```
#include "llvm/Analysis/Passes.h"
#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include "llvm/PassManager.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Transforms/Scalar.h"
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/ExecutionEngine/JIT.h"

#include <cctype>
#include <cstdio>
#include <map>
#include <string>
#include <vector>
#include <memory>
#include <string>

using namespace std;

#include <QFile>

using namespace llvm;

//===----------------------------------------------------------------------===//
// "Library" functions that can be "extern'd" from user code.
//===----------------------------------------------------------------------===//

/// putchard - putchar that takes a double and returns 0.
extern "C"
double putchard(double X) {
  putchar((char)X);
  return 0;
}

/// printd - printf that takes a double prints it as "%f\n", returning 0.
extern "C"
double printd(double X) {
```

```
  printf("%f\n", X);
  return 0;
}


//===----------------------------------------------------------------------===//
// Lexer
//===----------------------------------------------------------------------===//

// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
enum Token {
  tok_eof = -1,

  // commands
  tok_def = -2, tok_extern = -3,

  // primary
  tok_identifier = -4, tok_number = -5,

  // control
  tok_if = -6, tok_then = -7, tok_else = -8,
  tok_for = -9, tok_in = -10,

  // operators
  tok_binary = -11, tok_unary = -12,

  // var definition
  tok_var = -13
};

static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number

/// gettok - Return the next token from standard input.
static int gettok() {
  static int LastChar = ' ';

  // Skip any whitespace.
  while (isspace(LastChar))
    LastChar = getchar();

  if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
```

```
  IdentifierStr = LastChar;
  while (isalnum((LastChar = getchar())))
    IdentifierStr += LastChar;

  if (IdentifierStr == "def") return tok_def;
  if (IdentifierStr == "extern") return tok_extern;
  if (IdentifierStr == "if") return tok_if;
  if (IdentifierStr == "then") return tok_then;
  if (IdentifierStr == "else") return tok_else;
  if (IdentifierStr == "for") return tok_for;
  if (IdentifierStr == "in") return tok_in;
  if (IdentifierStr == "binary") return tok_binary;
  if (IdentifierStr == "unary") return tok_unary;
  if (IdentifierStr == "var") return tok_var;
  return tok_identifier;
}

if (isdigit(LastChar) || LastChar == '.') {   // Number: [0-9.]+
  std::string NumStr;
  do {
    NumStr += LastChar;
    LastChar = getchar();
  } while (isdigit(LastChar) || LastChar == '.');

  NumVal = strtod(NumStr.c_str(), 0);
  return tok_number;
}

if (LastChar == '#') {
  // Comment until end of line.
  do LastChar = getchar();
  while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

  if (LastChar != EOF)
    return gettok();
}

// Check for end of file.  Don't eat the EOF.
if (LastChar == EOF)
  return tok_eof;

// Otherwise, just return the character as its ascii value.
int ThisChar = LastChar;
```

```
  LastChar = getchar();
  return ThisChar;
}

//===----------------------------------------------------------------------===//
// Abstract Syntax Tree (aka Parse Tree)
//===----------------------------------------------------------------------===//
namespace {
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
  virtual Value *Codegen() = 0;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val) : Val(val) {}
  virtual Value *Codegen();
};

/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;
public:
  VariableExprAST(const std::string &name) : Name(name) {}
  const std::string &getName() const { return Name; }
  virtual Value *Codegen();
};

/// UnaryExprAST - Expression class for a unary operator.
class UnaryExprAST : public ExprAST {
  char Opcode;
  ExprAST *Operand;
public:
  UnaryExprAST(char opcode, ExprAST *operand)
    : Opcode(opcode), Operand(operand) {}
  virtual Value *Codegen();
};

/// BinaryExprAST - Expression class for a binary operator.
```

```
class BinaryExprAST : public ExprAST {
  char Op;
  ExprAST *LHS, *RHS;
public:
  BinaryExprAST(char op, ExprAST *lhs, ExprAST *rhs)
    : Op(op), LHS(lhs), RHS(rhs) {}
  virtual Value *Codegen();
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<ExprAST*> Args;
public:
  CallExprAST(const std::string &callee, std::vector<ExprAST*> &args)
    : Callee(callee), Args(args) {}
  virtual Value *Codegen();
};

/// IfExprAST - Expression class for if/then/else.
class IfExprAST : public ExprAST {
  ExprAST *Cond, *Then, *Else;
public:
  IfExprAST(ExprAST *cond, ExprAST *then, ExprAST *_else)
  : Cond(cond), Then(then), Else(_else) {}
  virtual Value *Codegen();
};

/// ForExprAST - Expression class for for/in.
class ForExprAST : public ExprAST {
  std::string VarName;
  ExprAST *Start, *End, *Step, *Body;
public:
  ForExprAST(const std::string &varname, ExprAST *start, ExprAST *end,
             ExprAST *step, ExprAST *body)
    : VarName(varname), Start(start), End(end), Step(step), Body(body) {}
  virtual Value *Codegen();
};

/// VarExprAST - Expression class for var/in
class VarExprAST : public ExprAST {
  std::vector<std::pair<std::string, ExprAST*> > VarNames;
  ExprAST *Body;
```

```
public:
  VarExprAST(const std::vector<std::pair<std::string, ExprAST*> > &varnames,
             ExprAST *body)
  : VarNames(varnames), Body(body) {}

  virtual Value *Codegen();
};

/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its argument names as well as if it is an operator.
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
  bool isOperator;
  unsigned Precedence;  // Precedence if a binary op.
public:
  PrototypeAST(const std::string &name, const std::vector<std::string> &args,
               bool isoperator = false, unsigned prec = 0)
  : Name(name), Args(args), isOperator(isoperator), Precedence(prec) {}

  bool isUnaryOp() const { return isOperator && Args.size() == 1; }
  bool isBinaryOp() const { return isOperator && Args.size() == 2; }

  char getOperatorName() const {
    assert(isUnaryOp() || isBinaryOp());
    return Name[Name.size()-1];
  }

  unsigned getBinaryPrecedence() const { return Precedence; }

  Function *Codegen();

  void CreateArgumentAllocas(Function *F);
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  PrototypeAST *Proto;
  ExprAST *Body;
public:
  FunctionAST(PrototypeAST *proto, ExprAST *body)
    : Proto(proto), Body(body) {}
```

```
  Function *Codegen();
};
} // end anonymous namespace

//===----------------------------------------------------------------------===//
// Parser
//===----------------------------------------------------------------------===//

/// CurTok/getNextToken - Provide a simple token buffer.  CurTok is the current
/// token the parser is looking at.  getNextToken reads another token from the
/// lexer and updates CurTok with its results.
static int CurTok;
static int getNextToken() {
  return CurTok = gettok();
}

/// BinopPrecedence - This holds the precedence for each binary operator that is
/// defined.
static std::map<char, int> BinopPrecedence;

/// GetTokPrecedence - Get the precedence of the pending binary operator token.
static int GetTokPrecedence() {
  if (!isascii(CurTok))
    return -1;

  // Make sure it's a declared binop.
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0) return -1;
  return TokPrec;
}

/// Error* - These are little helper functions for error handling.
ExprAST *Error(const char *Str) { fprintf(stderr, "Error: %s\n", Str);return 0;}
PrototypeAST *ErrorP(const char *Str) { Error(Str); return 0; }
FunctionAST *ErrorF(const char *Str) { Error(Str); return 0; }

static ExprAST *ParseExpression();

/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static ExprAST *ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;
```

```
  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return new VariableExprAST(IdName);

  // Call.
  getNextToken();  // eat (
  std::vector<ExprAST*> Args;
  if (CurTok != ')') {
    while (1) {
      ExprAST *Arg = ParseExpression();
      if (!Arg) return 0;
      Args.push_back(Arg);

      if (CurTok == ')') break;

      if (CurTok != ',')
        return Error("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  // Eat the ')'.
  getNextToken();

  return new CallExprAST(IdName, Args);
}

/// numberexpr ::= number
static ExprAST *ParseNumberExpr() {
  ExprAST *Result = new NumberExprAST(NumVal);
  getNextToken(); // consume the number
  return Result;
}

/// parenexpr ::= '(' expression ')'
static ExprAST *ParseParenExpr() {
  getNextToken();  // eat (.
  ExprAST *V = ParseExpression();
  if (!V) return 0;

  if (CurTok != ')')
```

```
    return Error("expected ')'");
  getNextToken();  // eat ).
  return V;
}

/// ifexpr ::= 'if' expression 'then' expression 'else' expression
static ExprAST *ParseIfExpr() {
  getNextToken();  // eat the if.

  // condition.
  ExprAST *Cond = ParseExpression();
  if (!Cond) return 0;

  if (CurTok != tok_then)
    return Error("expected then");
  getNextToken();  // eat the then

  ExprAST *Then = ParseExpression();
  if (Then == 0) return 0;

  if (CurTok != tok_else)
    return Error("expected else");

  getNextToken();

  ExprAST *Else = ParseExpression();
  if (!Else) return 0;

  return new IfExprAST(Cond, Then, Else);
}

/// forexpr ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression
static ExprAST *ParseForExpr() {
  getNextToken();  // eat the for.

  if (CurTok != tok_identifier)
    return Error("expected identifier after for");

  std::string IdName = IdentifierStr;
  getNextToken();  // eat identifier.

  if (CurTok != '=')
    return Error("expected '=' after for");
```

```
  getNextToken();  // eat '='.


  ExprAST *Start = ParseExpression();
  if (Start == 0) return 0;
  if (CurTok != ',')
    return Error("expected ',' after for start value");
  getNextToken();

  ExprAST *End = ParseExpression();
  if (End == 0) return 0;

  // The step value is optional.
  ExprAST *Step = 0;
  if (CurTok == ',') {
    getNextToken();
    Step = ParseExpression();
    if (Step == 0) return 0;
  }

  if (CurTok != tok_in)
    return Error("expected 'in' after for");
  getNextToken();  // eat 'in'.

  ExprAST *Body = ParseExpression();
  if (Body == 0) return 0;

  return new ForExprAST(IdName, Start, End, Step, Body);
}

/// varexpr ::= 'var' identifier ('=' expression)?
//                   (',' identifier ('=' expression)?)* 'in' expression
static ExprAST *ParseVarExpr() {
  getNextToken();  // eat the var.

  std::vector<std::pair<std::string, ExprAST*> > VarNames;

  // At least one variable name is required.
  if (CurTok != tok_identifier)
    return Error("expected identifier after var");

  while (1) {
    std::string Name = IdentifierStr;
```

```
    getNextToken();  // eat identifier.

    // Read the optional initializer.
    ExprAST *Init = 0;
    if (CurTok == '=') {
      getNextToken(); // eat the '='.

      Init = ParseExpression();
      if (Init == 0) return 0;
    }

    VarNames.push_back(std::make_pair(Name, Init));

    // End of var list, exit loop.
    if (CurTok != ',') break;
    getNextToken(); // eat the ','.

    if (CurTok != tok_identifier)
      return Error("expected identifier list after var");
  }

  // At this point, we have to have 'in'.
  if (CurTok != tok_in)
    return Error("expected 'in' keyword after 'var'");
  getNextToken();  // eat 'in'.

  ExprAST *Body = ParseExpression();
  if (Body == 0) return 0;

  return new VarExprAST(VarNames, Body);
}

/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
///   ::= ifexpr
///   ::= forexpr
///   ::= varexpr
static ExprAST *ParsePrimary() {
  switch (CurTok) {
  default: return Error("unknown token when expecting an expression");
  case tok_identifier: return ParseIdentifierExpr();
```

```
  case tok_number:     return ParseNumberExpr();
  case '(':            return ParseParenExpr();
  case tok_if:         return ParseIfExpr();
  case tok_for:        return ParseForExpr();
  case tok_var:        return ParseVarExpr();
  }
}

/// unary
///   ::= primary
///   ::= '!' unary
static ExprAST *ParseUnary() {
  // If the current token is not an operator, it must be a primary expr.
  if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
    return ParsePrimary();

  // If this is a unary operator, read it.
  int Opc = CurTok;
  getNextToken();
  if (ExprAST *Operand = ParseUnary())
    return new UnaryExprAST(Opc, Operand);
  return 0;
}

/// binoprhs
///   ::= ('+' unary)*
static ExprAST *ParseBinOpRHS(int ExprPrec, ExprAST *LHS) {
  // If this is a binop, find its precedence.
  while (1) {
    int TokPrec = GetTokPrecedence();

    // If this is a binop that binds at least as tightly as the current binop,
    // consume it, otherwise we are done.
    if (TokPrec < ExprPrec)
      return LHS;

    // Okay, we know this is a binop.
    int BinOp = CurTok;
    getNextToken();  // eat binop

    // Parse the unary expression after the binary operator.
    ExprAST *RHS = ParseUnary();
    if (!RHS) return 0;
```

```
    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec+1, RHS);
      if (RHS == 0) return 0;
    }

    // Merge LHS/RHS.
    LHS = new BinaryExprAST(BinOp, LHS, RHS);
  }
}

/// expression
///   ::= unary binoprhs
///
static ExprAST *ParseExpression() {
  ExprAST *LHS = ParseUnary();
  if (!LHS) return 0;

  return ParseBinOpRHS(0, LHS);
}

/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
///   ::= unary LETTER (id)
static PrototypeAST *ParsePrototype() {
  std::string FnName;

  unsigned Kind = 0; // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return ErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_unary:
```

```
  getNextToken();
  if (!isascii(CurTok))
    return ErrorP("Expected unary operator");
  FnName = "unary";
  FnName += (char)CurTok;
  Kind = 1;
  getNextToken();
  break;
case tok_binary:
  getNextToken();
  if (!isascii(CurTok))
    return ErrorP("Expected binary operator");
  FnName = "binary";
  FnName += (char)CurTok;
  Kind = 2;
  getNextToken();

  // Read the precedence if present.
  if (CurTok == tok_number) {
    if (NumVal < 1 || NumVal > 100)
      return ErrorP("Invalid precedecnce: must be 1..100");
    BinaryPrecedence = (unsigned)NumVal;
    getNextToken();
  }
  break;
}

if (CurTok != '(')
  return ErrorP("Expected '(' in prototype");

std::vector<std::string> ArgNames;
while (getNextToken() == tok_identifier)
  ArgNames.push_back(IdentifierStr);
if (CurTok != ')')
  return ErrorP("Expected ')' in prototype");

// success.
getNextToken();  // eat ')'.

// Verify right number of names for operator.
if (Kind && ArgNames.size() != Kind)
  return ErrorP("Invalid number of operands for operator");
```

```
  return new PrototypeAST(FnName, ArgNames, Kind != 0, BinaryPrecedence);
}

/// definition ::= 'def' prototype expression
static FunctionAST *ParseDefinition() {
  getNextToken();  // eat def.
  PrototypeAST *Proto = ParsePrototype();
  if (Proto == 0) return 0;

  if (ExprAST *E = ParseExpression())
    return new FunctionAST(Proto, E);
  return 0;
}

/// toplevelexpr ::= expression
static FunctionAST *ParseTopLevelExpr() {
  if (ExprAST *E = ParseExpression()) {
    // Make an anonymous proto.
    PrototypeAST *Proto = new PrototypeAST("", std::vector<std::string>());
    return new FunctionAST(Proto, E);
  }
  return 0;
}

/// external ::= 'extern' prototype
static PrototypeAST *ParseExtern() {
  getNextToken();  // eat extern.
  return ParsePrototype();
}

//===----------------------------------------------------------------------===//
// Code Generation
//===----------------------------------------------------------------------===//

static Module *TheModule;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, AllocaInst*> NamedValues;
static FunctionPassManager *TheFPM;

Value *ErrorV(const char *Str) { Error(Str); return 0; }

/// CreateEntryBlockAlloca - Create an alloca instruction in the entry block of
/// the function.  This is used for mutable variables etc.
```

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

Value *NumberExprAST::Codegen() {
  return ConstantFP::get(getGlobalContext(), APFloat(Val));
}

Value *VariableExprAST::Codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (V == 0) return ErrorV("Unknown variable name");

  // Load the value.
  return Builder.CreateLoad(V, Name.c_str());
}

Value *UnaryExprAST::Codegen() {
  Value *OperandV = Operand->Codegen();
  if (OperandV == 0) return 0;

  Function *F = TheModule->getFunction(std::string("unary")+Opcode);
  if (F == 0)
    return ErrorV("Unknown unary operator");

  return Builder.CreateCall(F, OperandV, "unop");
}

Value *BinaryExprAST::Codegen() {
  // Special case '=' because we don't want to emit the LHS as an expression.
  if (Op == '=') {
    // Assignment requires the LHS to be an identifier.
    VariableExprAST *LHSE = dynamic_cast<VariableExprAST*>(LHS);
    if (!LHSE)
      return ErrorV("destination of '=' must be a variable");
    // Codegen the RHS.
    Value *Val = RHS->Codegen();
    if (Val == 0) return 0;
```

```
    // Look up the name.
    Value *Variable = NamedValues[LHSE->getName()];
    if (Variable == 0) return ErrorV("Unknown variable name");

    Builder.CreateStore(Val, Variable);
    return Val;
  }

  Value *L = LHS->Codegen();
  Value *R = RHS->Codegen();
  if (L == 0 || R == 0) return 0;

  switch (Op) {
  case '+': return Builder.CreateFAdd(L, R, "addtmp");
  case '-': return Builder.CreateFSub(L, R, "subtmp");
  case '*': return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),
                                "booltmp");
  default: break;
  }

  // If it wasn't a builtin binary operator, it must be a user defined one. Emit
  // a call to it.
  Function *F = TheModule->getFunction(std::string("binary")+Op);
  assert(F && "binary operator not found!");

  Value *Ops[] = { L, R };
  return Builder.CreateCall(F, Ops, "binop");
}

Value *CallExprAST::Codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (CalleeF == 0)
    return ErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return ErrorV("Incorrect # arguments passed");
```

```
  std::vector<Value*> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->Codegen());
    if (ArgsV.back() == 0) return 0;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}

Value *IfExprAST::Codegen() {
  Value *CondV = Cond->Codegen();
  if (CondV == 0) return 0;

  // Convert condition to a bool by comparing equal to 0.0.
  CondV = Builder.CreateFCmpONE(CondV,
                                ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                                "ifcond");

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Create blocks for the then and else cases.  Insert the 'then' block at the
  // end of the function.
  BasicBlock *ThenBB = BasicBlock::Create(getGlobalContext(), "then", TheFunction);
  BasicBlock *ElseBB = BasicBlock::Create(getGlobalContext(), "else");
  BasicBlock *MergeBB = BasicBlock::Create(getGlobalContext(), "ifcont");

  Builder.CreateCondBr(CondV, ThenBB, ElseBB);

  // Emit then value.
  Builder.SetInsertPoint(ThenBB);

  Value *ThenV = Then->Codegen();
  if (ThenV == 0) return 0;

  Builder.CreateBr(MergeBB);
  // Codegen of 'Then' can change the current block, update ThenBB for the PHI.
  ThenBB = Builder.GetInsertBlock();

  // Emit else block.
  TheFunction->getBasicBlockList().push_back(ElseBB);
  Builder.SetInsertPoint(ElseBB);

  Value *ElseV = Else->Codegen();
```

```
  if (ElseV == 0) return 0;

  Builder.CreateBr(MergeBB);
  // Codegen of 'Else' can change the current block, update ElseBB for the PHI.
  ElseBB = Builder.GetInsertBlock();

  // Emit merge block.
  TheFunction->getBasicBlockList().push_back(MergeBB);
  Builder.SetInsertPoint(MergeBB);
  PHINode *PN = Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()), 2,
                                  "iftmp");

  PN->addIncoming(ThenV, ThenBB);
  PN->addIncoming(ElseV, ElseBB);
  return PN;
}

Value *ForExprAST::Codegen() {
  // Output this as:
  //   var = alloca double
  //   ...
  //   start = startexpr
  //   store start -> var
  //   goto loop
  // loop:
  //   ...
  //   bodyexpr
  //   ...
  // loopend:
  //   step = stepexpr
  //   endcond = endexpr
  //
  //   curvar = load var
  //   nextvar = curvar + step
  //   store nextvar -> var
  //   br endcond, loop, endloop
  // outloop:

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Create an alloca for the variable in the entry block.
  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
```

```
// Emit the start code first, without 'variable' in scope.
Value *StartVal = Start->Codegen();
if (StartVal == 0) return 0;

// Store the value into the alloca.
Builder.CreateStore(StartVal, Alloca);

// Make the new basic block for the loop header, inserting after current
// block.
BasicBlock *LoopBB = BasicBlock::Create(getGlobalContext(), "loop", TheFunction);

// Insert an explicit fall through from the current block to the LoopBB.
Builder.CreateBr(LoopBB);

// Start insertion in LoopBB.
Builder.SetInsertPoint(LoopBB);

// Within the loop, the variable is defined equal to the PHI node.  If it
// shadows an existing variable, we have to restore it, so save it now.
AllocaInst *OldVal = NamedValues[VarName];
NamedValues[VarName] = Alloca;

// Emit the body of the loop.  This, like any other expr, can change the
// current BB.  Note that we ignore the value computed by the body, but don't
// allow an error.
if (Body->Codegen() == 0)
  return 0;

// Emit the step value.
Value *StepVal;
if (Step) {
  StepVal = Step->Codegen();
  if (StepVal == 0) return 0;
} else {
  // If not specified, use 1.0.
  StepVal = ConstantFP::get(getGlobalContext(), APFloat(1.0));
}

// Compute the end condition.
Value *EndCond = End->Codegen();
if (EndCond == 0) return EndCond;

// Reload, increment, and restore the alloca.  This handles the case where
```

```
  // the body of the loop mutates the variable.
  Value *CurVar = Builder.CreateLoad(Alloca, VarName.c_str());
  Value *NextVar = Builder.CreateFAdd(CurVar, StepVal, "nextvar");
  Builder.CreateStore(NextVar, Alloca);

  // Convert condition to a bool by comparing equal to 0.0.
  EndCond = Builder.CreateFCmpONE(EndCond,
                          ConstantFP::get(getGlobalContext(), APFloat(0.0)),
                             "loopcond");

  // Create the "after loop" block and insert it.
  BasicBlock *AfterBB = BasicBlock::Create(getGlobalContext(), "afterloop", TheFunction);

  // Insert the conditional branch into the end of LoopEndBB.
  Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

  // Any new code will be inserted in AfterBB.
  Builder.SetInsertPoint(AfterBB);

  // Restore the unshadowed variable.
  if (OldVal)
    NamedValues[VarName] = OldVal;
  else
    NamedValues.erase(VarName);


  // for expr always returns 0.0.
  return Constant::getNullValue(Type::getDoubleTy(getGlobalContext()));
}

Value *VarExprAST::Codegen() {
  std::vector<AllocaInst *> OldBindings;

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // Register all variables and emit their initializer.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i) {
    const std::string &VarName = VarNames[i].first;
    ExprAST *Init = VarNames[i].second;

    // Emit the initializer before adding the variable to scope, this prevents
    // the initializer from referencing the variable itself, and permits stuff
    // like this:
```

```
    //  var a = 1 in
    //    var a = a in ...   # refers to outer 'a'.
    Value *InitVal;
    if (Init) {
      InitVal = Init->Codegen();
      if (InitVal == 0) return 0;
    } else { // If not specified, use 0.0.
      InitVal = ConstantFP::get(getGlobalContext(), APFloat(0.0));
    }

    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
    Builder.CreateStore(InitVal, Alloca);

    // Remember the old variable binding so that we can restore the binding when
    // we unrecurse.
    OldBindings.push_back(NamedValues[VarName]);

    // Remember this binding.
    NamedValues[VarName] = Alloca;
  }

  // Codegen the body, now that all vars are in scope.
  Value *BodyVal = Body->Codegen();
  if (BodyVal == 0) return 0;

  // Pop all our variables from scope.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i)
    NamedValues[VarNames[i].first] = OldBindings[i];

  // Return the body computation.
  return BodyVal;
}

Function *PrototypeAST::Codegen() {
  // Make the function type:  double(double,double) etc.
  std::vector<Type*> Doubles(Args.size(),
                             Type::getDoubleTy(getGlobalContext()));
  FunctionType *FT = FunctionType::get(Type::getDoubleTy(getGlobalContext()),
                                       Doubles, false);

  Function *F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule);

  // If F conflicted, there was already something named 'Name'.  If it has a
```

```
  // body, don't allow redefinition or reextern.
  if (F->getName() != Name) {
    // Delete the one we just made and get the existing one.
    F->eraseFromParent();
    F = TheModule->getFunction(Name);

    // If F already has a body, reject this.
    if (!F->empty()) {
      ErrorF("redefinition of function");
      return 0;
    }

    // If F took a different number of args, reject.
    if (F->arg_size() != Args.size()) {
      ErrorF("redefinition of function with different # args");
      return 0;
    }
  }

  // Set names for all arguments.
  unsigned Idx = 0;
  for (Function::arg_iterator AI = F->arg_begin(); Idx != Args.size();
       ++AI, ++Idx)
    AI->setName(Args[Idx]);

  return F;
}

/// CreateArgumentAllocas - Create an alloca for each argument and register the
/// argument in the symbol table so that references to it will succeed.
void PrototypeAST::CreateArgumentAllocas(Function *F) {
  Function::arg_iterator AI = F->arg_begin();
  for (unsigned Idx = 0, e = Args.size(); Idx != e; ++Idx, ++AI) {
    // Create an alloca for this variable.
    AllocaInst *Alloca = CreateEntryBlockAlloca(F, Args[Idx]);

    // Store the initial value into the alloca.
    Builder.CreateStore(AI, Alloca);

    // Add arguments to variable symbol table.
    NamedValues[Args[Idx]] = Alloca;
  }
}
```

```
Function *FunctionAST::Codegen() {
  NamedValues.clear();

  Function *TheFunction = Proto->Codegen();
  if (TheFunction == 0)
    return 0;

  // If this is an operator, install it.
  if (Proto->isBinaryOp())
    BinopPrecedence[Proto->getOperatorName()] = Proto->getBinaryPrecedence();

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);
  Builder.SetInsertPoint(BB);

  // Add all arguments to the symbol table and create their allocas.
  Proto->CreateArgumentAllocas(TheFunction);

  if (Value *RetVal = Body->Codegen()) {
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);

    // Optimize the function.
    TheFPM->run(*TheFunction);

    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();

  if (Proto->isBinaryOp())
    BinopPrecedence.erase(Proto->getOperatorName());
  return 0;
}

//===----------------------------------------------------------------------===//
// Top-Level parsing and JIT Driver
//===----------------------------------------------------------------------===//
```

```
static ExecutionEngine *TheExecutionEngine;

static void HandleDefinition() {
  if (FunctionAST *F = ParseDefinition()) {
    if (Function *LF = F->Codegen()) {
      fprintf(stderr, "Read function definition:");
      LF->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleExtern() {
  if (PrototypeAST *P = ParseExtern()) {
    if (Function *F = P->Codegen()) {
      fprintf(stderr, "Read extern: ");
      F->dump();
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (FunctionAST *F = ParseTopLevelExpr()) {
    if (Function *LF = F->Codegen()) {
      // JIT the function, returning a function pointer.
      void *FPtr = TheExecutionEngine->getPointerToFunction(LF);

      // Cast it to the right type (takes no arguments, returns a double) so we
      // can call it as a native function.
      double (*FP)() = (double (*)())(intptr_t)FPtr;
      fprintf(stderr, "Evaluated to %f\n", FP());
    }
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
```

```
}

/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (1) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:    return;
    case ';':        getNextToken(); break;  // ignore top-level semicolons.
    case tok_def:    HandleDefinition(); break;
    case tok_extern: HandleExtern(); break;
    default:         HandleTopLevelExpression(); break;
    }
  }
}
//===----------------------------------------------------------------------===//
// Main driver code.
//===----------------------------------------------------------------------===//

int main() {

  InitializeNativeTarget();

  LLVMContext &Context = getGlobalContext();

  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['='] = 2;
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.

  // Prime the first token.
  fprintf(stderr, "ready> ");
  getNextToken();

  // Make the module, which holds all the code.
  //std::unique_ptr<Module> Owner(new Module("my cool jit", Context));
  TheModule = new Module("my cool jit", Context);

  // Create the JIT.  This takes ownership of the module.
  std::string ErrStr;
```

```
TheExecutionEngine =
    EngineBuilder(TheModule).setErrorStr(&ErrStr).create();
if (!TheExecutionEngine) {
  fprintf(stderr, "Could not create ExecutionEngine: %s\n", ErrStr.c_str());
  exit(1);
}

FunctionPassManager OurFPM(TheModule);

// Set up the optimizer pipeline.  Start with registering info about how the
// target lays out data structures.
TheModule->setDataLayout(TheExecutionEngine->getDataLayout());
OurFPM.add(new DataLayoutPass(TheModule));
// Provide basic AliasAnalysis support for GVN.
OurFPM.add(createBasicAliasAnalysisPass());
// Promote allocas to registers.
OurFPM.add(createPromoteMemoryToRegisterPass());
// Do simple "peephole" optimizations and bit-twiddling optzns.
OurFPM.add(createInstructionCombiningPass());
// Reassociate expressions.
OurFPM.add(createReassociatePass());
// Eliminate Common SubExpressions.
OurFPM.add(createGVNPass());
// Simplify the control flow graph (deleting unreachable blocks, etc).
OurFPM.add(createCFGSimplificationPass());

OurFPM.doInitialization();

// Set the global so the code gen can use this.
TheFPM = &OurFPM;

// Run the main "interpreter loop" now.

MainLoop();

TheFPM = 0;

// Print out all of the generated code.
TheModule->dump();


//Write IR code
string test = "";
```

```
  raw_string_ostream rss(test);
  TheModule->print(rss, nullptr);
  QFile ft("test.txt");
  ft.open(QIODevice::WriteOnly);
  ft.write(QString(rss.str().c_str()).toLatin1());
  ft.close();


  //Write BitCode
  string test2 = "";
  raw_fd_ostream rss2("test.bc", test2, (sys::fs::OpenFlags)8);
  llvm::WriteBitcodeToFile(TheModule, rss2);

  return 0;
}


/*
Visited links

http://stackoverflow.com/questions/24899568/how-to-call-functions-from-external-dll-using-llvm-irbuilder,

http://stackoverflow.com/questions/25172258/exposing-internal-c-function-to-llvm-jitd-c,

https://groups.google.com/forum/#!topic/llvm-dev/nTsWdyuuuRA

...
*/
```