Topic: Kaleidoscope LLVM Tutorial

Reference: LLVM Tutorial

Team Members:

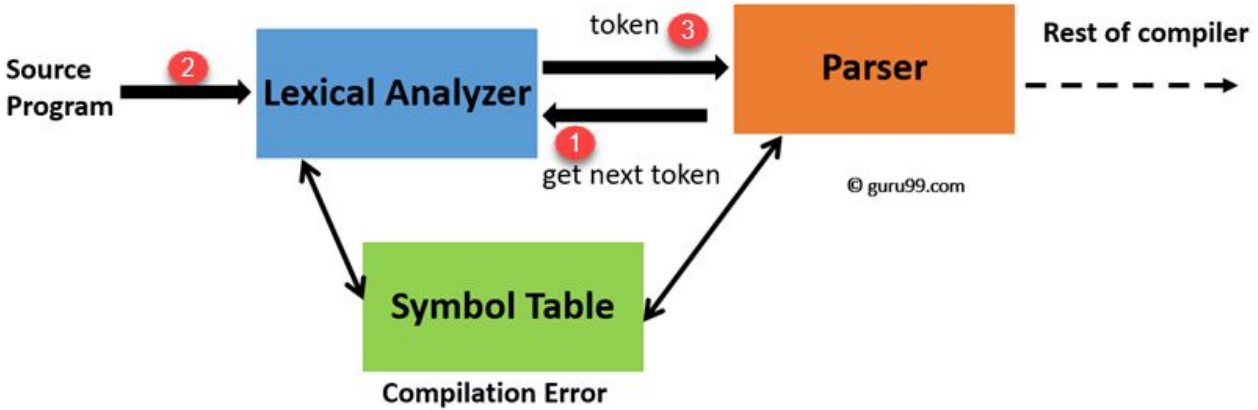
1. Siddharth Shankar
2. Rajat Lavekar
3. Monu Kumar Soyal
4. Raghav Magazine
5. Rahul Prajapat
6. Hrishikesh Pable
7. Pranav Talegaonkar
8. Shahil Patel
9. Harsh Chaudhari
10. Aryan Trimukhe

CH-1 - Introduction to Kaleidoscope Language

- Kaleidoscope is a procedural language that allows you to define functions, use conditionals, math, etc.
- The language supports features like if/then/else construct, for loop, user-defined operators, JIT compilation, and debug info.
- Only 64-bit floating-point type (double) is supported, and type declarations are not required, which gives the language a simple syntax.

Lexer Definition

- First step in processing a text file and recognizing tokens in the input
- Breaks the input into tokens, each with a token code and potentially some metadata.



Defining types of tokens

- Tokens in Kaleidoscope language are represented using an enum called *Token*. Each token returned by our lexer will either be one of the Token enum values

- For an unknown character like '+', the value returned will be its ASCII value.

- For all those tokens that are identifiers, we have a global string data type variable called IdentifierStr to store the name of the identifier.

Defining types of tokens

- For all those tokens that are numeric values, we have a global double data type variable called `NumVal` to store the numeric value.

- The values are assigned negative integers to avoid conflicts with ASCII or other character values, which typically fall within the range of 0-255.

```
// The lexer returns tokens [0-255] if it is an unknown character, otherwise one
// of these for known things.
// The Token enumeration has five defined values:
enum Token {
  // represents the end of the file token
  tok_eof = -1,

  // commands
  // represents a "def" command token
  tok_def = -2,
  tok_extern = -3,

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

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

Lexer Implementation - The `gettok()` function

- Function is called **gettok** to return the next token from standard input
- Ignores the whitespace between tokens and stores the last character read, but not processed

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

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

Lexer Implementation - Recognize identifiers and keywords

- Identifiers are sequences of letters (a-z, A-Z) followed by optional alphanumeric characters (a-z, A-Z, 0-9)

- Check if the last read character *LastChar* is an alphabet character, if true, it is assigned to the *IdentifierStr* variable as the first character of the identifier.

Lexer Implementation - Recognize identifiers and keywords

- Then it continues reading characters from input using *getchar*() and appending them to *IdentifierStr* as long as the characters are alphanumeric

- If the *IdentifierStr* does not match the keywords ("def" or "extern" in our case), the function returns *tok_identifier*, which represents an identifier token in the lexer.

```
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;
  return tok_identifier;
}
```

Lexer Implementation - Recognize numeric values

- C *strtod* function - to convert input to a numeric value that we store in *NumVal*
- First it checks if the last read character LastChar is a digit (0-9) or a decimal point ('.'), using the isdigit() function
- Loop reads characters from input using *getchar*() and appends them to *NumStr* as long as the characters are digits or decimal points, using the *isdigit*() function
- Converts *NumStr* to a floating-point number using *strtod* function, which takes a C-style string as input and returns the corresponding double value

```
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;
}
```

Lexer Implementation - Handle comments

- Handle comments by skipping to the end of the line and then return the next token

```
// '#' symbol typically indicates the start of a comment
if (LastChar == '#') {
// Comment until end of line.
do
  LastChar = getchar();
// effectively skips all characters in the current line after the '#' symbol, treating them
// as comments
while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

if (LastChar != EOF)
// ensures that any valid tokens following the comment are still properly processed and
// returned by the lexer.
  return gettok();
}
```

Lexer Implementation - Recognize unknown characters

- If the input doesn't match one of the previous cases, it is either an unknown character like '+'

```
  // 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;
  // This prepares LastChar for the next call to the lexer.
  LastChar = getchar();
  // Returns the ASCII value of the current character as a token
  return ThisChar;
}
```

CH-2 - Introduction to parser and AST

- Parser will use a combination of Recursive Descent Parsing and Operator-Precedence Parsing to parse the Kaleidoscope language

- We basically want one object for each construct in the language, and the AST should closely model the language

- In Kaleidoscope, we have expressions, a prototype, and a function object.

Expression AST node definitions

- The below code shows the definition of the base ExprAST class and one subclass which we use for numeric literals
- Note that the NumberExprAST class captures the numeric value of the literal as an instance variable.

```
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  // declares a virtual destructor for the ExprAST class
  // default - specifies that the destructor should have a default implementation
  virtual ~ExprAST() = default;
};

/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  // allows later phases of the compiler to know what the stored numeric value is.
  double Val;

public:
  NumberExprAST(double Val) : Val(Val) {}
};
```

Some more Expression AST node definitions

- Variables capture the variable name
- Binary operators capture their opcode (e.g. ‘+’)
- Calls capture a function name as well as a list of any argument expressions

```cpp
/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  // holds the name of the variable being referenced
  std::string Name;

public:
  VariableExprAST(const std::string &Name) : Name(Name) {}
};

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  // holds the binary operator like + or -
  char Op;
  // objects representing the left-hand side and right-hand side of the binary expression
  std::unique_ptr<ExprAST> LHS, RHS;

public:
  BinaryExprAST(char Op, std::unique_ptr<ExprAST> LHS,
                std::unique_ptr<ExprAST> RHS)
    : Op(Op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}
};

/// CallExprAST - Expression class for function calls.
class CallExprAST : public ExprAST {
  // holds the name of the function being called
  std::string Callee;
  // represents the arguments passed to the function
  std::vector<std::unique_ptr<ExprAST>> Args;

public:
  CallExprAST(const std::string &Callee,
              std::vector<std::unique_ptr<ExprAST>> Args)
    : Callee(Callee), Args(std::move(Args)) {}
};
```

Some more Expression AST node definitions

- Since all values are double precision floating point, the type of each argument doesn't need to be stored anywhere

- Functions are typed with just a count of their arguments

```cpp
/// PrototypeAST - This class represents the "prototype" for a function,
/// which captures its name, and its argument names (thus implicitly the number
/// of arguments the function takes).
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;

public:
  PrototypeAST(const std::string &Name, std::vector<std::string> Args)
    : Name(Name), Args(std::move(Args)) {}

 // function declared as a const member function that returns a reference to a constant
 // std::string object, and it promises not to modify the state of the object on which it is
 // called
  const std::string &getName() const { return Name; }
};

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  // a unique pointer to an instance of PrototypeAST class
  std::unique_ptr<PrototypeAST> Proto;
  // a unique pointer to an instance of ExprAST class
  std::unique_ptr<ExprAST> Body;

public:
  FunctionAST(std::unique_ptr<PrototypeAST> Proto,
              std::unique_ptr<ExprAST> Body)
    : Proto(std::move(Proto)), Body(std::move(Body)) {}
};
```

Some more Expression AST node definitions -

Additional Information:

- std::unique_ptr is a smart pointer provided by the C++ Standard Library that manages ownership of a dynamically allocated object and automatically deallocates the object when it goes out of scope. It is unique in that it cannot be copied, only moved.

- std::move is a utility function provided by the C++ Standard Library that enables moving the ownership of an object from one location to another without making a copy.

Basics of Parser - What we want to achieve ?

- Main idea here is that we want to parse something like “x+y” into an AST that could be generated with calls like shown below.

- The use of smart pointers helps manage memory ownership and ensures that memory is properly deallocated when it is no longer needed.

```
auto LHS = std::make_unique<VariableExprAST>("x");
//  The LHS smart pointer now owns the memory allocated for the VariableExprAST object.
auto RHS = std::make_unique<VariableExprAST>("y");
//  The RHS smart pointer now owns the memory allocated for the VariableExprAST object.
auto Result = std::make_unique<BinaryExprAST>('+', std::move(LHS), std::move(RHS));
// Result smart pointer of type std::unique_ptr<BinaryExprAST> now owns the memory allocated
for the BinaryExprAST object.
```

Basics of Parser - What we need to achieve ?

Additional Information:

- The std::move function is used to transfer ownership of the LHS and RHS smart pointers to the BinaryExprAST object, indicating that the ownership of the memory previously held by LHS and RHS is now transferred to Result.

- std::make_unique() is a C++11 and later standard library function that creates a new object on the heap and returns a std::unique_ptr that owns the object. It is a safer and more efficient way to create and manage ownership of dynamic objects compared to using raw pointers and new operator.

Basics of Parser - Defining helper routines

- We start by defining some basic helper routines - a simple token buffer around the lexer
- Allows us to look one token ahead at what the lexer is returning
- Every function in our parser will assume that CurTok is the current token that needs to be parsed

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

Basics of Parser - Defining helper routines

- Routines called LogError are simple routines that our parser will use to handle errors
- In both functions, the std::unique_ptr is used to manage ownership of the objects
- Returning a null pointer indicates that an error has occurred and no valid object is returned.

```
/// LogError* - These are little helper functions for error handling.
std::unique_ptr<ExprAST> LogError(const char *Str) {
  //  print the error message to the standard error stream
  fprintf(stderr, "Error: %s\n", Str);
  return nullptr;
}
//  used for reporting errors related to prototype abstract syntax trees
std::unique_ptr<PrototypeAST> LogErrorP(const char *Str) {
  LogError(Str);
  return nullptr;
}
```

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

  getNextToken();  // eat identifier.

  if (CurTok != '(') // Simple variable ref.
    return std::make_unique<VariableExprAST>(IdName);

  // Call
  getNextToken();  // eat (
  std::vector<std::unique_ptr<ExprAST>> Args;
  if (CurTok != ')') {
    while (true) {
      if (auto Arg = ParseExpression())
        Args.push_back(std::move(Arg));
      else
        return nullptr;

      if (CurTok == ')')
        break;

      if (CurTok != ',')
        return LogError("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }
  getNextToken();   // Eat the ')'.
  return std::make_unique<CallExprAST>(IdName, std::move(Args));
}
```

Basic Expression Parsing

- For each production in our grammar, we'll define a function which parses that production

```
/// numberexpr ::= number
static std::unique_ptr<ExprAST> ParseNumberExpr() {
  auto Result = std::make_unique<NumberExprAST>(NumVal);
  getNextToken(); // Consume the number
  return std::move(Result);
}
```

- The routine expects to be called, when the current is a *tok_number* token. It takes the current number value, creates a NumberExprAST node, advances the lexer to the next token, and finally returns.

Basic Expression Parsing

- A parenthesis operator is defined as :

```
/// parenexpr ::= '(' expression ')'
static std::unique_ptr<ExprAST> ParseParenExpr() {
  getNextToken(); // eat (.
  auto V = ParseExpression();
  if (!V)
    return nullptr;

  if (CurTok != ')')
    return LogError("expected ')'");
  getNextToken(); // eat ).
  return V;
}
```

- It can be observed how we use the *LogError* routine, and also recursive function *ParseExpression* (which calls *ParseParenExpr()* in its code later)

Basic Expression Parsing

- Now that we have all our simple-expression parsing logic in place, we can define a helper function to wrap it all together into 1 entry point :

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
static std::unique_ptr<ExprAST> ParsePrimary() {
  switch (CurTok) {
  default:
    return LogError("unknown token when expecting an expression");
  case tok_identifier:
    return ParseIdentifierExpr();
  case tok_number:
    return ParseNumberExpr();
  case '(':
    return ParseParenExpr();
  }
}
```

Binary Expression Parsing

- Binary expression can difficult to parse due to their ambiguous nature, hence we use an efficient way to handle this : Operator Precedence Parsing.

```
/// 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;
}
```

Operator Precedence Parsing

- This method of parsing assigns operators different precedence levels, and expressions are parsed based on the precedence of the operators.

- Eg. :- Expr = 2 + 3*4 , Here the multiplication operation has higher precedence, hence its performed first, followed by the addition operation.

Parsing the rest

- Handling of Function prototypes, both for extern declaration and function body definitions.

```
/// prototype
///   ::= id '(' id* ')'
static std::unique_ptr<PrototypeAST> ParsePrototype() {
  if (CurTok != tok_identifier)
    return LogErrorP("Expected function name in prototype");

  std::string FnName = IdentifierStr;
  getNextToken();

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

  // Read the list of argument names.
  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return LogErrorP("Expected ')' in prototype");

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

  return std::make_unique<PrototypeAST>(FnName, std::move(ArgNames));
}
```

Parsing the rest

- Function for *function* definition :

```
/// definition ::= 'def' prototype expression
static std::unique_ptr<FunctionAST> ParseDefinition() {
  getNextToken();  // eat def.
  auto Proto = ParsePrototype();
  if (!Proto) return nullptr;

  if (auto E = ParseExpression())
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  return nullptr;
}
```

The Driver

- We'll be defining a driver which invokes all the parsing sub bits with a top level dispatch loop.

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

Concluding Parser

- Till now, we have constructed a complete parser and AST builder , using this we have fully defined our minimal language .

- Now the executable will validate Kaleidoscope code and tell us if it is grammatically correct or invalid.

CH-3 Code generation to LLVM IR

- ★ Code Generation to LLVM builds upon the previous chapters and delves deeper into LLVM's Intermediate Representation (IR).

- ★ We will also cover how to define function prototypes and generate LLVM IR for function bodies.

Code Generation Setup

★ The section describes a simple setup for generating LLVM IR. It defines virtual code generation methods in each AST class, with a codegen() method to emit IR.

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

1

```
/// NumberExprAST - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;

public:
  NumberExprAST(double Val) : Val(Val) {}
  Value *codegen() override;
};
```

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

Value *LogErrorV(const char *Str) {
  LogError(Str);
  return nullptr;
}
```

Expression Code Generation

★ In this section we explore how to generate LLVM code for expression nodes in a simple version of the Kaleidoscope language.

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

Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (!V)
    LogErrorV("Unknown variable name");
  return V;
}
```

```
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;

  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(TheContext),
                                 "booltmp");
  default:
    return LogErrorV("invalid binary operator");
  }
}
```

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

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return LogErrorV("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())
      return nullptr;
  }

  return Builder->CreateCall(CalleeF, ArgsV, "calltmp");
}
```

Function Code Generation

★ This section describes the code generation for function prototypes and function definitions in LLVM IR.

```
Function *PrototypeAST::codegen() {
  // Make the function type:
double(double,double) etc.
  std::vector<Type*> Doubles(Args.size(),

Type::getDoubleTy(*TheContext));
  FunctionType *FT =

FunctionType::get(Type::getDoubleTy(*TheContext),
Doubles, false);

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

  // Set names for all arguments.
unsigned Idx = 0;
for (auto &Arg : F->args())
  Arg.setName(Args[Idx++]);

return F;
}
```

```
Function *FunctionAST::codegen() {
    // First, check for an existing function from a previous 'extern' declaration.
  Function *TheFunction = TheModule->getFunction(Proto->getName());

  if (!TheFunction)
    TheFunction = Proto->codegen();

  if (!TheFunction)
    return nullptr;

  if (!TheFunction->empty())
    return (Function*)LogErrorV("Function cannot be redefined.");

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

  // Record the function arguments in the NamedValues map.
  NamedValues.clear();
  for (auto &Arg : TheFunction->args())
    NamedValues[std::string(Arg.getName())] = &Arg;

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

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

    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();
  return nullptr;
}
```

Driver Changes and Closing Thoughts

★ In this particular section, we learn how to generate LLVM IR (Intermediate Representation) code for Kaleidoscope programs.

```
ready> 4+5;
Read top-level expression:
define double @0() {
entry:
  ret double 9.000000e+00
}

ready> def foo(a b) a*a + 2*a*b + b*b;
Read function definition:
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
}
```

```
ready> def bar(a) foo(a, 4.0) + bar(31337);
Read function definition:
define double @bar(double %a) {
entry:
  %calltmp = call double @foo(double %a, double 4.000000e+00)
  %calltmp1 = call double @bar(double 3.133700e+04)
  %addtmp = fadd double %calltmp, %calltmp1
  ret double %addtmp
}

ready> extern cos(x);
Read extern:
declare double @cos(double)

ready> cos(1.234);
Read top-level expression:
define double @1() {
entry:
  %calltmp = call double @cos(double 1.234000e+00)
  ret double %calltmp
}
```

CH-4 Adding JIT and Optimizer Support

- The objective is to optimize the IR code, so that the generated code executes efficiently
- Also we want to introduce JIT Compiler.The basic idea is that we want for Kaleidoscope is to have the user enter function bodies and immediately evaluate the top-level expressions they type in.
- For example, if they type in “1 + 2;”, we should evaluate and print out 3. If they define a function, they should be able to call it from the command line.

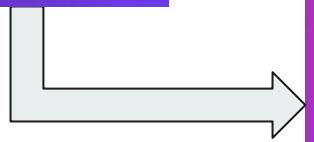
Changes in Code for Optimization

```
void InitializeModuleAndPassManager(void) {
  // Open a new context and module.
  TheModule = std::make_unique<Module>("my cool jit", *TheContext);

  // Create a new pass manager attached to it.
  TheFPM = std::make_unique<legacy::FunctionPassManager>(TheModule.get());

  // 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();
}
```

```
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;
}
```

Example (Optimization)

```
ready> def test(x) (1+2+x)*(x+(1+2));
ready> Read function definition:
define double @test(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
}
```



```
ready> def test(x) (1+2+x)*(x+(1+2));
ready> Read function definition:
define double @test(double %x) {
entry:
        %addtmp = fadd double %x, 3.000000e+00
        %multmp = fmul double %addtmp, %addtmp
        ret double %multmp
}
```

JIT Compiler

- First we need to prepare the environment to create code for the current native target and declare and initialize the JIT.
- This is done by calling some `InitializeNativeTarget\*` functions and adding a global variable `TheJIT`, and initializing it in `main`:

```
static std::unique_ptr<KaleidoscopeJIT> TheJIT;
...
int main() {
  InitializeNativeTarget();
  InitializeNativeTargetAsmPrinter();
  InitializeNativeTargetAsmParser();

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

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

  TheJIT = std::make_unique<KaleidoscopeJIT>();

  // Run the main "interpreter loop" now.
  MainLoop();

  return 0;
}
```

- After parsing and codegen succeed we add module containing the top-level expression to the JIT using **addModule()**, which triggers code generation for all the functions in the module, and accepts a **ResourceTracker** which can be used to remove the module from the JIT later.
- We use JIT's **lookup()** get a pointer to the final generated code
- We get the in-memory address of the __anon_expr function by calling **getaddress()**
- Finally we remove the module from the JIT when we're done to free the associated memory

```
static ExitOnError ExitOnErr;
...
static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (auto FnAST = ParseTopLevelExpr()) {
    if (FnAST->codegen()) {
      // Create a ResourceTracker to track JIT'd memory allocated to our
      // anonymous expression -- that way we can free it after executing.
      auto RT = TheJIT->getMainJITDylib().createResourceTracker();

      auto TSM = ThreadSafeModule(std::move(TheModule), std::move(TheContext));
      ExitOnErr(TheJIT->addModule(std::move(TSM), RT));
      InitializeModuleAndPassManager();

      // Search the JIT for the __anon_expr symbol.
      auto ExprSymbol = ExitOnErr(TheJIT->lookup("__anon_expr"));
      assert(ExprSymbol && "Function not found");

      // Get the symbol's address and 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)ExprSymbol.getAddress();
      fprintf(stderr, "Evaluated to %f\n", FP());

      // Delete the anonymous expression module from the JIT.
      ExitOnErr(RT->remove());
    }
```

- We want change the code so as to allow each function to live in its own module
- We add a new global, **FunctionProtos**, that holds the most recent prototype for each function and getfunction() to replace calls to **TheModule->getFunction()**.
- **getfunction()** searches **TheModule** for an existing function declaration, falling back to generating a new declaration from FunctionProtos if it doesn't find one.

```
static std::unique_ptr<KaleidoscopeJIT> TheJIT;

...

Function *getFunction(std::string Name) {
  // First, see if the function has already been added to the current module.
  if (auto *F = TheModule->getFunction(Name))
    return F;

  // If not, check whether we can codegen the declaration from some existing
  // prototype.
  auto FI = FunctionProtos.find(Name);
  if (FI != FunctionProtos.end())
    return FI->second->codegen();

  // If no existing prototype exists, return null.
  return nullptr;
}

...

Value *CallExprAST::codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = getFunction(Callee);

...

Function *FunctionAST::codegen() {
  // Transfer ownership of the prototype to the FunctionProtos map, but keep a
  // reference to it for use below.
  auto &P = *Proto;
  FunctionProtos[Proto->getName()] = std::move(Proto);
  Function *TheFunction = getFunction(P.getName());
  if (!TheFunction)
    return nullptr;
```

CH-5 - Extending the Language: Control Flow

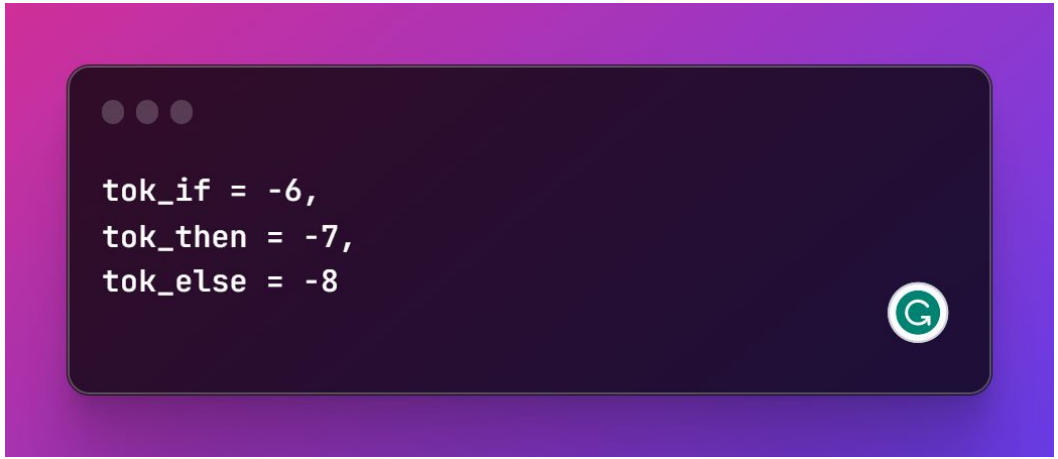
- In this chapter, we will be extending our Kaleidoscope to have an if/then/else expression plus a simple ‘for’ loop.

- To add support for if/then/else statements in Kaleidoscope, we need to make changes to the lexer, parser, AST, and LLVM code emitter.

Lexer Extension For If/Then/Else

We just need to add the new enums values for the relevant tokens.

- tok_if
- tok_then
- tok_else



AST Extensions for If/Then/Else

IfExprAST node provides a way for the compiler to generate code that evaluates a condition and returns a value based on the result of that condition.

```
/// IfExprAST - Expression class for if/then/else.
class IfExprAST : public ExprAST {
  std::unique_ptr<ExprAST> Cond, Then, Else;

public:
  IfExprAST(std::unique_ptr<ExprAST> Cond, std::unique_ptr<ExprAST> Then,
            std::unique_ptr<ExprAST> Else)
    : Cond(std::move(Cond)), Then(std::move(Then)), Else(std::move(Else)) {}

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

Parser Extension for If/Then/Else

Now we have relevant tokens coming from the parser and node to build.

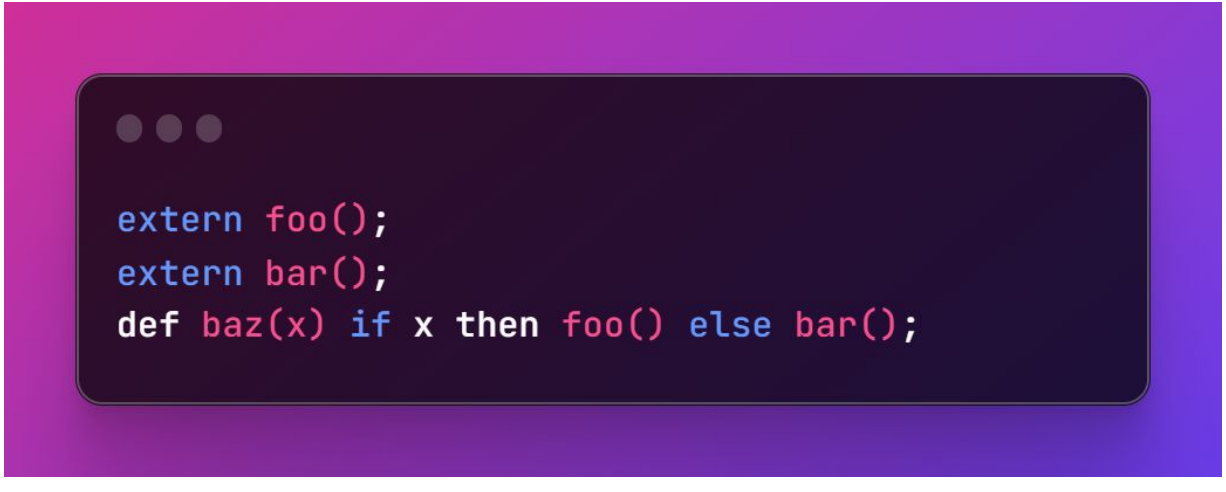
At the end, function creates a node of parsed condition

```
static std::unique_ptr<ExprAST> ParseIfExpr() {
  getNextToken();  // eat the if.

  // condition.
  auto Cond = ParseExpression();
  if (!Cond)
    return nullptr;

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

  auto Then = ParseExpression();
  if (!Then)
    return nullptr;
```

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

  getNextToken();

  auto Else = ParseExpression();
  if (!Else)
    return nullptr;

  return std::make_unique<IfExprAST>(std::move(Cond),std::move(Then),
                                     std::move(Else));
}
```

LLVM IR For If/Then/Else

Now, we have it parsing and building the AST, the final piece is adding LLVM code generation support.



LLVM IR Code

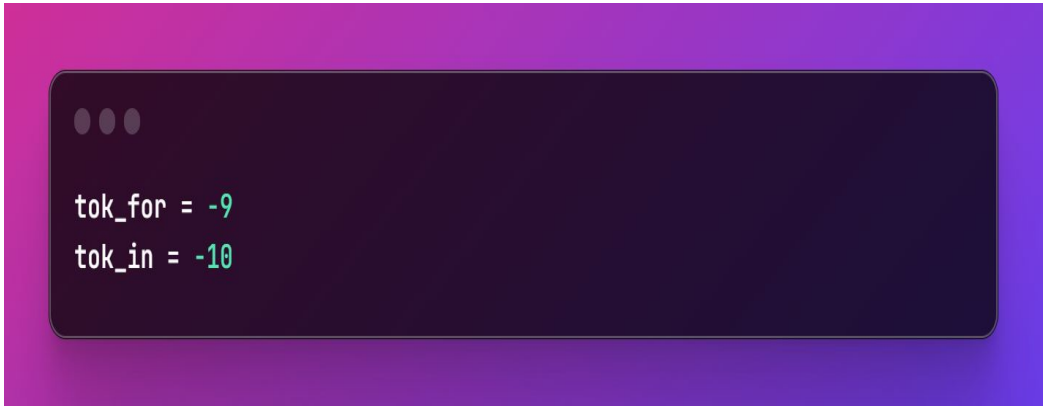
Now, this LLVM IR code can be executed by a machine and will produce the correct result when the "baz" function is called with an argument.

```
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
}
```

Lexer Extension For ‘for’ Loop

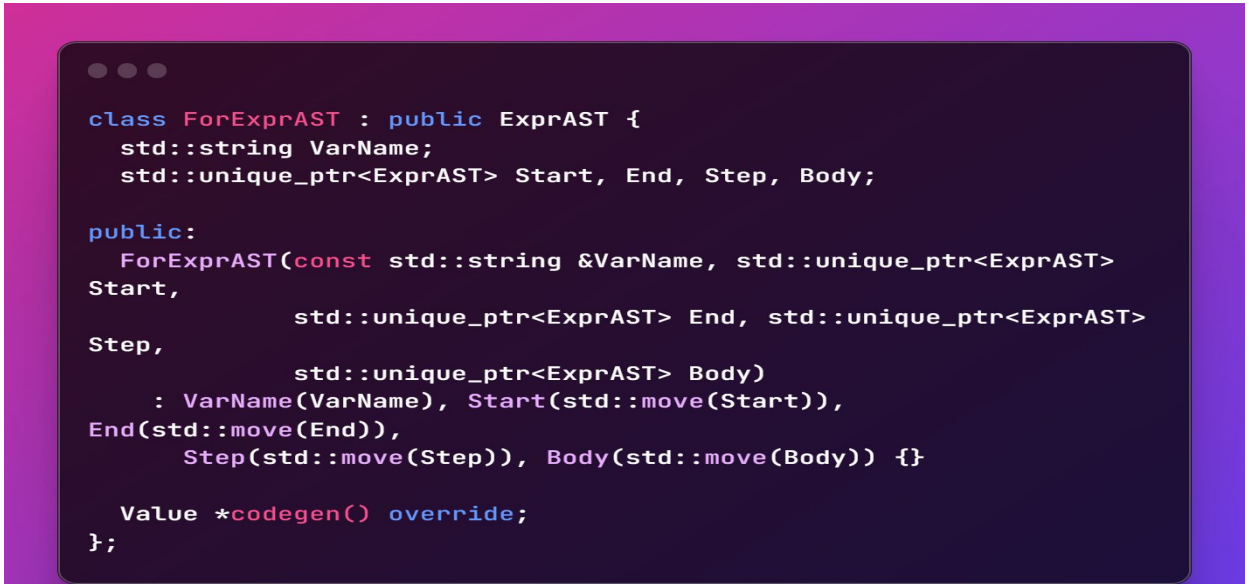
We just need to add the new enums values for the relevant tokens.

- tok _for = -9
- Tok_in = -10



AST Extension For ‘for’ Loop

It captures the variable name, the starting and ending values, the step size, and the body of the loop as separate expressions. This information is used to generate code for the 'for' loop.



```
class ForExprAST : public ExprAST {
  std::string VarName;
  std::unique_ptr<ExprAST> Start, End, Step, Body;

public:
  ForExprAST(const std::string &VarName, std::unique_ptr<ExprAST>
Start,
             std::unique_ptr<ExprAST> End, std::unique_ptr<ExprAST>
Step,
             std::unique_ptr<ExprAST> Body)
    : VarName(VarName), Start(std::move(Start)),
End(std::move(End)),
      Step(std::move(Step)), Body(std::move(Body)) {}

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

Parser Extension For ‘for’ Loop

- The Parser is responsible for reading the input code and turning it into an Abstract Syntax Tree (AST).
- It reads the input code and creates an instance of the ForExprAST class with the appropriate values

LLVM IR Code For ‘for’ loop

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

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

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

CH-6 User Defined Operators

- Until Now, we have developed a minimalistic fully functional Language
- But it has very few operators
- Adding Support for User Defined Operators

The Idea

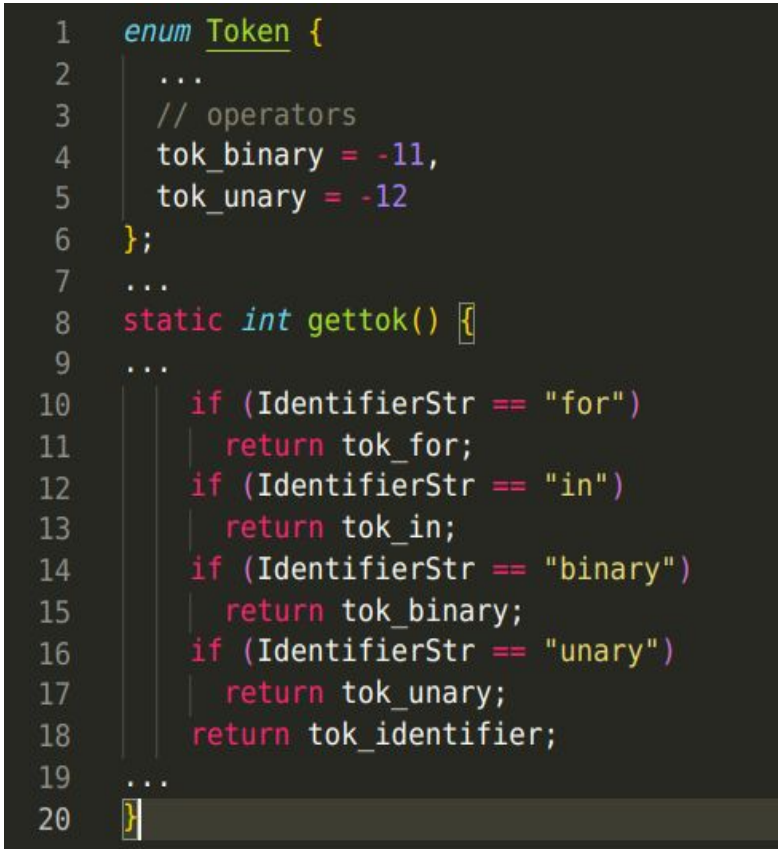
- More Powerful Operator Overloading in Kaleidoscope
- Hand-written parsers for User Defined Operators
- We will add two specific features:
  - Programmable Unary Operators
  - Programmable Binary Operators

```
# Logical unary not.
def unary!(v)
  if v then
    0
  else
    1;

# Define > with the same precedence as <.
def binary> 10 (LHS RHS)
  RHS < LHS;

# Binary "logical or", (note that it does not "short circuit")
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;

# Define = with slightly lower precedence than relationals.
def binary= 9 (LHS RHS)
  !(LHS < RHS | LHS > RHS);
```

User-defined Binary Operators

Adding Lexer Support:

- Tokens for binary and unary
  - tok_binary = -11
  - tok_unary = -12



```
enum Token {
  ...
  // operators
  tok_binary = -11,
  tok_unary = -12
};
...
static int gettok() {
...
    if (IdentifierStr == "for")
      return tok_for;
    if (IdentifierStr == "in")
      return tok_in;
    if (IdentifierStr == "binary")
      return tok_binary;
    if (IdentifierStr == "unary")
      return tok_unary;
    return tok_identifier;
...
}
```

User-defined Binary Operators

Parser Support:

- ASCII code as opcode for Binary Operators
- No new AST or parser support

User-defined Binary Operators

- Adding Definitions
- Eg: def binary| 5
- Function names parsed as prototype production in PrototypeAST AST Node
- Extend PrototypeAST AST Node

```cpp
/// 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, std::vector<std::string> Args,
               bool IsOperator = false, unsigned Prec = 0)
  : Name(Name), Args(std::move(Args)), IsOperator(IsOperator),
    Precedence(Prec) {}

  Function *codegen();
  const std::string &getName() const { return Name; }

  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; }
};
```

User-defined Binary Operators

- We have defined a way of representing the prototype of a user defined operators.
- Now, we need to parse it.

```
static std::unique_ptr<PrototypeAST> ParsePrototype() {
  std::string FnName;

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

  switch (CurTok) {
  default:
    return LogErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_binary:
    getNextToken();
    if (!isascii(CurTok))
      return LogErrorP("Expected binary operator");
    FnName = "binary";
    FnName += (char)CurTok;
    Kind = 2;
    getNextToken();
```

```
21      Kind = 2;
22      getNextToken();
23
24      // Read the precedence if present.
25      if (CurTok == tok_number) {
26        if (NumVal < 1 || NumVal > 100)
27          return LogErrorP("Invalid precedence: must be 1..100");
28        BinaryPrecedence = (unsigned)NumVal;
29        getNextToken();
30      }
31      break;
32    }
33
34    if (CurTok != '(')
35      return LogErrorP("Expected '(' in prototype");
36
37    std::vector<std::string> ArgNames;
38    while (getNextToken() == tok_identifier)
39      ArgNames.push_back(IdentifierStr);
40    if (CurTok != ')')
41      return LogErrorP("Expected ')' in prototype");
42
43    // success.
44    getNextToken();  // eat ')'.
45
46    // Verify right number of names for operator.
47    if (Kind && ArgNames.size() != Kind)
48      return LogErrorP("Invalid number of operands for operator");
49
50    return std::make_unique<PrototypeAST>(FnName, std::move(ArgNames), Kind != 0,
51                                          BinaryPrecedence);
52  }
```

User-defined Binary Operators

- Finally, we add codegen support for these binary operation.
- Here, we lookup for the operator in the symbol table and generate a function call to it (since these operators are just like normal functions).

```cpp
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;

  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(*TheContext),
                                 "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 = getFunction(std::string("binary") + Op);
  assert(F && "binary operator not found!");

  Value *Ops[2] = { L, R };
  return Builder->CreateCall(F, Ops, "binop");
}
```

User-defined Binary Operators

- Before codegening a function, we need to check if it is a binary operator
- If yes, then register it into the precedence table.

User-defined Unary Operators

- We currently don’t support unary operators; So need to add everything to support them.
- Added tokens.

```
1   /// UnaryExprAST - Expression class for a unary operator.
2   class UnaryExprAST : public ExprAST {
3     char Opcode;
4     std::unique_ptr<ExprAST> Operand;
5
6   public:
7     UnaryExprAST(char Opcode, std::unique_ptr<ExprAST> Operand)
8       : Opcode(Opcode), Operand(std::move(Operand)) {}
9
10    Value *codegen() override;
11  };
```

User-defined Unary Operators

- AST Node for Unary is same as of Binary with one child.
- Now we parse the Unary Operator.

```
1   static std::unique_ptr<ExprAST> ParseUnary() {
2     // If the current token is not an operator, it must be a primary expr.
3     if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
4       return ParsePrimary();
5
6     // If this is a unary operator, read it.
7     int Opc = CurTok;
8     getNextToken();
9     if (auto Operand = ParseUnary())
10      return std::make_unique<UnaryExprAST>(Opc, std::move(Operand));
11    return nullptr;
12  }
```

User-defined Unary Operators

- If we see a unary operator when parsing a primary operator, we eat the operator as a prefix and parse the remaining piece as another unary operator
- But we need to call this function from somewhere.
- So, we change the previous callers of ParsePrimary() to ParseUnary().

```
1   /// binoprhs
2   ///   ::= ('+' unary)*
3   static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,
4                                                 std::unique_ptr<ExprAST> LHS) {
5     ...
6       // Parse the unary expression after the binary operator.
7       auto RHS = ParseUnary();
8       if (!RHS)
9         return nullptr;
10    ...
11  }
12  /// expression
13  ///   ::= unary binoprhs
14  ///
15  static std::unique_ptr<ExprAST> ParseExpression() {
16    auto LHS = ParseUnary();
17    if (!LHS)
18      return nullptr;
19
20    return ParseBinOpRHS(0, std::move(LHS));
21  }
```

User-defined Unary Operators

- Next, we add parser support for prototypes to parse unary operator prototype.
- For this, we extend the binary operator code for parsing unary operators.
- And finally add the codegen function.

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

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

  switch (CurTok) {
  default:
    return LogErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_unary:
    getNextToken();
    if (!isascii(CurTok))
      return LogErrorP("Expected unary operator");
    FnName = "unary";
    FnName += (char)CurTok;
    Kind = 1;
    getNextToken();
    break;
  case tok_binary:
    ...
```

```
1   Value *UnaryExprAST::codegen() {
2     Value *OperandV = Operand->codegen();
3     if (!OperandV)
4       return nullptr;
5
6     Function *F = getFunction(std::string("unary") + Opcode);
7     if (!F)
8       return LogErrorV("Unknown unary operator");
9
10    return Builder->CreateCall(F, OperandV, "unop");
11  }
```

CH-7 Extending the Language: Mutable Variables

A mutable variable is a variable that can be changed during the program's execution.

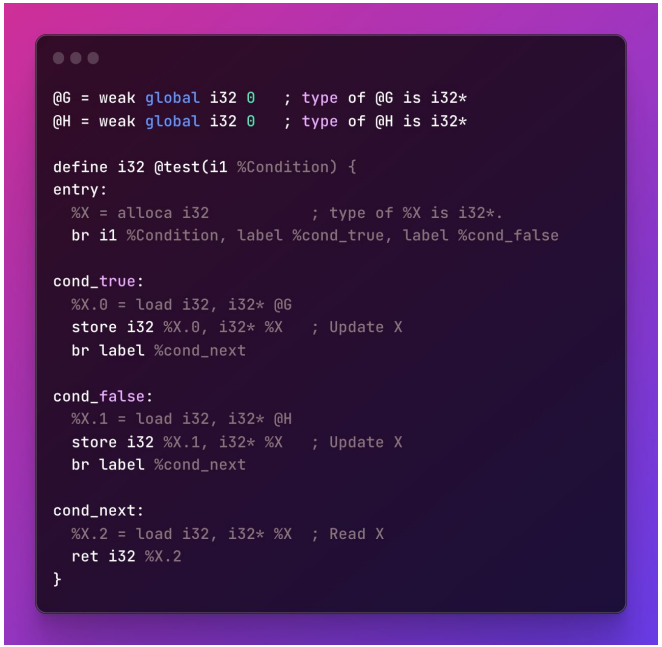
The issue here is that LLVM requires that its IR be in SSA (Single Static Assignment)form. However, SSA construction requires non-trivial algorithms and data structures, so it is inconvenient and wasteful for every front-end to have to reproduce this logic.

```
@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:
  br i1 %Condition, label %cond_true, label %cond_false

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

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

cond_next:
  %X.2 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.2
}
```

Memory in LLVM

In our previous example, we could rewrite the example to use the alloca technique to avoid using a PHI node:

we have now introduced a lot of stack traffic for very simple and common operations, a major performance problem. Fortunately for us, the LLVM optimiser has a highly-tuned optimisation pass named “mem2reg” that handles this case, promoting allocas like this into SSA registers, inserting Phi nodes as appropriate.



```
@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, i32* @G
  store i32 %X.0, i32* %X   ; Update X
  br label %cond_next

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

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

Mutable Variable in Kaleidoscope

We're going to add two features:

1. The ability to mutate variables with the '=' operator.

2. The ability to define new variables.

In order to mutate variables, we have to change our existing variables to use the "alloca trick". Once we have that, we'll add our new operator, then extend Kaleidoscope to support new variable definitions.

Adjusting Existing Variables for Mutation

An "AllocaInst" in LLVM is an instruction that allocates memory on the stack for a variable.

```
static std::map<std::string, AllocaInst*> NamedValues;
```

The function takes two arguments: a pointer to the function in which the AllocaInst will be created, and a string representing the name of the variable for which memory will be allocated.
The function returns a pointer to the created AllocaInst instruction.

```
/// 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(*TheContext),
nullptr,
                           VarName);
}
```

Now we need to update the things that define the variables to set up the alloca.

In our new scheme, variables live on the stack, so code generating a reference to them actually needs to produce a load from the stack slot:

The method returns a pointer to the generated Load instruction, which represents the value of the variable in LLVM IR.

```
Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  AllocaInst *A = NamedValues[Name];
  if (!A)
    return LogErrorV("Unknown variable name");

  // Load the value.
  return Builder->CreateLoad(A->getAllocatedType(), A,
Name.c_str());
}
```

The big difference is that we no longer have to construct a PHI node, and we use load/store to access the variable as needed.

```
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)
  return nullptr;

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

// Compute the end condition.
Value *EndCond = End->codegen();
if (!EndCond)
  return nullptr;

// Reload, increment, and restore the alloca.  This handles the
case where
// the body of the loop mutates the variable.
Value *CurVar = Builder->CreateLoad(Alloca->getAllocatedType(),
Alloca,
                                    VarName.c_str());
Value *NextVar = Builder->CreateFAdd(CurVar, StepVal,
"nextvar");
Builder->CreateStore(NextVar, Alloca);
...
```

To support mutable argument variables, we need to also make allocas for them. The code for this is also pretty simple:

```
Function *FunctionAST::codegen() {
  ...
  Builder->SetInsertPoint(BB);

  // Record the function arguments in the NamedValues map.
  NamedValues.clear();
  for (auto &Arg : TheFunction->args()) {
    // Create an alloca for this variable.
    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction,
Arg.getName());

    // Store the initial value into the alloca.
    Builder->CreateStore(&Arg, Alloca);

    // Add arguments to variable symbol table.
    NamedValues[std::string(Arg.getName())] = Alloca;
  }

  if (Value *RetVal = Body->codegen()) {
    ...
```

For each argument, we make an alloca, store the input value to the function into the alloca, and register the alloca as the memory location for the argument.

The final missing piece is adding the mem2reg pass, which allows us to get good codegen once again:

```
// Promote allocas to registers.
TheFPM->add(createPromoteMemoryToRegisterPass());
// Do simple "peephole" optimizations and bit-twiddling optzns.
TheFPM->add(createInstructionCombiningPass());
// Reassociate expressions.
TheFPM->add(createReassociatePass());
...
```

Mem2reg optimisation

Before:

```
define double @fib(double %x) {
entry:
  %x1 = alloca double
  store double %x, double* %x1
  %x2 = load double, double* %x1
  %cmptmp = fcmp ult double %x2, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:       ; preds = %entry
  br label %ifcont

else:       ; preds = %entry
  %x3 = load double, double* %x1
  %subtmp = fsub double %x3, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %x4 = load double, double* %x1
  %subtmp5 = fsub double %x4, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else
]
  ret double %iftmp
}
```

After:

```
define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00
  %booltmp = uitofp i1 %cmptmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:
  br label %ifcont

else:
  %subtmp = fsub double %x, 1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp5 = fsub double %x, 2.000000e+00
  %calltmp6 = call double @fib(double %subtmp5)
  %addtmp = fadd double %calltmp, %calltmp6
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else
]
  ret double %iftmp
}
```

New Assignment Operator

Now that all symbol table references are updated to use stack variables, we'll add the assignment operator.

The first step is to set a precedence:

```
int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['='] = 2;
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
```

We just need to implement codegen for the assignment operator. This looks like:

```
Value *BinaryExprAST::codegen() {
  // Special case '=' because we don't want to
emit the LHS as an expression.
  if (Op == '=') {
    // This assume we're building without RTTI
because LLVM builds that way by
    // default. If you build LLVM with RTTI this
can be changed to a
    // dynamic_cast for automatic error checking.
    VariableExprAST *LHSE =
static_cast<VariableExprAST*>(LHS.get());
    if (!LHSE)
      return LogErrorV("destination of '=' must be
a variable");
```

The thing is that it requires the LHS to be a variable.

Once we have the variable, codegen'ing the assignment is straightforward: we emit the RHS of the assignment, create a store, and return the computed value. Returning a value allows for chained assignments like "X = (Y = Z)".

Now that we have an assignment operator, we can mutate loop variables and arguments.

```
  // Codegen the RHS.
  Value *Val = RHS->codegen();
  if (!Val)
    return nullptr;

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

  Builder->CreateStore(Val, Variable);
  return Val;
}
...
```

User-defined Local Variable

The first step for adding our new ‘var/in’ construct is to extend the lexer. As before, this is pretty trivial, the code looks like this:

```
enum Token {
  ...
  // var definition
  tok_var = -13
...
}
...
static int gettok() {
...
    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;
...
```

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

public:
  VarExprAST(std::vector<std::pair<std::string,
std::unique_ptr<ExprAST>>> VarNames,
             std::unique_ptr<ExprAST> Body)
    : VarNames(std::move(VarNames)),
Body(std::move(Body)) {}

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

The next step is to define the AST node that we will construct. For var/in, it looks like this:

With this in place, we can define the parser pieces. The first thing we do is add it as a primary expression:

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
///   ::= ifexpr
///   ::= forexpr
///   ::= varexpr
static std::unique_ptr<ExprAST> ParsePrimary() {
  switch (CurTok) {
  default:
    return LogError("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();
  }
}
```

Next we define ParseVarExpr:

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

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

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

The first part of this code parses the list of identifier/expr pairs into the local VarNames vector.

```
while (true) {
  std::string Name = IdentifierStr;
  getNextToken();  // eat identifier.

  // Read the optional initializer.
  std::unique_ptr<ExprAST> Init;
  if (CurTok == '=') {
    getNextToken(); // eat the '='.

    Init = ParseExpression();
    if (!Init) return nullptr;
  }

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

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

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

Once all the variables are parsed, we then parse the body and create the AST node:

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

  auto Body = ParseExpression();
  if (!Body)
    return nullptr;

  return std::make_unique<VarExprAST>
(std::move(VarNames),

std::move(Body));
}
```

Now that we can parse and represent the code, we need to support emission of LLVM IR for it. This code starts out with:

```
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.get();
```

Basically it loops over all the variables, installing them one at a time. For each variable we put into the symbol table, we remember the previous value that we replace in OldBindings.

```
  // 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)
      return nullptr;
  } else { // If not specified, use 0.0.
    InitVal = ConstantFP::get(*TheContext,
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)
  return nullptr;
```

Once all the variables are installed in the symbol table, we evaluate the body of the var/in expression:

Finally, before returning, we restore the previous variable bindings:

```
  // 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;
}
```

The end result of all of this is that we get properly scoped variable definitions, and we even (trivially) allow mutation of them :).

With this, we completed what we set out to do. The mem2reg pass optimizes all of our stack variables into SSA registers, inserting PHI nodes where needed, and our front-end remains simple.

CH-8 Compiling to Object

This segment refers to the process of ***converting LLVM IR code*** generated from the Kaleidoscope language to a ***native object file*** format that can be executed on the target platform.

LLVM can easily cross-compile code to other architectures, in addition to compiling for the architecture of the current machine.
To specify the architecture that you want to target, we use a string called "***target triple***". This takes the form ***<arch><sub>-<vendor>-<sys>-<abi>***
Arch => architecture (e.g. x86_64, arm)
Sub => version
Vendor => Name of the vendor (apple, Nvidia, IBM, etc)
sys => Operating System (Linux, win32, Darwin, cuda, etc.)
Env => Environment (eabi, gnu, android, mach-O)
<abi> => Application Binary Interface

Target Machine:

we don’t need to hard-code a target triple to target the current machine.
LLVM provides `sys::getDefaultTargetTriple()`, which returns the ***target triple of the current device***.

The '**TargetMachine**' class gives a detailed description of the machine we are aiming for. If we want to focus on a particular feature or CPU, like SSE or Intel's Sandylake, we can do it using this class.

To see which features and CPUs that LLVM knows about, we can use `llc`

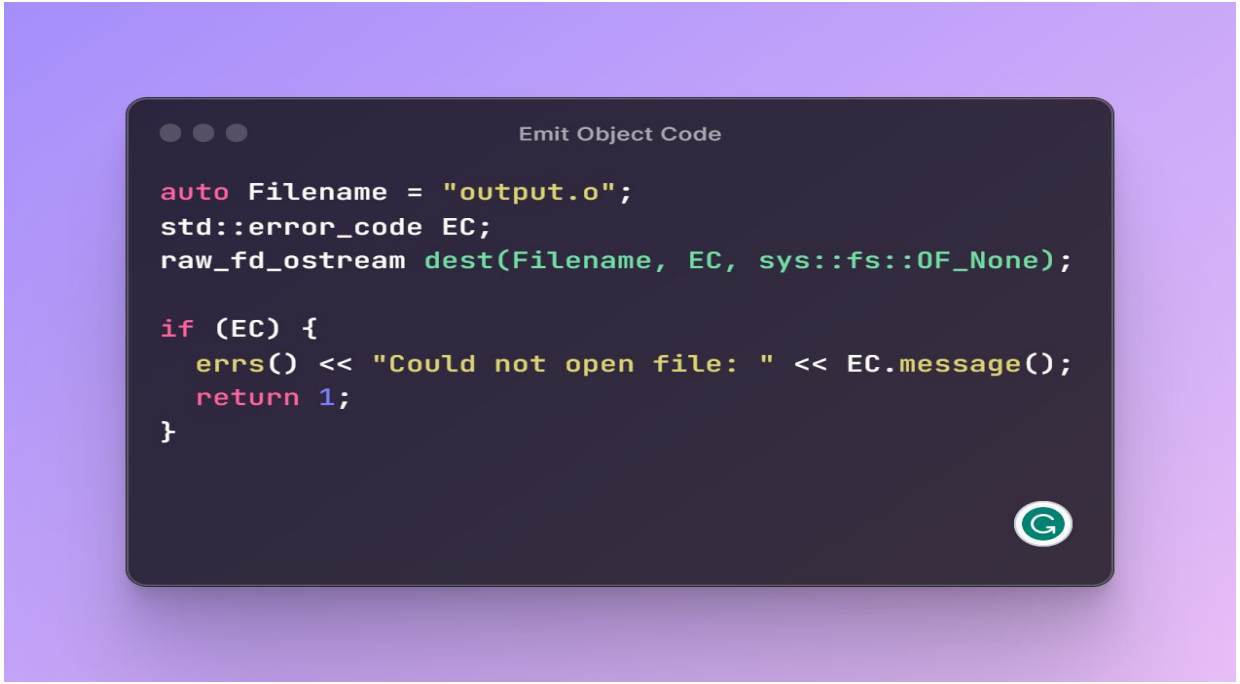
`auto CPU = "generic";` ← *declare a variable CPU of type auto, which will be used as a* ***parameter*** *when creating a TargeMachine object.*
`auto Features = "";` ← ***parameter*** *for creating a TargetMachine object.*
`TargetOptions opt;` ← *the* ***new object*** *of class TargetOptions*
`auto RM = std::optional<Reloc::Model>();` ← ***parameter*** *to create a TargetMachine object.*
`auto TargetMachine = Target->createTargetMachine(TargetTriple, CPU, Features, opt, RM);` ← *creates a new TargetMachine object using the* ***createTargetMachine*** *method of a Target object*

Configuring the Module

This is optional, but the frontend performance guide recommends this.
Optimizations benefit from knowing about the target and data layout.

In LLVM, these two lines of code help **configure a module** for a specific target machine. The first line (***setDataLayout***) creates a ***way for data to be stored in memory*** that works with the target machine, while the second line (***setTargetTriple***) ***identifies the target machine's architecture and operating system***. By doing this, LLVM can generate the right code for the target machine.

Emit Object Code:



This code sets up a **file stream** to write **binary** data to an output file named "*output.o*" and checks for errors during this process. If an error occurs, it prints an error message and returns with an **error code of 1**.

Define a pass that emits the object code, then we run that pass:

This code sets up a **pass manager** to manage a *sequence of passes* to **optimize the LLVM module**, adds passes to emit an object file to the 'pass' manager using the ***'TargetMachine' object***, applies the passes to the module, and ***writes the optimized module to an output file***. If an error occurs during this process, an error message is printed, and the program returns with an error code of 1.

CH-9 Adding Debug

- Source level debugging uses formatted data to translate from binary and the state of the machine back to the source code.
- In LLVM, DWARF is used as a format for representing types, source locations, and variable locations.
- Debugging via JIT is not possible, so the program needs to be compiled into a small standalone file.
- This means that we'll have a source file with a simple program written in Kaleidoscope rather than the interactive JIT.
- The limitation of having only one "top level" command at a time is necessary to reduce the number of changes required.

Why is this a Hard Problem ?

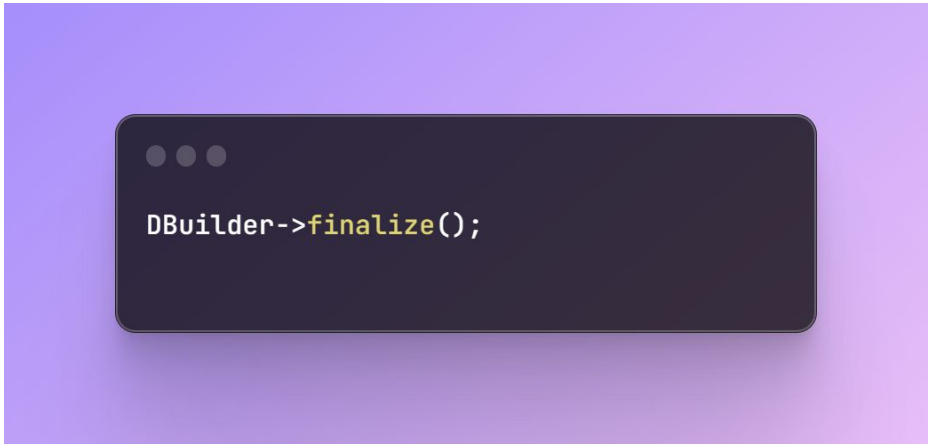
- Debugging optimized code is challenging due to difficulties in keeping track of source locations and variables as it keeps the original source location for each instruction but merged instruction only get to keep a single location.
- Optimization can move variables in ways that make them difficult to track such as being optimized out or shared in memory with other variables.
- Disable all of the optimization passes and the JIT so that the only thing that happens after parsing and generating code is that the LLVM IR goes to standard error.

DWARF Emission Setup

- The top level container for a section of code in DWARF is a compile unit. This contains the type and function data for an individual translation unit.
- Similar to the IRBuilder class we have a DIBuilder class that helps in constructing debug metadata for an LLVM IR file.
- we're going to create a small container to cache some of our frequent data. The first will be our compile unit..

```
static std::unique_ptr<DIBuilder> DBuilder;

struct DebugInfo {
  DICompileUnit *TheCU;
  DIType *DblTy;

  DIType *getDoubleTy();
} KSDbgInfo;

DIType *DebugInfo::getDoubleTy() {
  if (DblTy)
    return DblTy;

  DblTy = DBuilder->createBasicType("double", 64, dwarf::DW_ATE_float);
  return DblTy;
}
```

- The compile unit language constant for C is used to ensure compatibility with debuggers. This is because a debugger wouldn't necessarily understand the calling conventions or default ABI for a language it doesn't recognize and we follow the C ABI in our LLVM code generation.
- Debug information emitted via DIBuilder needs to be finalized near the end of main before dumping out the module. The reasons are part of the underlying API for DIBuilder.



Functions

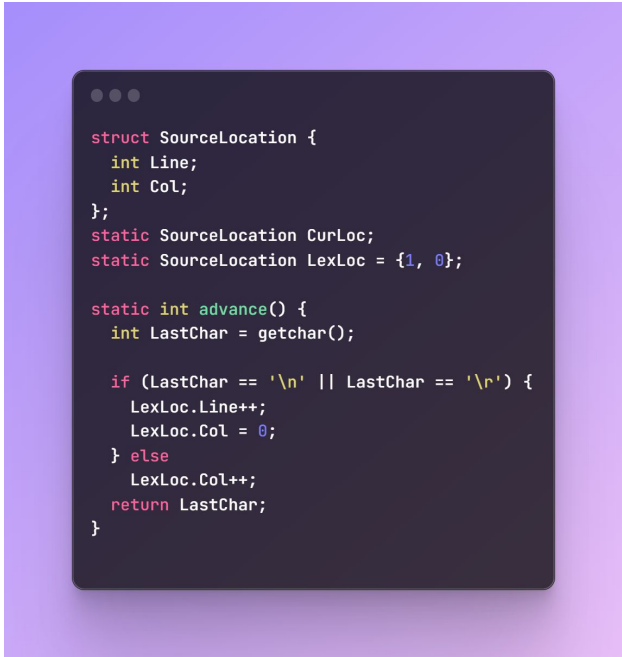
- The FunctionAST::codegen() function is used to add function definitions to the debug info.
- The debug information includes information about the context of the subprogram and the definition of the function itself.
- A DIFile is created to provide the context for the subprogram by asking the Compile Unit for the directory and filename of the current location.

- Currently, the source location information is not available in the AST, so some source locations are set to 0.
- A DISubprogram is constructed that contains a reference to all of our metadata for the function.

```
DIScope *FContext = Unit;
unsigned LineNo = 0;
unsigned ScopeLine = 0;
DISubprogram *SP = DBuilder->createFunction(
    FContext, P.getName(), StringRef(), Unit, LineNo,
    CreateFunctionType(TheFunction->arg_size()),
    ScopeLine,
    DINode::FlagPrototyped,
    DISubprogram::SPFlagDefinition);
TheFunction->setSubprogram(SP);
```

Source Locations

- Accurate source location is crucial for debug information and to map back to source code.
- Kaleidoscope does not have any source location information in the lexer or parser, so functionality has been added to keep track of the line and column of the "source file".
- Current "lexical location" is set to the corresponding line and column for the beginning of each token, overriding previous calls to getchar(), using a new function called advance() that keeps track of the information.
- A source location is added to all AST classes, passed down when creating a new expression, giving locations for each expression and variable.



```
struct SourceLocation {
  int Line;
  int Col;
};
static SourceLocation CurLoc;
static SourceLocation LexLoc = {1, 0};

static int advance() {
  int LastChar = getchar();

  if (LastChar == '\n' || LastChar == '\r') {
    LexLoc.Line++;
    LexLoc.Col = 0;
  } else
    LexLoc.Col++;
  return LastChar;
}
```

Source Locations

- To ensure that every instruction gets proper source location information, a helper function is used to tell the Builder whenever a new source location is encountered.
- A stack of scopes is created in DebugInfo, and the scope (function) is pushed to the top of the stack when generating code for each function.
- The scope is popped off the scope stack at the end of code generation for the function.
- The location is emitted every time code generation starts for a new AST object.

```
class ExprAST {
  SourceLocation Loc;

  public:
    ExprAST(SourceLocation Loc = CurLoc) : Loc(Loc) {}
    virtual ~ExprAST() {}
    virtual Value* codegen() = 0;
    int getLine() const { return Loc.Line; }
    int getCol() const { return Loc.Col; }
    virtual raw_ostream &dump(raw_ostream &out, int ind) {
      return out << ':' << getLine() << ':' << getCol() <<
'\n';
    }
```

```
void DebugInfo::emitLocation(ExprAST *AST) {
  if (!AST)
    return Builder->SetCurrentDebugLocation(DebugLoc());
  DIScope *Scope;
  if (LexicalBlocks.empty())
    Scope = TheCU;
  else
    Scope = LexicalBlocks.back();
  Builder->SetCurrentDebugLocation(
      DILocation::get(Scope->getContext(), AST->getLine(),
AST->getCol(), Scope));
}
```

Variables

- Printing out variables in scope is important when dealing with functions.
- The argument allocas in FunctionAST::codegen are created with the variable's scope (SP), name, source location, type, and argument index.
- An lvm.dbg.declare call is created at the IR level to indicate that a variable is in an alloca, with a starting location for the variable, and setting a source location for the beginning of the scope on the declare.
- A hack is added in FunctionAST::CodeGen to avoid generating line information for the function prologue to allow the debugger to skip those instructions when setting a breakpoint.
- A new location is emitted when generating code for the body of the function to provide enough debug information to set breakpoints, print out argument variables, and call functions.

```
// Record the function arguments in the NamedValues map.
NamedValues.clear();
unsigned ArgIdx = 0;
for (auto &Arg : TheFunction->args()) {
  // Create an alloca for this variable.
  AllocaInst *Alloca =
CreateEntryBlockAlloca(TheFunction, Arg.getName());

  // Create a debug descriptor for the variable.
  DILocalVariable *D = DBuilder->createParameterVariable(
      SP, Arg.getName(), ++ArgIdx, Unit, LineNo,
KSDbgInfo.getDoubleTy(),
      true);

  DBuilder->insertDeclare(Alloca, D, DBuilder-
>createExpression(),
                          DILocation::get(SP-
>getContext(), LineNo, 0, SP),
                          Builder->GetInsertBlock());

  // Store the initial value into the alloca.
  Builder->CreateStore(&Arg, Alloca);

  // Add arguments to variable symbol table.
  NamedValues[std::string(Arg.getName())] = Alloca;
}
```

Properties of LLVM

Target Independence:

- LLVM IR can preserve target independence in the code
- You can run LLVM IR for a program on any target that LLVM supports
- Kaleidoscope compiler generates target-independent code as it never queries for target-specific information.

Safety Guarantees:

- LLVM IR does not guarantee safety
- Safety needs to be implemented as a layer on top of LLVM

Properties of LLVM

Language-Specific Optimizations:

- LLVM loses some information, such as the difference between "int" and "long" on an ILP32 machine
- LLVM continues to improve and extend its IR to capture important information for optimization
- It is easy to add language-specific optimization passes that know things about the code compiled for a language
- You can embed a variety of other language-specific information into the LLVM IR

LLVM: Tips and Tricks

- Implementing portable offsetof/sizeof: To keep code generated by your compiler target-independent, you may need to know the size of an LLVM type or the offset of a field in an LLVM structure.
- This can vary widely across targets, but using the getelementptr instruction allows you to compute this in a portable way.
- Garbage Collected Stack Frames: Some languages want to explicitly manage their stack frames for garbage collection or to implement closures.
- LLVM supports this through the use of Continuation Passing Style and tail calls, but there are often better ways to implement these features.

Conclusion

- The lexer, parser, AST, code generator, interactive run-loop, JIT, and debug information were built in under 1000 lines of code.
- Kaleidoscope supports user-defined binary and unary operators, JIT compilation, and a few control flow constructs with SSA construction.
- Readers are encouraged to take the code and hack on it, adding features such as global variables, typed variables, arrays, structs, vectors, and more. Memory management and exception handling are other features that can be added to the language.

Conclusion

- As the language evolves, it may make sense to add higher-level constructs to a standard runtime rather than inlining them.
- There is no limit to the number of features that can be added to the language, such as object orientation, generics, database access, complex numbers, geometric programming, and more.
- LLVM can be used for many other domains besides building compilers, such as implementing graphics acceleration and translating code to other languages.