2. Kaleidoscope: Implementing a Parser and AST

2.1. Chapter 2 Introduction

Welcome to Chapter 2 of the "Implementing a language with LLVM" tutorial. This chapter shows you how to use the lexer, built in Chapter 1, to build a full parser for our Kaleidoscope language. Once we have a parser, we'll define and build an Abstract Syntax Tree (AST).

The parser we will build uses a combination of Recursive Descent Parsing and Operator-Precedence Parsing to parse the Kaleidoscope language (the latter for binary expressions and the former for everything else). Before we get to parsing though, let's talk about the output of the parser: the Abstract Syntax Tree.

2.2. The Abstract Syntax Tree (AST)

The AST for a program captures its behavior in such a way that it is easy for later stages of the compiler (e.g. code generation) to interpret. 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. We'll start with expressions first:

```
/// ExprAST - Base class for all expression nodes.
class ExprAST {
public:
  virtual ~ExprAST() {}
};

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

public:
```

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

The code above shows the definition of the base ExprAST class and one subclass which we use for numeric literals. The important thing to note about this code is that the NumberExprAST class captures the numeric value of the literal as an instance variable. This allows later phases of the compiler to know what the stored numeric value is.

Right now we only create the AST, so there are no useful accessor methods on them. It would be very easy to add a virtual method to pretty print the code, for example. Here are the other expression AST node definitions that we'll use in the basic form of the Kaleidoscope language:

```
/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;

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

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  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 {
  std::string Callee;
  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)) {}
};
```

This is all (intentionally) rather straight-forward: variables capture the variable name, binary operators capture their opcode (e.g. '+'), and calls capture a function name as well as a list of any argument expressions. One thing that is nice about our AST is that it captures the language features without talking about the syntax of the language. Note that there is no discussion about precedence of binary operators, lexical structure, etc.

For our basic language, these are all of the expression nodes we'll define. Because it doesn't have conditional control flow, it isn't Turing-complete; we'll fix that in a later installment. The

two things we need next are a way to talk about the interface to a function, and a way to talk about functions themselves:

```
/// 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)) {}

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

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  std::unique_ptr<PrototypeAST> Proto;
  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)) {}
};
```

In Kaleidoscope, functions are typed with just a count of their arguments. Since all values are double precision floating point, the type of each argument doesn't need to be stored anywhere. In a more aggressive and realistic language, the "ExprAST" class would probably have a type field.

With this scaffolding, we can now talk about parsing expressions and function bodies in Kaleidoscope.

2.3. Parser Basics

Now that we have an AST to build, we need to define the parser code to build it. The idea here is that we want to parse something like "x+y" (which is returned as three tokens by the lexer) into an AST that could be generated with calls like this:

```
auto LHS = std::make_unique<VariableExprAST>("x");
auto RHS = std::make_unique<VariableExprAST>("y");
auto Result = std::make_unique<BinaryExprAST>('+', std::move(LHS),
                                              std::move(RHS));
```

In order to do this, we'll start by defining some basic helper routines:

```
/// 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();
}
```

This implements a simple token buffer around the lexer. This 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.

```
/// LogError* - These are little helper functions for error handling.
std::unique_ptr<ExprAST> LogError(const char *Str) {
  fprintf(stderr, "LogError: %s\n", Str);
  return nullptr;
}
std::unique_ptr<PrototypeAST> LogErrorP(const char *Str) {
  LogError(Str);
  return nullptr;
}
```

The `LogError` routines are simple helper routines that our parser will use to handle errors. The error recovery in our parser will not be the best and is not particular user-friendly, but it will be enough for our tutorial. These routines make it easier to handle errors in routines that have various return types: they always return null.

With these basic helper functions, we can implement the first piece of our grammar: numeric literals.

2.4. Basic Expression Parsing

We start with numeric literals, because they are the simplest to process. For each production in our grammar, we'll define a function which parses that production. For numeric literals, we have:

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

This routine is very simple: it expects to be called when the current token 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.

There are some interesting aspects to this. The most important one is that this routine eats all of the tokens that correspond to the production and returns the lexer buffer with the next token (which is not part of the grammar production) ready to go. This is a fairly standard way to go for recursive descent parsers. For a better example, the parenthesis operator is defined like this:

```
/// 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;
}
```

This function illustrates a number of interesting things about the parser:

1) It shows how we use the LogError routines. When called, this function expects that the current token is a '(' token, but after parsing the subexpression, it is possible that there is no ')' waiting. For example, if the user types in "(4 x" instead of "(4)", the parser should emit an error. Because errors can occur, the parser needs a way to indicate that they happened: in our parser, we return null on an error.

2) Another interesting aspect of this function is that it uses recursion by calling `ParseExpression` (we will soon see that `ParseExpression` can call `ParseParenExpr`). This is powerful because it allows us to handle recursive grammars, and keeps each production very simple. Note that parentheses do not cause construction of AST nodes themselves. While we could do it this way, the most important role of parentheses are to guide the parser and provide grouping. Once the parser constructs the AST, parentheses are not needed.

The next simple production is for handling variable references and function calls:

```
/// 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 (1) {
      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();
    }
  }

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

  return std::make_unique<CallExprAST>(IdName, std::move(Args));
}
```

This routine follows the same style as the other routines. (It expects to be called if the current token is a `tok_identifier` token). It also has recursion and error handling. One interesting aspect of this is that it uses *look-ahead* to determine if the current identifier is a stand alone variable reference or if it is a function call expression. It handles this by checking to see if the token after the identifier is a '(' token, constructing either a `VariableExprAST` or `CallExprAST` node as appropriate.

Now that we have all of our simple expression-parsing logic in place, we can define a helper function to wrap it together into one entry point. We call this class of expressions "primary" expressions, for reasons that will become more clear later in the tutorial. In order to parse an arbitrary primary expression, we need to determine what sort of expression it is:

```
/// 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();
  }
}
```

Now that you see the definition of this function, it is more obvious why we can assume the state of CurTok in the various functions. This uses look-ahead to determine which sort of expression is being inspected, and then parses it with a function call.

Now that basic expressions are handled, we need to handle binary expressions. They are a bit more complex.

2.5. Binary Expression Parsing

Binary expressions are significantly harder to parse because they are often ambiguous. For example, when given the string "x+y*z", the parser can choose to parse it as either "(x+y)*z" or "x+(y*z)". With common definitions from mathematics, we expect the later parse, because "*" (multiplication) has higher *precedence* than "+" (addition).

There are many ways to handle this, but an elegant and efficient way is to use Operator-Precedence Parsing. This parsing technique uses the precedence of binary operators to guide recursion. To start with, we need a table of precedences:

```
/// 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;
}

int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.
  ...
}
```

For the basic form of Kaleidoscope, we will only support 4 binary operators (this can obviously be extended by you, our brave and intrepid reader). The `GetTokPrecedence` function returns the precedence for the current token, or -1 if the token is not a binary operator. Having a map makes it easy to add new operators and makes it clear that the algorithm doesn't depend on the specific operators involved, but it would be easy enough to eliminate the map and do the comparisons in the `GetTokPrecedence` function. (Or just use a fixed-size array).

With the helper above defined, we can now start parsing binary expressions. The basic idea of operator precedence parsing is to break down an expression with potentially ambiguous binary operators into pieces. Consider, for example, the expression "a+b+(c+d)*e*f+g". Operator precedence parsing considers this as a stream of primary expressions separated by binary operators. As such, it will first parse the leading primary expression "a", then it will see the pairs [+, b] [+, (c+d)] [*, e] [*, f] and [+, g]. Note that because parentheses are primary expressions, the binary expression parser doesn't need to worry about nested subexpressions like (c+d) at all.

To start, an expression is a primary expression potentially followed by a sequence of [binop,primaryexpr] pairs:

```
/// expression
///   ::= primary binoprhs
///
static std::unique_ptr<ExprAST> ParseExpression() {
  auto LHS = ParsePrimary();
  if (!LHS)
    return nullptr;

  return ParseBinOpRHS(0, std::move(LHS));
}
```

`ParseBinOpRHS` is the function that parses the sequence of pairs for us. It takes a precedence and a pointer to an expression for the part that has been parsed so far. Note that "x" is a perfectly valid expression: As such, "binoprhs" is allowed to be empty, in which case it returns the expression that is passed into it. In our example above, the code passes the expression for "a" into `ParseBinOpRHS` and the current token is "+".

The precedence value passed into `ParseBinOpRHS` indicates the *minimal operator precedence* that the function is allowed to eat. For example, if the current pair stream is [+, x] and `ParseBinOpRHS` is passed in a precedence of 40, it will not consume any tokens (because the precedence of '+' is only 20). With this in mind, `ParseBinOpRHS` starts with:

```
/// binoprhs
///   ::= ('+' primary)*
static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,
                                              std::unique_ptr<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;
```

This code gets the precedence of the current token and checks to see if if is too low. Because we defined invalid tokens to have a precedence of -1, this check implicitly knows that the

pair-stream ends when the token stream runs out of binary operators. If this check succeeds, we know that the token is a binary operator and that it will be included in this expression:

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

// Parse the primary expression after the binary operator.
auto RHS = ParsePrimary();
if (!RHS)
  return nullptr;
```

As such, this code eats (and remembers) the binary operator and then parses the primary expression that follows. This builds up the whole pair, the first of which is [+, b] for the running example.

Now that we parsed the left-hand side of an expression and one pair of the RHS sequence, we have to decide which way the expression associates. In particular, we could have "(a+b) binop unparsed" or "a + (b binop unparsed)". To determine this, we look ahead at "binop" to determine its precedence and compare it to BinOp's precedence (which is '+' in this case):

```
// 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) {
```

If the precedence of the binop to the right of "RHS" is lower or equal to the precedence of our current operator, then we know that the parentheses associate as "(a+b) binop ...". In our example, the current operator is "+" and the next operator is "+", we know that they have the same precedence. In this case we'll create the AST node for "a+b", and then continue parsing:

```
      ... if body omitted ...
    }

    // Merge LHS/RHS.
    LHS = std::make_unique<BinaryExprAST>(BinOp, std::move(LHS),
                                          std::move(RHS));
  }  // loop around to the top of the while loop.
}
```

In our example above, this will turn "a+b+" into "(a+b)" and execute the next iteration of the loop, with "+" as the current token. The code above will eat, remember, and parse "(c+d)" as the primary expression, which makes the current pair equal to [+, (c+d)]. It will then evaluate the 'if' conditional above with "*" as the binop to the right of the primary. In this case, the precedence of "*" is higher than the precedence of "+" so the if condition will be entered.

The critical question left here is "how can the if condition parse the right hand side in full"? In particular, to build the AST correctly for our example, it needs to get all of "(c+d)*e*f" as the

RHS expression variable. The code to do this is surprisingly simple (code from the above two blocks duplicated for context):

```
    // 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, std::move(RHS));
      if (!RHS)
        return nullptr;
    }
    // Merge LHS/RHS.
    LHS = std::make_unique<BinaryExprAST>(BinOp, std::move(LHS),
                                          std::move(RHS));
  }  // loop around to the top of the while loop.
}
```

At this point, we know that the binary operator to the RHS of our primary has higher precedence than the binop we are currently parsing. As such, we know that any sequence of pairs whose operators are all higher precedence than "+" should be parsed together and returned as "RHS". To do this, we recursively invoke the `ParseBinOpRHS` function specifying "TokPrec+1" as the minimum precedence required for it to continue. In our example above, this will cause it to return the AST node for "(c+d)*e*f" as RHS, which is then set as the RHS of the '+' expression.

Finally, on the next iteration of the while loop, the "+g" piece is parsed and added to the AST. With this little bit of code (14 non-trivial lines), we correctly handle fully general binary expression parsing in a very elegant way. This was a whirlwind tour of this code, and it is somewhat subtle. I recommend running through it with a few tough examples to see how it works.

This wraps up handling of expressions. At this point, we can point the parser at an arbitrary token stream and build an expression from it, stopping at the first token that is not part of the expression. Next up we need to handle function definitions, etc.

2.6. Parsing the Rest

The next thing missing is handling of function prototypes. In Kaleidoscope, these are used both for 'extern' function declarations as well as function body definitions. The code to do this is straight-forward and not very interesting (once you've survived expressions):

```
/// 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));
}
```

Given this, a function definition is very simple, just a prototype plus an expression to implement the body:

```
/// 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;
}
```

In addition, we support ‘extern’ to declare functions like ‘sin’ and ‘cos’ as well as to support forward declaration of user functions. These ‘extern’s are just prototypes with no body:

```
/// external ::= 'extern' prototype
static std::unique_ptr<PrototypeAST> ParseExtern() {
  getNextToken();  // eat extern.
  return ParsePrototype();
}
```

Finally, we’ll also let the user type in arbitrary top-level expressions and evaluate them on the fly. We will handle this by defining anonymous nullary (zero argument) functions for them:

```
/// toplevelexpr ::= expression
static std::unique_ptr<FunctionAST> ParseTopLevelExpr() {
  if (auto E = ParseExpression()) {
    // Make an anonymous proto.
    auto Proto = std::make_unique<PrototypeAST>("", std::vector<std::string>());
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  }
```

```
  return nullptr;
}
```

Now that we have all the pieces, let's build a little driver that will let us actually *execute* this code we've built!

2.7. The Driver

The driver for this simply invokes all of the parsing pieces with a top-level dispatch loop. There isn't much interesting here, so I'll just include the top-level loop. See below for full code in the "Top-Level Parsing" section.

```
/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (1) {
    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;
    }
  }
}
```

The most interesting part of this is that we ignore top-level semicolons. Why is this, you ask? The basic reason is that if you type "4 + 5" at the command line, the parser doesn't know whether that is the end of what you will type or not. For example, on the next line you could type "def foo..." in which case 4+5 is the end of a top-level expression. Alternatively you could type "* 6", which would continue the expression. Having top-level semicolons allows you to type "4+5;", and the parser will know you are done.

2.8. Conclusions

With just under 400 lines of commented code (240 lines of non-comment, non-blank code), we fully defined our minimal language, including a lexer, parser, and AST builder. With this

done, the executable will validate Kaleidoscope code and tell us if it is grammatically invalid. For example, here is a sample interaction:

```
$ ./a.out
ready> def foo(x y) x+foo(y, 4.0);
Parsed a function definition.
ready> def foo(x y) x+y y;
Parsed a function definition.
Parsed a top-level expr
ready> def foo(x y) x+y );
Parsed a function definition.
Error: unknown token when expecting an expression
ready> extern sin(a);
ready> Parsed an extern
ready> ^D
$
```

There is a lot of room for extension here. You can define new AST nodes, extend the language in many ways, etc. In the next installment, we will describe how to generate LLVM Intermediate Representation (IR) from the AST.

2.9. Full Code Listing

Here is the complete code listing for our running example. Because this uses the LLVM libraries, we need to link them in. To do this, we use the llvm-config tool to inform our makefile/command line about which options to use:

```
# Compile
clang++ -g -O3 toy.cpp `llvm-config --cxxflags`
# Run
./a.out
```

Here is the code:

```
#include <cctype>
#include <cstdio>
#include <cstdlib>
#include <map>
#include <memory>
#include <string>
#include <utility>
#include <vector>

//===----------------------------------------------------------------------===//
// 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
};

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;
    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(), nullptr);
    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() = default;
};

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

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

/// VariableExprAST - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;

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

/// BinaryExprAST - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;
  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 {
  std::string Callee;
  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)) {}
};

/// 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)) {}

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

/// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  std::unique_ptr<PrototypeAST> Proto;
  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)) {}
};

} // 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;
}

/// LogError* - These are little helper functions for error handling.
std::unique_ptr<ExprAST> LogError(const char *Str) {
  fprintf(stderr, "Error: %s\n", Str);
  return nullptr;
}
std::unique_ptr<PrototypeAST> LogErrorP(const char *Str) {
  LogError(Str);
  return nullptr;
}

static std::unique_ptr<ExprAST> ParseExpression();

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

/// 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;
}

/// 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();
    }
  }

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

  return std::make_unique<CallExprAST>(IdName, std::move(Args));
}

/// 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();
  }
}

/// binoprhs
///   ::= ('+' primary)*
static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,
                                              std::unique_ptr<ExprAST> LHS) {
  // If this is a binop, find its precedence.
  while (true) {
    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 primary expression after the binary operator.
    auto RHS = ParsePrimary();
    if (!RHS)
      return nullptr;
```

```
    // 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, std::move(RHS));
      if (!RHS)
        return nullptr;
    }

    // Merge LHS/RHS.
    LHS =
        std::make_unique<BinaryExprAST>(BinOp, std::move(LHS), std::move(RHS));
  }
}

/// expression
///   ::= primary binoprhs
///
static std::unique_ptr<ExprAST> ParseExpression() {
  auto LHS = ParsePrimary();
  if (!LHS)
    return nullptr;

  return ParseBinOpRHS(0, std::move(LHS));
}

/// 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");

  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));
}

/// 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;
}

/// toplevelexpr ::= expression
static std::unique_ptr<FunctionAST> ParseTopLevelExpr() {
  if (auto E = ParseExpression()) {
    // Make an anonymous proto.
    auto Proto = std::make_unique<PrototypeAST>("__anon_expr",
                                                std::vector<std::string>());
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  }
  return nullptr;
}

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

//===----------------------------------------------------------------------===//
// Top-Level parsing
//===----------------------------------------------------------------------===//

static void HandleDefinition() {
  if (ParseDefinition()) {
    fprintf(stderr, "Parsed a function definition.\n");
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleExtern() {
  if (ParseExtern()) {
    fprintf(stderr, "Parsed an extern\n");
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (ParseTopLevelExpr()) {
    fprintf(stderr, "Parsed a top-level expr\n");
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}
```

```
/// 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;
    }
  }
}

//===----------------------------------------------------------------------===//
// Main driver code.
//===----------------------------------------------------------------------===//

int main() {
  // 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();

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

  return 0;
}
```