使用LLVM实现一门语言（二）Parser

编译的第二个步骤称为Parse，其功能是将Lexer输出的tokens转为AST (Abstract Syntax Tree).

我们首先定义表达式的AST Node

```
// 所有 `表达式` 节点的基类
class ExprAST {
 public:
  virtual ~ExprAST() {}
};

// 字面值表达式
class NumberExprAST : public ExprAST {
 public:
  NumberExprAST(double val) : val_(val) {}

 private:
  double val_;
};

// 变量表达式
class VariableExprAST : public ExprAST {
 public:
  VariableExprAST(const std::string& name) : name_(name) {}

 private:
  std::string name_;
};

// 二元操作表达式
class BinaryExprAST : public ExprAST {
 public:
  BinaryExprAST(char op, std::unique_ptr<ExprAST> lhs,
                std::unique_ptr<ExprAST> rhs)
      : op_(op), lhs_(std::move(lhs)), rhs_(std::move(rhs)) {}

 private:
  char op_;
  std::unique_ptr<ExprAST> lhs_;
  std::unique_ptr<ExprAST> rhs_;
};

// 函数调用表达式
class CallExprAST : public ExprAST {
 public:
  CallExprAST(const std::string& callee,
```

```
              std::vector<std::unique_ptr<ExprAST>> args)
      : callee_(callee), args_(std::move(args)) {}

 private:
  std::string callee_;
  std::vector<std::unique_ptr<ExprAST>> args_;
};
```

为了便于理解，关于条件表达式的内容放在后面，这里暂不考虑。

接着我们定义函数声明和函数的AST Node

```
// 函数接口
class PrototypeAST {
 public:
  PrototypeAST(const std::string& name, std::vector<std::string> args)
      : name_(name), args_(std::move(args)) {}

  const std::string& name() const { return name_; }

 private:
  std::string name_;
  std::vector<std::string> args_;
};

// 函数
class FunctionAST {
 public:
  FunctionAST(std::unique_ptr<PrototypeAST> proto,
              std::unique_ptr<ExprAST> body)
      : proto_(std::move(proto)), body_(std::move(body)) {}

 private:
  std::unique_ptr<PrototypeAST> proto_;
  std::unique_ptr<ExprAST> body_;
};
```

接下来我们要进行Parse，在正式Parse前，定义如下函数方便后续处理

```
int g_current_token;  // 当前待处理的Token
int GetNextToken() {
  return g_current_token = GetToken();
}
```

首先我们处理最简单的字面值

```
// numberexpr ::= number
std::unique_ptr<ExprAST> ParseNumberExpr() {
  auto result = std::make_unique<NumberExprAST>(g_number_val);
  GetNextToken();
  return std::move(result);
}
```

这段程序非常简单，当前Token为TOKEN_NUMBER时被调用，使用g_number_val

创建一个NumberExprAST，因为当前Token处理完毕，让Lexer前进一个Token，最后返回。

接着我们处理圆括号操作符、变量、函数调用

```
// parenexpr ::= ( expression )
std::unique_ptr<ExprAST> ParseParenExpr() {
  GetNextToken();  // eat (
  auto expr = ParseExpression();
  GetNextToken();  // eat )
  return expr;
}

/// identifierexpr
///   ::= identifier
///   ::= identifier ( expression, expression, ..., expression )
std::unique_ptr<ExprAST> ParseIdentifierExpr() {
  std::string id = g_identifier_str;
  GetNextToken();
  if (g_current_token ≠ '(') {
    return std::make_unique<VariableExprAST>(id);
  } else {
    GetNextToken();  // eat (
    std::vector<std::unique_ptr<ExprAST>> args;
    while (g_current_token ≠ ')') {
      args.push_back(ParseExpression());
      if (g_current_token == ')') {
        break;
      } else {
        GetNextToken();  // eat ,
      }
    }
    GetNextToken();  // eat )
    return std::make_unique<CallExprAST>(id, std::move(args));
  }
}
```

上面代码中的ParseExpression与ParseParenExpr等存在循环依赖，这里按照其名字理解意思即可，具体实现在后面。

我们将NumberExpr、ParenExpr、IdentifierExpr视为PrimaryExpr，封装ParsePrimary方便后续调用

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
std::unique_ptr<ExprAST> ParsePrimary() {
  switch (g_current_token) {
    case TOKEN_IDENTIFIER: return ParseIdentifierExpr();
```

```
    case TOKEN_NUMBER: return ParseNumberExpr();
    case '(': return ParseParenExpr();
    default: return nullptr;
  }
}
```

接下来我们考虑如何处理二元操作符，为了方便，Kaleidoscope只支持4种二元操作符，优先级为

```
'<' < '+' = '-' < '*'
```

即'<'的优先级最低，而'*'的优先级最高，在代码中实现为

```
// 定义优先级
const std::map<char, int> g_binop_precedence = {
    {'<', 10}, {'+', 20}, {'-', 20}, {'*', 40}};

// 获得当前Token的优先级
int GetTokenPrecedence() {
  auto it = g_binop_precedence.find(g_current_token);
  if (it ≠ g_binop_precedence.end()) {
    return it→second;
  } else {
    return -1;
  }
}
```

对于带优先级的二元操作符的解析，我们会将其分成多个片段。比如一个表达式

```
a + b + (c + d) * e * f + g
```

首先解析a，然后处理多个二元组[+, b], [+, (c+d)], [*, e], [*, f], [+, g]

即，复杂表达式可以抽象为一个PrimaryExpr跟着多个[binop, PrimaryExpr]二元组，注意由于圆括号属于PrimaryExpr，所以这里不需要考虑怎么特殊处理(c+d)，因为会被ParsePrimary自动处理。

```
// parse
//   lhs [binop primary] [binop primary] ...
// 如遇到优先级小于min_precedence的操作符，则停止
std::unique_ptr<ExprAST> ParseBinOpRhs(int min_precedence,
                                       std::unique_ptr<ExprAST> lhs) {
  while (true) {
    int current_precedence = GetTokenPrecedence();
    if (current_precedence < min_precedence) {
      // 如果当前token不是二元操作符，current_precedence为-1，结束任务
      // 如果遇到优先级更低的操作符，也结束任务
      return lhs;
    }
    int binop = g_current_token;
    GetNextToken();  // eat binop
```

```
    auto rhs = ParsePrimary();
    // 现在我们有两种可能的解析方式
    //    * (lhs binop rhs) binop unparsed
    //    * lhs binop (rhs binop unparsed)
    int next_precedence = GetTokenPrecedence();
    if (current_precedence < next_precedence) {
      // 将高于current_precedence的右边的操作符处理掉返回
      rhs = ParseBinOpRhs(current_precedence + 1, std::move(rhs));
    }
    lhs =
        std::make_unique<BinaryExprAST>(binop, std::move(lhs), std::move(rhs));
    // 继续循环
  }
}

// expression
//   ::= primary [binop primary] [binop primary] ...
std::unique_ptr<ExprAST> ParseExpression() {
  auto lhs = ParsePrimary();
  return ParseBinOpRhs(0, std::move(lhs));
}
```

最复杂的部分完成后，按部就班把function写完

```
// prototype
//   ::= id ( id id ... id)
std::unique_ptr<PrototypeAST> ParsePrototype() {
  std::string function_name = g_identifier_str;
  GetNextToken();
  std::vector<std::string> arg_names;
  while (GetNextToken() == TOKEN_IDENTIFIER) {
    arg_names.push_back(g_identifier_str);
  }
  GetNextToken();  // eat )
  return std::make_unique<PrototypeAST>(function_name, std::move(arg_names));
}

// definition ::= def prototype expression
std::unique_ptr<FunctionAST> ParseDefinition() {
  GetNextToken();  // eat def
  auto proto = ParsePrototype();
  auto expr = ParseExpression();
  return std::make_unique<FunctionAST>(std::move(proto), std::move(expr));
}

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

最后，我们为顶层的代码实现匿名function

```
// toplevelexpr ::= expression
std::unique_ptr<FunctionAST> ParseTopLevelExpr() {
  auto expr = ParseExpression();
  auto proto = std::make_unique<PrototypeAST>("", std::vector<std::string>());
  return std::make_unique<FunctionAST>(std::move(proto), std::move(expr));
}
```

顶层代码的意思是放在全局而不放在function内定义的一些执行语句比如变量赋值，函数调用等。

编写一个main函数

```
int main() {
  GetNextToken();
  while (true) {
    switch (g_current_token) {
      case TOKEN_EOF: return 0;
      case TOKEN_DEF: {
        ParseDefinition();
        std::cout << "parsed a function definition" << std::endl;
        break;
      }
      case TOKEN_EXTERN: {
        ParseExtern();
        std::cout << "parsed a extern" << std::endl;
        break;
      }
      default: {
        ParseTopLevelExpr();
        std::cout << "parsed a top level expr" << std::endl;
        break;
      }
    }
  }
  return 0;
}
```

编译

```
clang++ main.cpp `llvm-config --cxxflags --ldflags --libs`
```

输入如下代码进行测试

```
def foo(x y)
    x + foo(y, 4)

def foo(x y)
    x + y

y

extern sin(a)
```

得到输出

```
parsed a function definition
parsed a function definition
parsed a top level expr
parsed a extern
```

成功将Lexer输出的tokens转为AST