidoskop omogući sledeće:

```
extern foo();
extern bar();
def baz(x) if x then foo() else bar();
```

Prvo što treba da uradimo je da proširimo lexer i parser, i naravno dodati u hijerarhiju klasa klase LessThenExprAST i GreaterThenExprAST, IfThenElse. Trivijalno.

```
// U ast.hpp

class GreaterThenExprAST: public InnerExprAST {
public:
    GreaterThenExprAST(ExprAST* e1, ExprAST* e2)
    : InnerExprAST(e1, e2) {}
    Value* codegen() const;
};

class IfThenElse: public InnerExprAST {
public:
    IfThenElse(ExprAST* e1, ExprAST* e2, ExprAST* e3)
    : InnerExprAST(e1, e2, e3) {}
    Value* codegen() const;
};
```

Sada je na redu da implementiramo metod codegen() za dodate klase. Trebamo dobiti sledeće:

```
; ModuleID = 'Moj Modul'

declare double @foo()

declare double @bar()

define double @baz(double %x) {
entry:
  %ifcond = fcmp one double %x, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:       ; preds = %entry
  %calltmp = call double @foo()
  br label %ifcont

else:       ; preds = %entry
  %calltmp1 = call double @bar()
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ %calltmp, %then ], [ %calltmp1, %else ]
  ret double %iftmp
```

```
}
```

Ulazni blok ("entry") ocenjuje uslovni izraz i upoređuje rezultat sa 0.0 koristeći instrukciju **fcmp one** (one je za ordered nonequal). Na osnovu ovog rezultata kod preskače ili na "then" ili na "else" blok.
Kada then/else blokovi završe, oba se odlaze u "ifcont" blok. Sada je pitanje kako će kod znati koju vrednost da vrati.
Odgovor na ovo pitanje uključuje važnu SSA operaciju: uvodimo veštačku **phi** funkciju. Phi funkcija zapamti iz kog bloka dolaze promenljive i uzima onu vrednost iz kog smo bloka došli.

5.2 Codegen za LessThen i GreaterThen

```
Value* LessThenExprAST::codegen() const {
    Value *l = _v[0]->codegen();
    Value *d = _v[0]->codegen();
    if(l == NULL || d == NULL)
        return NULL;

    Value* tmp = Builder.CreateFCmpULT(l, d, "lttmp");

    return Builder.CreateUIToFP(
            tmp, Type::getDoubleTy(TheContext), "booltmp");
}
Value* GreaterThenExprAST::codegen() const {
    Value *l = _v[0]->codegen();
    Value *d = _v[0]->codegen();
    if(l == NULL || d == NULL)
        return NULL;

    Value* tmp = Builder.CreateFCmpUGT(l, d, "gttmp");

    return Builder.CreateUIToFP(
            tmp, Type::getDoubleTy(TheContext), "booltmp");
}
```

LLVM instrukcije su ograničene strogim pravilima, na primer, levi i desni operatori dodate naredbe moraju imati isti tip a rezultat dodavanja mora odgovarati tipovima operanada. Pošto su sve vrednosti u kaleidoskopu double ovo je vrlo jednostavno za implementaciju zbira, razlike...
Sa druge strane, LLVM određuje da instrukcija **Builder.CreateFCmp..** uvek vraća vrednost "i1" (integer dužine 1 što predstavlja bool koji nemamo). Problem sa ovim je što kaleidoskop želi vrednost 0.0 ili 1.0. Da bismo dobili ovu semantiku, kombinujemo instrukciju Builder.CreateFCmp... sa instrukcijom **Builder.CreateUIToFP**.

5.3 Codegen za If/Then/Else

```
Value* IfThenElse::codegen() const {
    Value* CondV = _v[0]->codegen();
        if(CondV == NULL)
            return NULL;
    Value* tmp = Builder.CreateFCmpONE(CondV, ConstantFP::get(
       TheContext, APFloat(0.0)), "ifcond");
    ...
}
```

Ovo smo već videli. Izračunamo vrednost za uslov i upoređujemo je sa nulom.

```
Value* IfThenElse::codegen() const {
   ...
  Function *F = Builder.GetInsertBlock()->getParent();
  BasicBlock *ThenBB = BasicBlock::Create(TheContext, "then", F);
  BasicBlock *ElseBB = BasicBlock::Create(TheContext, "else");
  BasicBlock *MergeBB = BasicBlock::Create(TheContext, "ifcont");

  Builder.CreateCondBr(tmp, ThenBB, ElseBB);
    ...
}
```

Ovaj deo koda kreira osnovne blokove koji su povezani sa if/then/else. U našoj klasi IfThenElseExprAST nemamo ime funkcije u koju želimo smestiti blokove, zato i koristimo poziv funkcije **Builder.GetInsertBlock()->getParent()** i sada kada to imamo, kreiramo tri bloka.
Blok "ThenBB" je kao argument primio i funkciju F u koju želimo smestiti blok, ali ilustracije radi za sledeća dva bloka smo uradili na drugačiji način.
Blokovi ElseBB i MergeBB su kreirani ali za sad nisu ubačeni u funkciju.
I naravno, kreiramo uslovni skok pozivom funkcije **Builder.CreateCondBr(...)**.

```
Value* IfThenElse::codegen() const {
  ...
    Builder.SetInsertPoint(ThenBB);
    Value* ThenV = _v[1]->codegen();
    if(ThenV == NULL)
        return NULL;
    Builder.CreateBr(MergeBB);

    /* JAKO BITNA LINIJA */
    ThenBB = Builder.GetInsertBlock();

  ...
}
```

NAPOMENA: Označena bitna linija u kodu je objašnjena malo kasnije.
Nakon što smo ubacili uslovni skok želimo da pišemo u "then" bloku. To omogućavamo sa **Builder.SetInsertPoint**(ime bloka po kom želimo pisati). Izračunavamo vrednost u "then" bloku kao i inače. I kreiramo bezuslovni skok na "merge" blok.

```
Value* IfThenElse::codegen() const {
  ...
    F->getBasicBlockList().push_back(ElseBB);
    Builder.SetInsertPoint(ElseBB);
    Value* ElseV = _v[2]->codegen();
    if(ElseV == NULL)
        return NULL;
    Builder.CreateBr(MergeBB);

    /* JAKO BITNA LINIJA */
    ElseBB = Builder.GetInsertBlock();
  ...
}
```

Generisanje koda za "else" blok je u osnovi identično sem prve linije. Obratimo pažnju da smo pri kreiranju bloka "then" naveli funkciju u koju želimo pisati blok "then" a u okviru kreiranja bloka "else" nismo. Zato pozivom metoda **getBasicBlockList()** dobijamo listu blokova neke funkcije i onda dodamo željeni blok na kraj.

```
Value* IfThenElse::codegen() const {
  ...
    F->getBasicBlockList().push_back(MergeBB);
    Builder.SetInsertPoint(MergeBB);
    PHINode* PHI = Builder.CreatePHI(
        Type::getDoubleTy(TheContext), 2, "iftmp");

        PHI->addIncoming(ThenV, ThenBB);
        PHI->addIncoming(ElseV, ElseBB);

    return PHI;
}
```

Prve dve linije su poznate: prva dodaje blok "merge" u listu blokova željene funkcije i druga postavlja da se piše baš po našem željenom bloku.
Zatim je potrebno kreirati PHI čvor i dodati parove oblika blok/vrednost za PHI.

NAPOMENA: Da objasnimo one dve bitne linije koda.
Bez te dve linije koda, prolazi bi jednostavni test primeri, koji ne bi imali naprimer if konstrukciju u okviru if konstrukcije. Pa ovaj primer ne bi radio ono što bi mi hteli.

```
def f(x)
    if x < 0 then
       1
    else if x < 10 then
       2
    else
       3
;
```

Svaki poziv if/then/else kreira 3 bloka: entry, then i else.
(Nazovimo ih ENTRY1, THEN1, ELSE1).
Pa smo zato u našem primeru mi zapravo napravili 6 blokova. Phi čvor nam zapravo kaže odakle smo došli, odnosno iz kog bloka. U našem primeru je problem što je telo od prvog else malo komplikovanije, odnosno sadrži još jedan if/then/else, koji ima svoja 3 bloka: neka su to blokovi ENTRY2, THEN2, ELSE2. I kada pozovemo codegen() za telo tog ELSE1 on će kreirati te naše blokove ENTRY2, THEN2, ELSE2. Pa je logično da želimo da Phi čvor zna da se dolazi iz tih novih blokova, a ne starih. ZATO IH I AŽURIRAMO.

5.4 LLVM IR za FOR petlju

Motivacija: Želimo da naš kaleidoskop omogući:

```
extern putchard(x);
def printstar(n)
  for i = 1, i < n, 1.0 in
    putchard(42);   # ascii 42 = '*'

# print 100 '*' characters
printstar(100);
```

Ovde je interesantno što imamo novu varijablu("i" u ovom slučaju) koja iterira od početne vrednosti, sve dok važi neki uslov, povećavajući se za neku vrednost. Ako je vrednost koraka izostavljena, podrazumevaćemo da je 1.0. Sve dok je petlja tačna, ona izvršava svoje telo. S obzirom da petlja ne vraća ništa, definisaćemo je tako da vraća 0.0.

Sada je na redu da implementiramo metod codegen() za dodatu klasu. Trebamo dobiti sledeće:

```
declare double @putchard(double)

define double @printstar(double %n) {
entry:
  ; initial value = 1.0 (inlined into phi)
  br label %loop

loop:       ; preds = %loop, %entry
  %i = phi double [ 1.000000e+00, %entry ], [ %nextvar, %loop ]
  ; body
  %calltmp = call double @putchard(double 4.200000e+01)
  ; increment
  %nextvar = fadd double %i, 1.000000e+00

  ; termination test
  %cmptmp = fcmp ult double %i, %n
  %booltmp = uitofp i1 %cmptmp to double
  %loopcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %loopcond, label %loop, label %afterloop

afterloop:      ; preds = %loop
  ; loop always returns 0.0
```

```
23   ret double 0.000000e+00
24 }
```

5.5 Codegen za FOR petlju

Prvo što treba uraditi je naravno ne zaboraviti proširiti svoj lexer i gramatiku.
Ovako će izgledati klasa koja će nam predstavljati FOR petlju.

```
1 class ForIn: public InnerExprAST {
2 public:
3     ForIn(string s, ExprAST* e1, ExprAST* e2, ExprAST* e3,
          ExprAST* e4)
4     : InnerExprAST(e1, e2, e3, e4), _s(s)
5     {}
6     Value* codegen() const;
7 private:
8     string _s;
9 };
```

Primetimo da smo dodali konstruktor u InnerExprAST na osnovu 4 prosledjena izraza.
Vidimo da ova klasa čuva jedan string, koji bi predstavljao naše "i" u gornjem primeru.
Prosleđujemo izraz koji predstavlja pocetnu vrednost za "i", uslov, inkrement i telo for petlje.

Pa da počnemo sa metodom codegen().
Prvi korak je veoma jednostavan. Želimo da izračunamo vrednost na koju bi se brojačka promenljiva postavila.

```
1 Value* ForIn::codegen() const {
2     Value* StartVal = _v[0]->codegen();
3     if(StartVal == NULL)
4         return NULL;
5     ...
6 }
```

Dalje, želimo da kreiramo blok "loop" u koji ćemo smeštati instrukcije dobijene iz tela for petlje.

```
1  Value* ForIn::codegen() const {
2      ...
3   Function *F = Builder.GetInsertBlock()->getParent();
4   BasicBlock *LoopBB = BasicBlock::Create(TheContext, "loop", F);
5   Builder.CreateBr(LoopBB);
6
7   /* OPET BITNA LINIJA IZ ISTOG RAZLOGA KAO KOD IF/THEN/ELSE */
8   BasicBlock *BeforeLoopBB = Builder.GetInsertBlock();
9      ...
10 }
```

Sada želimo da pišemo instrukcije unutar bloka "loop".
Za početak, moramo videti šta se dešava sa brojačem. Jedna varijanta je pri prvoj iteraciji for petlje, kada nam vrednost za brojač dolazi iz entry bloka. A druga varijanta

je kada vrednost brojača menjamo kako se petlja izvršava, odnosno kada dolazi iz svog bloka, odnosno "loop" bloka. Čim imamo dve varijante, kreiramo Phi čvor. Pošto već znamo dolaznu vrednost za pocetnu vrednost, dodajemo u Phi čvor. Drugu, inkrementiranu vrednost, ne možemo dodati, jer je još nismo ni sračunali.

```
Value* ForIn::codegen() const {
    ...
  Builder.SetInsertPoint(LoopBB);
  PHINode* Variable = Builder.CreatePHI(
                      Type::getDoubleTy(TheContext), 2, "i");
  Variable->addIncoming(StartVal, BeforeLoopBB);
    ...
}
```

Naša petlja uvodi novu varijablu u tablicu simbola. To znači da naša tabela simbola sada može sadržati ili argumente funkcija ili varijable petlje. Pre nego što generešemo codegen() za telo for petlje, želimo dodati naš brojač u tablicu simbola. Medjutim, može da se desi da takva promenljiva, odnosno sa takvim imenom, već postoji, ali to neće biti problem jer ćemo staru vrednost sačuvati u privremenoj promenljivoj OldVal, dodati brojač, završtiti sve i posle vratiti OldVal u tablicu simbola.
Zatim smo izgenerisali telo for petlje.

```
Value* ForIn::codegen() const {
    ...
    Value* OldVal = NamedValues[_s];
    NamedValues[_s] = Variable;

    Value* BodyVal = _v[3]->codegen();
    if (BodyVal == NULL)
      return NULL;

    Value* IncVal = _v[2]->codegen();
    if (IncVal == NULL)
      return NULL;

    Value* NextVar = Builder.CreateFAdd(Variable, IncVal, "
       nextvar");
    ...
}
```

Takodje smo izračunali sledeću vrednost brojača dodavanjem vrednosti inkrementa ili 1.0 ako nije naveden. NextVar će biti nova vrednost.

```
Value* ForIn::codegen() const {
    ...
    // Compute the end condition.
    Value* CondV = _v[1]->codegen();
    if (CondV == NULL)
      return NULL;

    // Convert condition to a bool by comparing non-equal to 0.
    Value* Tmp = Builder.CreateFCmpONE(CondV,
                 ConstantFP::get(TheContext, APFloat(0.0)), "
                    loopcond");
    ...
}
```

```
Value* ForIn::codegen() const {
    ...
    BasicBlock *AfterLoopBB = BasicBlock::
                              Create(TheContext, "afterloop", F);
    Builder.CreateCondBr(Tmp, LoopBB, AfterLoopBB);

    LoopBB = Builder.GetInsertBlock();
    Builder.SetInsertPoint(AfterLoopBB);
    ...
}
```

Procenjujemo izlaznu vrednost petlje, da bi utvrdili da li iz petlje treba izaći.

Naravno, ne zaborimo da dodamo u PHI čvor onu drugu varijantu dolaska vrednosti brojača.
Vratimo staru vrednost koja je bila u tablici simbola.
I vratimo 0 kao povratnu vrednost for petlje.

```
Value* ForIn::codegen() const {
    ...
    Variable->addIncoming(NextVar, LoopBB);

    if (OldVal != NULL)
      NamedValues[_s] = OldVal;
    else
      NamedValues.erase(_s);

    return ConstantFP::get(TheContext, APFloat(0.0));
}
```

6 Korisnički definisani operatori

Sada bismo želeli da dodamo jedan lep operator ':', odnosno operator sekvenciranja. Omogućimo ovako nešto:

```
extern printd(x);

def f(x)
    printd(x) : printd(x + 1)
;
```

Ovo je krajnje jednostavno.
Kreiramo novu klasu - SeqExprAST

```
class SeqExprAST : public InnerExprAST {
public:
  SeqExprAST(ExprAST* e1, ExprAST* e2)
    :InnerExprAST(e1, e2)
  {}
  Value* codegen() const;
};
```

Zatim dodajemo metod codegen().

```
Value* SeqExprAST::codegen() const {
  Value *l = _v[0]->codegen();
  Value *d = _v[1]->codegen();
  if (!l || !d)
    return NULL;
  return d;
}
```

I naravno ne zaboravimo ažurirati parser!

7 Mutable Variables

7.1 Uvod

Kaleidoskop je interesantan ako ga posmatramo kao funkcionalni jezik, i činjenica da je funkcionalan olakšava generisanje LLVM IR, jer je veoma lako izgraditi LLVM IR u SSA obliku.
U ovom poglavlju želimo da definišemo promenljive, koje će moći da se promene.

7.2 Zašto je ovo problematično?

Da bismo shvatili zašto mutabilne varijable izazivaju složenost u SSA konstrukciji, pogledaćemo ovaj jednostavan primer u C-u.

```
1 int G, H;
2 int test(_Bool Condition) {
3   int X;
4   if (Condition)
5     X = G;
6   else
7     X = H;
8   return X;
9 }
```

U ovom primeru imamo promenljivu X, čija vrednost zavisi od toga odakle smo došli do nje. Pošto postoje dve različite moguće vrednosti za X pre instrukcije za povratak (odnosno return), mi bismo ubacili Phi čvor, kao i do sad.
I u ovakvom slučaju bismo želeli da to izgleda ovako:

```
1  @G = weak global i32 0   ; type of @G is i32*
2  @H = weak global i32 0   ; type of @H is i32*
3
4  define i32 @test(i1 %Condition) {
5  entry:
6    br i1 %Condition, label %cond_true, label %cond_false
7
8  cond_true:
9    %X.0 = load i32* @G
10   br label %cond_next
11
12 cond_false:
13   %X.1 = load i32* @H
14   br label %cond_next
15
16 cond_next:
17   %X.2 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
18   ret i32 %X.2
19 }
```

load je instrukcija koja učitava promenljivu sa neke adrese.
Ali ono čemu težimo je kako da ne iskoristimo PHI čvor.

7.3 Memorija u LLVM

Trik je da LLVM zahteva da sve vrednosti registara budu u SSA obliku, dok ne zahteva da memorijski objekti budu u SSA.
Ono što mi želimo je da napravimo stekovsku promenljivu, i onda da koristimo samo ovakve promenljive. Ali, ovakav pristup je veoma skup, zbog konstantnog čitanja iz memorije i pisanja po memoriji. Pa ćemo zato iskoristiti neku optimizaciju, koja će rešiti ovaj problem.

Dakle, u LLVM-u, svi pristupi memoriji su eksplicitno zadati isntrukcijama load/store. Obratiti pažnju na prethodni primer. Tip globalnih varijabli @G i @X je zapravo "i32 *" iako su varijable definisane kao "i32". Ovo znači da je @G zapravo adresa na kom se čuva neka promenljiva. Stekovske varijable će biti kreirane na sličan način, umesto deklarisanja sa globalnim definicijama, koristićemo LLVM instrukciju **alloca**.

```
define i32 @example() {
entry:
  %X = alloca i32           ; type of %X is i32*.
  ...
  %tmp = load i32* %X       ; load the stack value %X from the
     stack.
  %tmp2 = add i32 %tmp, 1   ; increment it
  store i32 %tmp2, i32* %X  ; store it back
  ...
```

Naredni primer pokazuje kako možemo deklarisati i manipulisati stekovskom promenljivom u LLVM IR.

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  %X = alloca i32           ; type of %X is i32*.
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  store i32 %X.0, i32* %X   ; Update X
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  store i32 %X.1, i32* %X   ; Update X
  br label %cond_next

cond_next:
  %X.2 = load i32* %X       ; Read X
  ret i32 %X.2
}
```

Ovim smo otkrili način rešavanja proizvoljnih mutablinih varijabli bez potrebe stvaranja Phi čvorova uopšte. Znači:
1. Svaka promenljiva postaje stekovska promenljiva.
2. Svako čitanje promenljive postaje load instrukcija.
3. Svako ažuriranje promenljive postaje store instrukcija.
4. Uzimanje adrese promenljive je direktno korišćenje stekovske adrese.

Kao što sam već spomenuli, problem je veliki broj jednostavnih i uobičajenih operacija. Sreća za nas, pa LLVM ima optimizator koji se zove **mem2reg**, koji može da to sredi, ubacujući Phi čvorove gde smatra da treba.

```
$ llvm-as < example.ll | opt -mem2reg | llvm-dis
```

7.4 Mutable promenljive u Kaleidoskopu - motivacija

Ono što želimo dodati je operator dodele ("=") i mogućnost da definišemo nove promenljive. Motivacija:

```
# Define ':' for sequencing: as a low-precedence operator that
   ignores operands
# and just returns the RHS.
def binary : 1 (x y) y;

# Recursive fib, we could do this before.
def fib(x)
  if (x < 3) then
    1
  else
    fib(x-1)+fib(x-2);

# Iterative fib.
def fibi(x)
  var a = 1, b = 1, c in
  (for i = 3, i < x in
     c = a + b :
     a = b :
     b = c) :
  b;

# Call it.
fibi(10);
```

7.5 Mutable promenljive u Kaleidoskopu

Tablicom simbola u našem Kaleidoskopu se upravlja uz pomoć mape *"NamedValues"*. Ova mapa sadrži LLVM "Value *" objekat, koji sadrži double vrednosti promenljivih. Da bismo podržali mutabilne promenljive želimo ovo promeniti tako da čuva memorijske lokacije promenljivih.
U ovom trenutku naš Kaleidoskop podržava samo promenljive za dve stvari: dolazni argumenti funkcijama i induktivna promenljiva petlje (naprimer u for petlji). Dozvolićemo mutaciju ovih promenljivih.

Započinjemo transformaciju Kaleidoskopa tako što menjamo mapu NamedValues tako da mapira *AllocaInst** umesto *Value**.

```
map<string, AllocaInst*> NamedValues;
```

S obzirom da moramo da kreiramo ove allocas, koristićemo pomoćnu funkciju koja će osigurati da se allocas kreiraju u entry bloku odgovarajuće funkcije.

```

AllocaInst *CreateEntryBlockAlloca(Function *TheFunction, const
    string &VarName) {
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(), TheFunction->
      getEntryBlock().begin());
  return TmpB.CreateAlloca(Type::getDoubleTy(TheContext), 0,
      VarName.c_str());
}
```

Ovaj kod kreira IRBuilder objekat koji ukazuje na prvu naredbu ulaznog bloka. Zatim kreira alokaciju sa prosleđenim imenom i vraća je.

Sada svugde gde se koristila mapa NamedValues moramo nešto menjati.

Prvo pojavljivanje je u codegen metodi za klasu *VariableExprAST*.

```
Value* VariableExprAST::codegen() const {
    AllocaInst* Alloca = NamedValues[_v];
    if(Alloca == NULL) {
        cerr << "Nepostojeca promenljiva " << _v << endl;
        exit(EXIT_FAILURE);
    }
    return Builder.CreateLoad(Alloca)
}
```

Umesto ručnog ubacivanja promenljive _v u mapu, kreirali smo instrukciju load.

Sada pogledajmo metod codegen u klasi ForIn.
Sada, više nije potreban PHI čvor.

```
Value* ForIn::codegen() const {
  Value* StartVal = _v[0]->codegen();
  if(StartVal == NULL)
      return NULL;

  Function *F = Builder.GetInsertBlock()->getParent();
  BasicBlock *LoopBB = BasicBlock::Create(TheContext, "loop", F);

  AllocaInst* Alloca = CreateEntryBlockAlloca(F, _s);
  Builder.CreateStore(StartVal, Alloca);

  Builder.CreateBr(LoopBB);

  Builder.SetInsertPoint(LoopBB);
  AllocaInst* OldVal = NamedValues[_s];
  NamedValues[_s] = Alloca;
```

```
17
18    Value* BodyVal = _v[3]->codegen();
19    if(BodyVal == NULL)
20        return NULL;
21
22    Value* IncVal = _v[2]->codegen();
23    if(IncVal == NULL)
24        return NULL;
25
26    Value* CurrVal = Builder.CreateLoad(Alloca);
27    Value* NextVar = Builder.CreateFAdd(CurrVal, IncVal,"nextvar");
28    Builder.CreateStore(NextVar, Alloca);
29
30    Value* CondV = _v[1]->codegen();
31    if(CondV == NULL)
32        return NULL;
33
34    Value* Tmp = Builder.CreateFCmpONE(CondV,
35          ConstantFP::get(TheContext, APFloat(0.0)), "loopcond");
36
37    BasicBlock* AfterLoopBB = BasicBlock::Create(TheContext,
        "afterloop", F);
38    Builder.CreateCondBr(Tmp, LoopBB, AfterLoopBB);
39
40    LoopBB = Builder.GetInsertBlock();
41    Builder.SetInsertPoint(AfterLoopBB);
42
43    if(OldVal != NULL)
44        NamedValues[_s] = OldVal;
45    else
46        NamedValues.erase(_s);
47
48    return ConstantFP::get(TheContext, APFloat(0.0));
49 }
```

Za klasu Function potrebno je promeniti for petlju unutar nje.

```
1  Value* FunctionAST::codegen() const {
2      ...
3      NamedValues.clear();
4      for(auto &Arg: F->args()) {
5          AllocaInst* Alloca = CreateEntryBlockAlloca(F, Arg.
               getName());
6          NamedValues[Arg.getName()] = Alloca;
7          Builder.CreateStore(&Arg, Alloca);
8      }
9
10     ...
```

7.6 Novi operator dodele

Motivacija:

```
extern printd(x);

def test(x)
  printd(x) :
  x = 4 :
  printd(x);
```

Ovo neće biti komplikovano.
Želimo da kreiramo novu klasu AssignExprAST.

```
class AssignExprAST: public InnerExprAST {
public:
    AssignExprAST(string s, ExprAST* e)
    : InnerExprAST(e), _s(s) {}
    Value* codegen() const;
private:
    string _s;
};
```

Metod codegen je trivijalan. Naime, izračunamo vrednost koju želimo da pridružimo nekoj promenljivoj i učitamo je u mapu.

```
Value* AssignExprAST::codegen() const {
    Value* Val = _v[0]->codegen();
    if(Val == NULL)
        return NULL;

    AllocaInst* Alloca = NamedValues[_s];
    if(Alloca == NULL) {
        cerr << "Promenljiva " << _s << " ne postoji." << endl;
        exit(EXIT_FAILURE);
    }

    Builder.CreateStore(Val, Alloca);
    return Val;
}
```

7.7 Definisanje novih promenljivih

Takođe želimo da obezbedimo kreiranje novih promenljivih upotrebom ključne reči **var**. Motivacija:

```
def fib(x)
    if (x < 3) then
       1
    else
       fib(x-1) + fib(x-2)
;

def fibi(x)
  var a = 1, b = 1, c in
  (for i = 2, i < x in
     c = a + b :
     a = b :
     b = c) :
  b;
```

Prvo ne zaboraviti ažurirati lexer i parser. Ovde ćemo koristiti parove. Naime, parser za ovaj deo izgleda ovako:

```
%union {
    ...
    vector< pair<string, ExprAST*> > *v2;
    pair<string, ExprAST*> *p1;
}
%type <v2> NizInic
%type <p1> Inic

%%
...
E
    : ...
    | var_token NizInic in_token E {
        $$ = new VarExprAST(*$2, $4);
        delete $2;
    }
    ;
NizInic
    : NizInic ',' Inic {
        $$ = $1;
        $$->push_back(*$3);
        delete $3;
    }
    | Inic {
        $$ = new vector<pair<string, ExprAST*>>();
        $$->push_back(*$1);
        delete $1;
    }
    ;

```

```
Inic
    : id_token '=' E {
        $$ = new pair<string, ExprAST*>(*$1, $3);
        delete $1;
    }
    | id_token {
      $$ = new pair<string, ExprAST*>(*$1, new NumberExprAST(0));
        delete $1;
    }
    ;
%%
```

Klasa VarExprAST će izgledati ovako:

```
class VarExprAST: public InnerExprAST {
public:
    VarExprAST(vector<pair<string, ExprAST*>> v, ExprAST* e)
    : InnerExprAST(e), _vars(v) {}
    Value* codegen() const;
    ~VarExprAST();
private:
    VarExprAST(const VarExprAST&);
    VarExprAST& operator=(const VarExprAST&);
    vector< pair <string, ExprAST*> > _vars;
};
```

Metod codegen.
Prvo tražimo funkciju u kojoj se nalazimo, a zatim kreiramo vektor AllocaInst* elemenata sa starim vrednostima iz mape.
Za svaki element u nizu inicijalizacije promenljivih uzimamo ime promenljive i izračunatu vrednost. I čuvamo to u Alloca. Kada sačuvamo staru vrednost u mapi za tu promenljivu, postavimo novu vrednost na Alloca.

```
Value* VarExprAST::codegen() const {
    Function* F = Builder.GetInsertBlock()->getParent();
    vector<AllocaInst*> OldValues;

    for(unsigned i=0; i < _vars.size(); i++) {
        string VarName = _vars[i].first;
        AllocaInst* Alloca = CreateEntryBlockAlloca(F, VarName);
        Value* InitVal = _vars[i].second->codegen();
        if(InitVal == NULL)
            return NULL;
        Builder.CreateStore(InitVal, Alloca);

        OldValues.push_back(NamedValues[VarName]);
        NamedValues[VarName] = Alloca;
        ...
    }
```

Zatim generišemo codegen za telo i vratimo stare vrednosti.

```
    ...
    Value* Val = _v[0]->codegen();
    if(Val == NULL)
        return NULL;

    for(unsigned int i=0; i < _vars.size(); i++) {
        string VarName = _vars[i].first;
        if(OldValues[i])
            NamedValues[VarName] = OldValues[i];
        else
            NamedValues.erase(VarName);
    }
    return Val;
}
```

8 Kompajliranje do objektnog koda

LLVM ima izvornu podršku za unakrsnu kompilaciju. Možemo kompajlirati na arhitekturu naše trenutne mašine ili na druge. Ovde ciljamo na našu arhitekturu.

Ovako će izgledati main funkcija našeg parsera. Ne mora da se zna čemu ove funkcije služe u svrhu ispita, a ako vas ipak interesuje, pogledajte llvm tutorial.

```
// U parseru
extern llvm::Module* TheModule;
extern llvm::LLVMContext TheContext;

int main() {
  //yydebug = 1;

  InitializeModuleAndPassManager();

  yyparse();

  //TheModule->dump();

  auto TargetTriple = sys::getDefaultTargetTriple();

  LLVMInitializeAllTargetInfos();
  LLVMInitializeAllTargets();
  LLVMInitializeAllTargetMCs();
  LLVMInitializeAllAsmParsers();
  LLVMInitializeAllAsmPrinters();

  string Error;
  auto Target = TargetRegistry::lookupTarget(TargetTriple, Error)
     ;
  if (!Target) {
    cerr << Error;
```

```
    return 1;
  }

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

  TargetOptions opt;
  auto RM = Reloc::Model();
  auto TargetMachine = Target->createTargetMachine(TargetTriple,
     CPU, Features, opt, RM);

  TheModule->setDataLayout(TargetMachine->createDataLayout());
  TheModule->setTargetTriple(TargetTriple);

  string Filename = "output.o";
  error_code EC;
  raw_fd_ostream dest(Filename, EC, sys::fs::F_None);

  if (EC) {
    cerr << "Could not open file: " << EC.message();
    return 1;
  }

  legacy::PassManager pass;
  auto FileType = TargetMachine::CGFT_ObjectFile;

  if (TargetMachine->addPassesToEmitFile(pass, dest, FileType)) {
    cerr << "TargetMachine can't emit a file of this type";
    return 1;
  }

  pass.run(*TheModule);
  dest.close();

  delete TheModule;

  return 0;
}
```

Ne zaboraviti uključiti potrebne biblioteke u ast.hpp.

```
#include "llvm/Transforms/Scalar.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/Target/TargetOptions.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Support/FileSystem.h"

using namespace llvm;
```

9 Dodatno - Testiranje i linkovanje sa C kodom

Funkcije generisanog koda možemo testirati tako što ćemo je linkovati sa nekim C programom *test.c*. Neka je sledeći kod izlaz našeg kaleidoskopa koji želimo testirati za konkretne vrenosti (kod za $(a + b)^2$).

```
; ModuleID = 'Moj Modul'
define double @foo(double %a, double %b) {
entry:
  %multmp = fmul double %a, %a
  %multmp1 = fmul double 2.000000e+00, %a
  %multmp2 = fmul double %multmp1, %b
  %addtmp = fadd double %multmp, %multmp2
  %multmp3 = fmul double %b, %b
  %addtmp4 = fadd double %addtmp, %multmp3
  ret double %addtmp4
}
```

Njega ćemo sačuvati u *.ll* tekstualnoj datoteti, a potom od nje sa *llc* kompajlerom generisati asemblerski kod koji će biti u formatu *.s* sledećim komandama u terminalu:

```
$ ./kaleidoskop < test 2> test.ll
$ llc test.ll
```

U *test.c* napisaćemo deklaraciju funkcije *foo* i pozvati za neku konkretnu vrednost, a potom linkovati *test.c* sa *test.s* i dobiti izvršni fajl *a.out*
test.c:

```
#include <stdio.h>
double foo(double, double);
int main() {
  printf("%g\n", foo(2, 4));
  return 0;
}
```

Posle linkovanja i pokretanja dobijamo izračunatu vrednost.

```
$ clang test.c test.s
$ ./a.out
$ 36
```