Kaleidoscope

代码解释(1/3) 第1-3章

万花筒语言

LLVM 新手入门教程

achieveartificialintelligence

Kaleidoscope简介

- Kaleidoscope是一个简单、完整、使用LLVM框架构建的编译器。
- 为新手准备，需要一定的C++、编译原理基础。
- Kaleidoscope可以定义函数、流程控制、自定义操作符、JIT编译、调试信息。
- 数值类型只有double，未提供完备的错误处理。
- 代码符合LLVM Coding Standard。

环境准备

- Ubuntu 20.04 (WSL版)
- 下载LLVM 12.0.0最新源码
  - git clone https://github.com/llvm/llvm-project.git
- Cmake 3.14+生成Makefile
  - mkdir build && cd build
  - cmake –G “Unix Makefiles” ../llvm
- Make编译
  - make –j32 Kaleidoscope
- 运行Kaleidoscope
  - bin/Kaleidoscope-Ch9

Kaleidoscope整体结构

0 预处理部分

```
//#include "llvm/ADT/STLExtras.h"
// LLVM实现的标准库拓展，本次用不到,本次可直接clang++ toy.cpp进行编译
//#include <algorithm>
#include <cctype>  // isspace isalpha isalnum
#include <cstdio>  // EOF getchar stderr fprintf
#include <cstdlib> // strtod
#include <map>     // map
#include <memory>  // move unique_ptr
#include <string>  // string
#include <vector>  // vector
```

1 词法分析器

```
enum Token             // 词法单元
{ tok_eof = -1,        // EOF 文件结束信号值-1
  tok_def = -2,        // 函数定义关键字 def
  tok_extern = -3,     //                extern
  tok_identifier = -4, // 标识符
  tok_number = -5      // 数字
};

static std::string IdentifierStr; // 用于存储标识符的名称
static double NumVal;             // 用于存放数字的值

static int gettok() // 从cin中读取下一个词法单元的枚举值
{
  static int LastChar = ' '; // 第一次定义初始化

  while (isspace(LastChar)) // 跳过所有:' ','\r','\n','\t','\v','\f'
    LastChar = getchar();   // 读取下一个字符,直到不是空格

  if (isalpha(LastChar)) // 标识符正则[a-zA-Z][a-zA-Z0-9]*
  {
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getchar()))) //正则[a-zA-Z0-9]*
      // 外层括号为函数调用,内层关闭编译器的提示
      IdentifierStr += LastChar;

    if (IdentifierStr == "def") // 特殊情况def
      return tok_def;
    if (IdentifierStr == "extern") //特殊情况extern
      return tok_extern;
    return tok_identifier; // 返回Token的枚举值
  }

  if (isdigit(LastChar) || LastChar == '.') // 数字正则[0-9.]+
```

1

```
  if (isdigit(LastChar) || LastChar == '.') // 数字正则[0-9.]+
  {
    std::string NumStr; // 存储数字值的string形式
    do {
      NumStr += LastChar;
      LastChar = getchar();
    } while (isdigit(LastChar) || LastChar == '.'); //[0-9.],无法处理多个小数点

    NumVal = strtod(NumStr.c_str(), nullptr);
    // string转为double, c_str()为const char*,
    // 第二个指针可以指定为下一次的起始值
    return tok_number;
  }

  if (LastChar == '#') //注释
  {
    do
      LastChar = getchar();
    while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

    if (LastChar != EOF) // 如果没有到结尾,就读取下一个Token
      return gettok();
  }

  if (LastChar == EOF) // 到EOF,结束函数
    return tok_eof;

  int ThisChar = LastChar; // 其他情况,直接返回ASCII值,如运算符+-*/
  LastChar = getchar();    // 为下一次运行提供初始值
  return ThisChar;         // 返回ASCII值
}
```

2 抽象语法树

```
namespace { // 匿名空间,包括了所有的AST

class ExprAST { // AST基类
public:
  virtual ~ExprAST() = default; // 析构的虚函数，采用编译器的默认初始化
};

class NumberExprAST : public ExprAST { // 数字的AST
  double Val;                          //存储数字的字面量

public:
  NumberExprAST(double Val) : Val(Val) {} // 构造函数
};

class VariableExprAST : public ExprAST { // 变量的AST
  std::string Name;                        // 变量名

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

class BinaryExprAST : public ExprAST { // 二元表达式AST
  char Op;                             // 二元运算符如+-*/
  std::unique_ptr<ExprAST> LHS, RHS; // 独占指针: 左操作数,右操作数
  // Left Hand Side, Right Hand Side

public:
  BinaryExprAST(char Op, std::unique_ptr<ExprAST> LHS,
                std::unique_ptr<ExprAST> RHS)
      : Op(Op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}
  // 移动操作数左值指针的右值
};
```

```
  // 移动操作数左值指针的右值
};

class CallExprAST : public ExprAST {          // 函数调用AST
  std::string Callee;                         //调用者函数名称
  std::vector<std::unique_ptr<ExprAST>> Args; // 参数列表

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

class PrototypeAST { // 函数声明AST
  std::string Name;  // 函数名
  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; }
};

class FunctionAST {                     // 函数定义AST
  std::unique_ptr<PrototypeAST> Proto; // 函数声明的AST
  std::unique_ptr<ExprAST> Body;       // 函数体(具体的内容)

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

} // namespace
```

3 语法解析

```
static int CurTok; // 记录当前最近一个还未用到的Token枚举值,为之后的读取进行缓存
static int getNextToken() { return CurTok = gettok(); } // 给CurTok赋值

static std::map<char, int>
    BinopPrecedence; // 红黑树实现 操作符到优先级数字的映射

static int GetTokPrecedence() { // 获取操作符的优先级数字
  if (!isascii(CurTok))         // 处理非ASCII字符
    return -1;

  int TokPrec = BinopPrecedence[CurTok]; // 获取
  if (TokPrec <= 0)                      // 操作符未声明(=0)或非法(<0)
    return -1;
  return TokPrec;
}

std::unique_ptr<ExprAST> LogError(const char *Str) { // AST错误
  fprintf(stderr, "Error: %s\n", Str);               // 错误输出
  return nullptr;
}
std::unique_ptr<PrototypeAST> LogErrorP(const char *Str) { // 函数声明错误
  LogError(Str);
  return nullptr;
}

static std::unique_ptr<ExprAST> ParseExpression(); // 表达式声明

// 基本的数字产生式
/// numberexpr ::= number
static std::unique_ptr<ExprAST> ParseNumberExpr() {
  auto Result = std::make_unique<NumberExprAST>(NumVal);
  getNextToken();           // 获取下一个Token
  return std::move(Result); // 这个Token的构造结果移动出去
}
```

3

```
// 基本的数字产生式
/// numberexpr ::= number
static std::unique_ptr<ExprAST> ParseNumberExpr() {
  auto Result = std::make_unique<NumberExprAST>(NumVal);
  getNextToken();           // 获取下一个Token
  return std::move(Result); // 这个Token的构造结果移动出去
}

// 基本的括号运算符产生式
/// parenexpr ::= '(' expression ')'
static std::unique_ptr<ExprAST> ParseParenExpr() {
  getNextToken(); // 获取下一个Token
  auto V = ParseExpression();
  if (!V)
    return nullptr;

  if (CurTok != ')') // 匹配')'
    return LogError("expected ')'");
  getNextToken(); // 获取下一个Token
  return V;
}

// 基本的标识符产生式
/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static std::unique_ptr<ExprAST> ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();

  if (CurTok != '(') // 判断是函数还是变量
    return std::make_unique<VariableExprAST>(IdName);

  getNextToken();
  std::vector<std::unique_ptr<ExprAST>> Args;
  if (CurTok != ')') {
```

3

```
// 基本的标识符产生式
/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static std::unique_ptr<ExprAST> ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;

  getNextToken();

  if (CurTok != '(') // 判断是函数还是变量
    return std::make_unique<VariableExprAST>(IdName);

  getNextToken();
  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();

  return std::make_unique<CallExprAST>(IdName, std::move(Args)); // 转到函数调用
}

// 基本的表达式结点产生式
/// primary
```

3

```
// 基本的表达式结点产生式
/// 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();
  }
}

// 基本的二元产生式
// 参数:左侧二元操作符优先级(无为0),左操作数
/// binoprhs
///   ::= ('+' primary)*
static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,
                                              std::unique_ptr<ExprAST> LHS) {
  // 如果是二元表达式,计算其优先级
  while (true) {
    int TokPrec = GetTokPrecedence();

    // 如果比传入的ExprPrec还小,就结束函数
    if (TokPrec < ExprPrec)
      return LHS;

    // 这时,获取到了正确的二元操作符
    int BinOp = CurTok;
    getNextToken(); // 获取下一个Token

    // 产生二元表达式在解析了二元操作符后
```

3

```
// 基本的二元产生式
// 参数:左侧二元操作符优先级(无为0),左操作数
/// binoprhs
///   ::= ('+' primary)*
static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,
                                              std::unique_ptr<ExprAST> LHS) {
  // 如果是二元表达式,计算其优先级
  while (true) {
    int TokPrec = GetTokPrecedence();

    // 如果比传入的ExprPrec还小,就结束函数
    if (TokPrec < ExprPrec)
      return LHS;

    // 这时,获取到了正确的二元操作符
    int BinOp = CurTok;
    getNextToken(); // 获取下一个Token

    // 产生二元表达式在解析了二元操作符后
    auto RHS = ParsePrimary();
    if (!RHS) //如果没有右边,出错
      return nullptr;

    // 如果例如1+2*3,左边+ TokPrec 优先级小于右边* NextPrec
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) { // 右侧不存在二元操作符的话,NextPrec=-1
      RHS = ParseBinOpRHS(TokPrec + 1, std::move(RHS));
      if (!RHS)
        return nullptr;
    }

    // 生成二元树
    LHS =
        std::make_unique<BinaryExprAST>(BinOp, std::move(LHS), std::move(RHS));
  }
}
```

3

```
// LHS构造表达式产生式
/// expression
///   ::= primary binoprhs
///
static std::unique_ptr<ExprAST> ParseExpression() {
  auto LHS = ParsePrimary();
  if (!LHS)
    return nullptr;

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

// 表达式声明产生式
/// 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");

  getNextToken();

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

// 函数定义产生式
```

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

// 函数定义产生式
/// 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;
}

// 顶层表达式产生式,将表达式构造为函数
/// toplevelexpr ::= expression
static std::unique_ptr<FunctionAST> ParseTopLevelExpr() {
  if (auto E = ParseExpression()) {
    // 将表达式构造为匿名函数原型
    auto Proto = std::make_unique<PrototypeAST>("__anon_expr",
                                                std::vector<std::string>());
    // 将匿名函数原型构造为函数
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  }
  return nullptr;
}

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

6 顶层语法解析

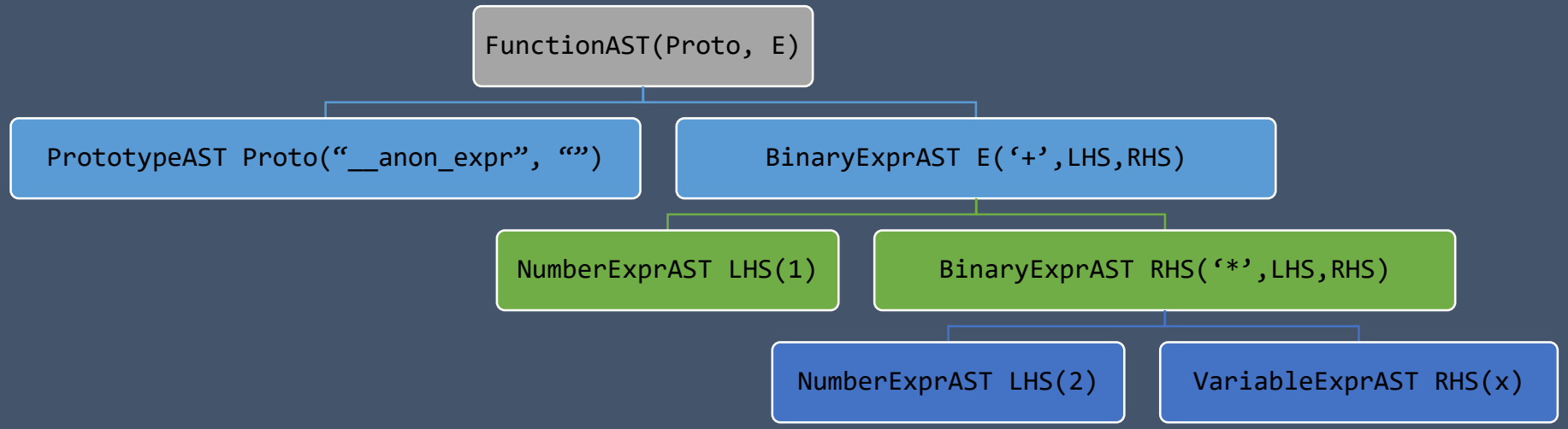
```
static void HandleDefinition() { // 顶层函数处理
  if (ParseDefinition()) {
    fprintf(stderr, "Parsed a function definition.\n");
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

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

static void HandleTopLevelExpression() { // 处理顶层表达式，输出相关提示
  if (ParseTopLevelExpr()) { // 判断是否解析成功顶层表达式
    fprintf(stderr, "Parsed a top-level expr\n");
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

// 主循环
/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (true) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
```

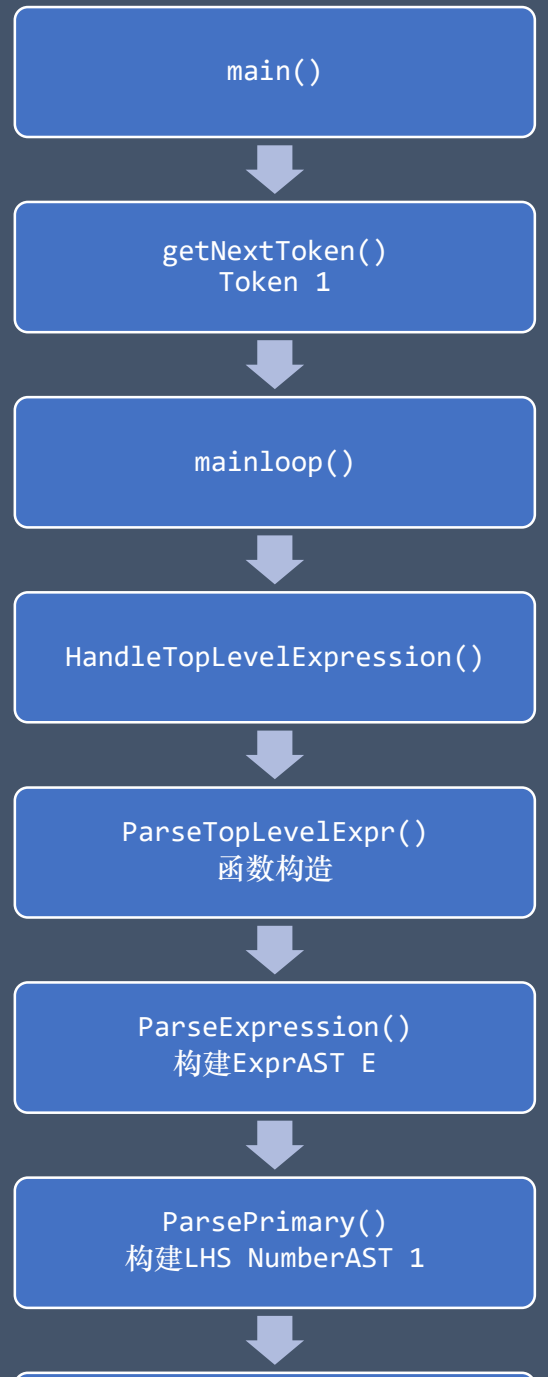
```
}

static void HandleTopLevelExpression() { // 处理顶层表达式，输出相关提示
  if (ParseTopLevelExpr()) { // 判断是否解析成功顶层表达式
    fprintf(stderr, "Parsed a top-level expr\n");
  } else {
    // Skip token for error recovery.
    getNextToken();
  }
}

// 主循环
/// top ::= definition | external | expression | ';'
static void MainLoop() {
  while (true) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof: // Ctrl+D / Ctrl+Z 结束程序
      return;
    case ';': // 结束语句
      getNextToken();
      break;
    case tok_def:
      HandleDefinition(); // 处理函数定义
      break;
    case tok_extern: // 处理extern
      HandleExtern();
      break;
    default:
      HandleTopLevelExpression(); // 顶层表达式结点处理
      break;
    }
  }
}
```

8 主函数

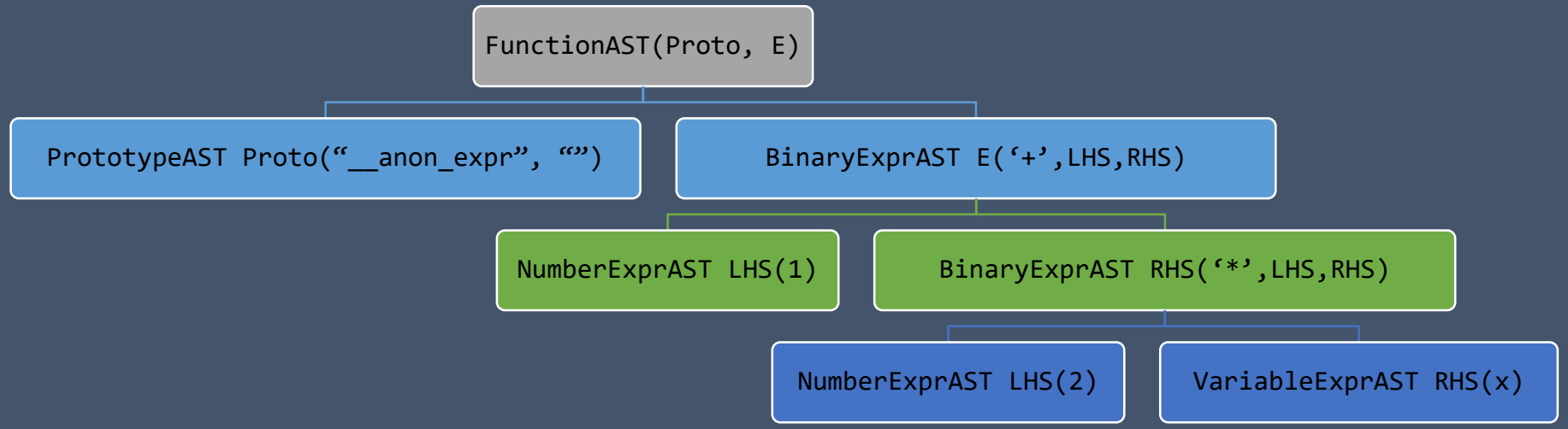
```
int main() {
  // 设置一下各个操作符的优先级,1最小
  // 操作符到优先级数字的映射
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;

  fprintf(stderr, "ready> ");
  getNextToken(); // 先为CurTok获取一个值

  MainLoop(); // 主循环

  return 0;
}
```

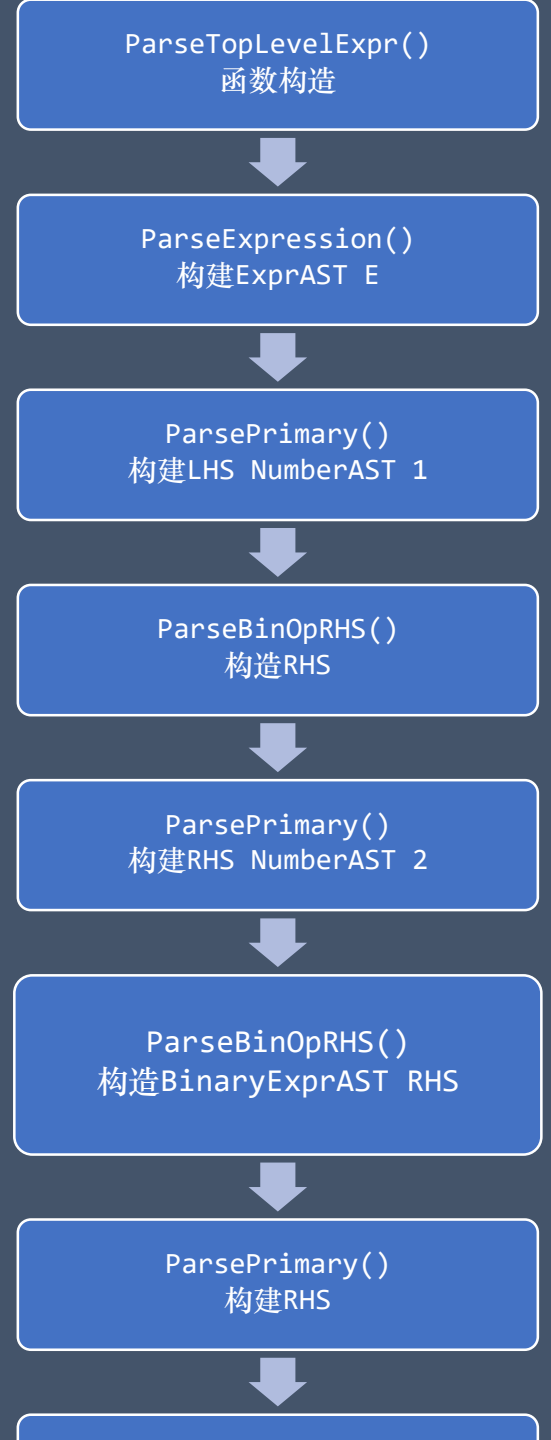
代码解析实例：1+2*x;


FunctionAST(Proto, E)
PrototypeAST Proto(“__anon_expr”, “”)
BinaryExprAST E(‘+’,LHS,RHS)
NumberExprAST LHS(1)
BinaryExprAST RHS(‘*’,LHS,RHS)
NumberExprAST LHS(2)
VariableExprAST RHS(x)

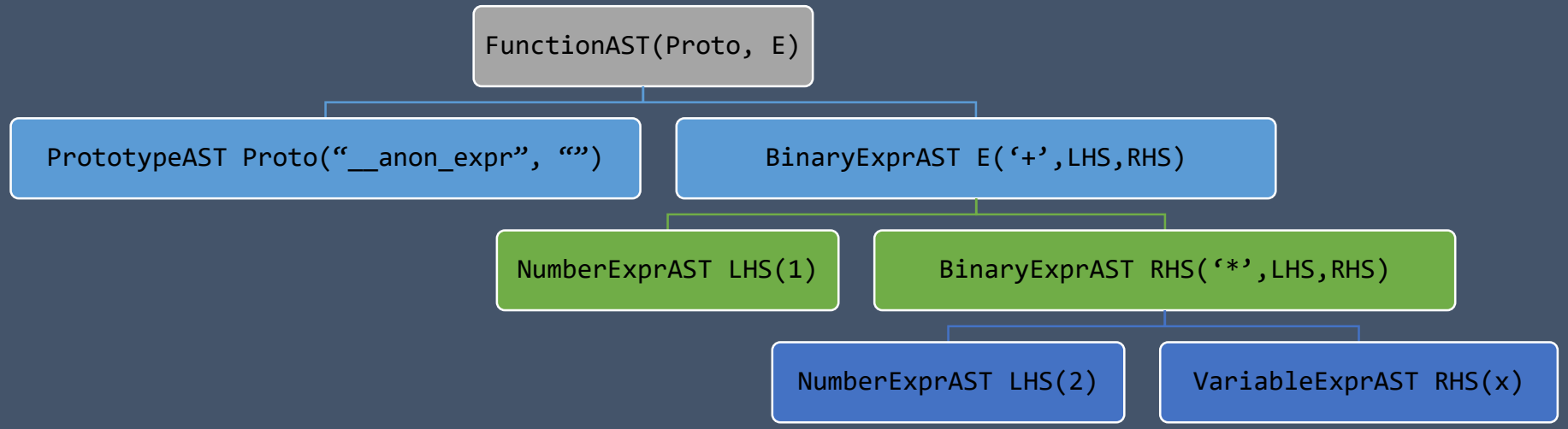

main()
getNextToken()
Token 1
mainloop()
HandleTopLevelExpression()
ParseTopLevelExpr()
函数构造
ParseExpression()
构建ExprAST E
ParsePrimary()
构建LHS NumberAST 1

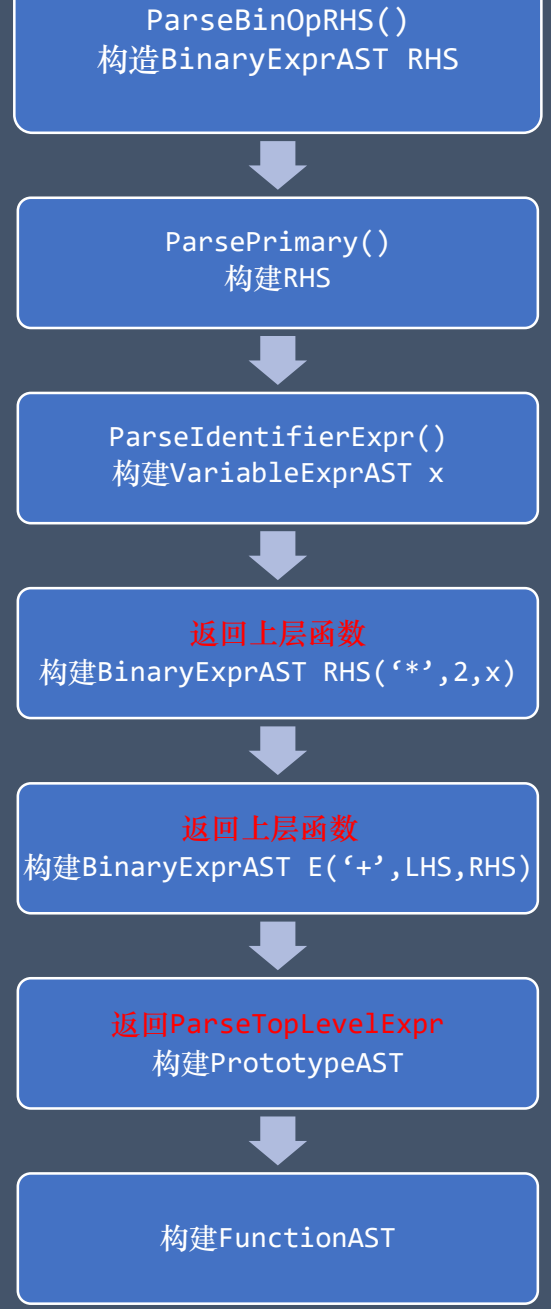
代码解析实例：1+2*x;


FunctionAST(Proto, E)
PrototypeAST Proto(“__anon_expr”, “”)
BinaryExprAST E(‘+’,LHS,RHS)
NumberExprAST LHS(1)
BinaryExprAST RHS(‘*’,LHS,RHS)
NumberExprAST LHS(2)
VariableExprAST RHS(x)


ParseTopLevelExpr()
函数构造
ParseExpression()
构建ExprAST E
ParsePrimary()
构建LHS NumberAST 1
ParseBinOpRHS()
构造RHS
ParsePrimary()
构建RHS NumberAST 2
ParseBinOpRHS()
构造BinaryExprAST RHS
ParsePrimary()
构建RHS

代码解析实例：1+2*x;


FunctionAST(Proto, E)
PrototypeAST Proto(“__anon_expr”, “”)
BinaryExprAST E(‘+’,LHS,RHS)
NumberExprAST LHS(1)
BinaryExprAST RHS(‘*’,LHS,RHS)
NumberExprAST LHS(2)
VariableExprAST RHS(x)


ParseBinOpRHS()
构造BinaryExprAST RHS
ParsePrimary()
构建RHS
ParseIdentifierExpr()
构建VariableExprAST x
返回上层函数
构建BinaryExprAST RHS(‘*’,2,x)
返回上层函数
构建BinaryExprAST E(‘+’,LHS,RHS)
返回ParseTopLevelExpr
构建PrototypeAST
构建FunctionAST