IR是方舟编译器的核心，跨平台的原因所在。

方舟编译器的ir结构是层级的，越上层的ir越接近高级语言，越下层的语言越接近机器语言。

对ir进行处理，包括词法分析、语法分析、优化等操作。

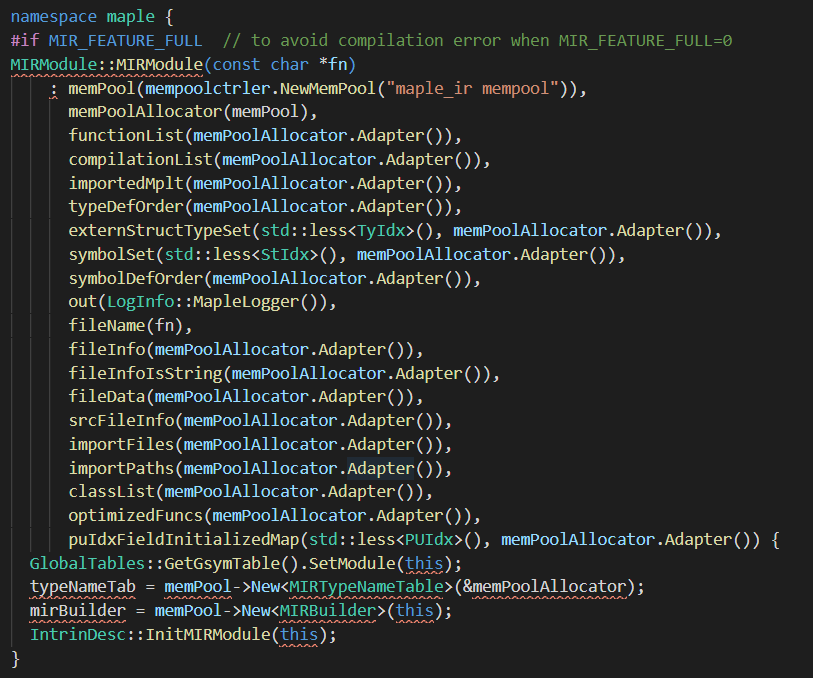
首先要知道ir的结构，由module、function到语句。

都在命名空间maple下。

Module对应一个源文件，支持不同的语言。

声明和实现分别在

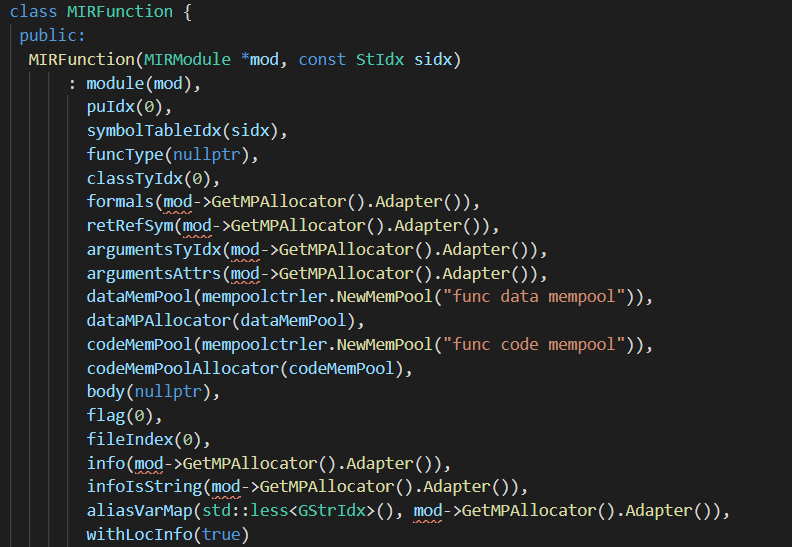
src\maple\_ir\include\mir\_module.h与src\maple\_ir\src\mir\_module.cpp中



Function：

声明和实现分别在

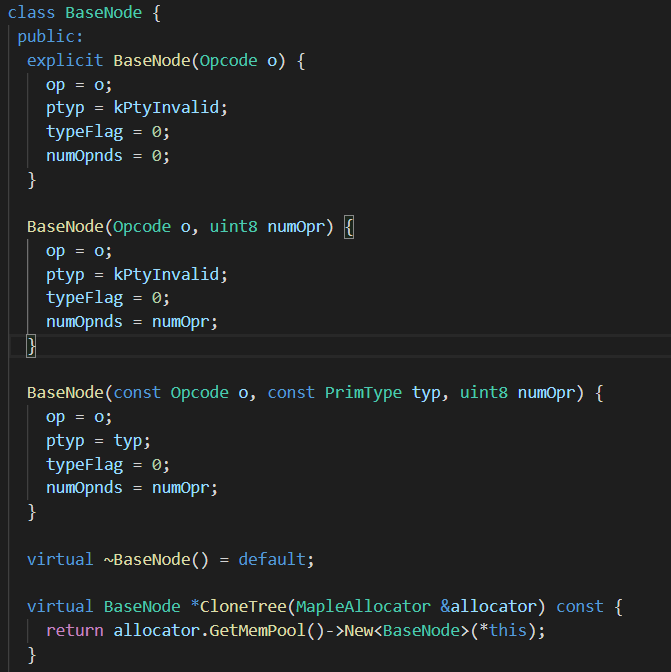
src\maple\_ir\include\mir\_function.h与src\maple\_ir\src\mir\_function.cpp中



Node:

声明和实现分别在

src\maple\_ir\include\mir\_nodes.h与src\maple\_ir\src\mir\_nodes.cpp中



注：explicit：关闭该类的隐式转换。

这是一个虚类，向下不断衍生出各种具体的语法树结点：

Unary即一元。

class UnaryNode : public BaseNode

class ExtractbitsNode : public UnaryNode

class TypeCvtNode : public UnaryNode

class GCMallocNode : public BaseNode

GCmalloc即内存分配（我们知道ir支持静态内存回收）。

Mir\_nodes.h有3000行之多，不赘述。包含了所有可能用到语法树结点。

词法分析由lexer完成。

src\maple\_ir\src\lexer.cpp

用到的token定义在tokens.h文件中。

lexer服务于parser，共同对源程序进行扫描。每当parser需要获取当前词素的信息时，就调用lexer提供的函数。简而言之，使用一趟扫描完成词法分析和语法分析的工作。

 private:

  MIRModule &module;

使用哈希表存放词素和类型的关系，如下：

 MapleUnorderedMap<std::string, TokenKind> keywordMap;

主函数为

TokenKind MIRLexer::LexToken(void)

也是使用switch，使用指针扫描源程序。

不同编译器，词法分析程序都大同小异。值得注意的有这么几点（写得比我们好的）：

面向的是maple IR，可以从IR识别为token，也可以由token反向生成IR:（使用name属性）  
void MIRLexer::GenName() {

  uint32 startIdx = curIdx;

  char c = GetCharAtWithUpperCheck(curIdx);

  if (isalnum(c) || c < 0 || c == '\_' || c == '$' || c == '@') {

    c = GetNextCurrentCharWithUpperCheck();

  }

  char cp = GetCharAtWithLowerCheck(curIdx - 1);

  if (c == '@' && (cp == 'h' || cp == 'f')) {

    // special pattern for exception handling labels: catch or finally

    c = GetNextCurrentCharWithUpperCheck();

  }

  while (isalnum(c) || c < 0 || c == '\_' || c == '$' || c == ';' || c == '/' || c == '|' || c == '.' || c == '?' ||

         c == '@') {

    c = GetNextCurrentCharWithUpperCheck();

  }

  name = line.substr(startIdx, curIdx - startIdx);

}

有点长，就是根据token类型反向生成IR词素。

std::string MIRLexer::GetTokenString() const {

  std::string temp;

  switch (kind) {

    case kTkGname: {

      temp = "$";

      temp.append(name);

      return temp;

    }

    case kTkLname:

    case kTkPreg: {

      temp = "%";

      temp.append(name);

      return temp;

    }

    case kTkSpecialreg: {

      temp = "%%";

      temp.append(name);

      return temp;

    }

    case TK\_label: {

      temp = "@";

      temp.append(name);

      return temp;

    }

做的更加详细。我感觉以后学弟学妹有的抄了。

包括字符串转义字符处理、多种进制转换。

ir的基本类型包含各种int、float，但在词法分析中优先使用最大范围来表示。

如int统一使用int64，uint统一使用unit32。  
这是基本类型定义：（在types\_def.h）

using int8 = std::int8\_t;

using int16 = std::int16\_t;

using int32 = std::int32\_t;

using int64 = std::int64\_t;

using uint8 = std::uint8\_t;

using uint16 = std::uint16\_t;

using uint32 = std::uint32\_t;

using uint64 = std::uint64\_t;

ir词法分析的结果，存放在类变量之中，包括float、double、int以及string。

注：这部分get函数都用const修饰，保证了变量的安全性。

  const std::string &GetName() const {

    return name;

  }

  int64 GetTheIntVal() const {

    return theIntVal;

  }

  float GetTheFloatVal() const {

    return theFloatVal;

  }

  double GetTheDoubleVal() const {

    return theDoubleVal;

  }

Parser：

Parser直接被driver中的main函数调用。调用如下：

maple::MIRParser theParser(module);

theParser.ParseMIR()

存放在mir\_parser、parser中。

Mir\_parser针对的是ir。Parser针对的是java。（这里存疑。因为我确实看到parser里的具体函数都是根据java语法来的，但它获取的还是ir的词法分析输出结果，而且这俩还经常互相调用）。

语法分析的目的是判断记号流是否符合ir语法，并给出错误信息。Parsemir返回一个布尔值。

整体来看，可以分为三个层次来完成，分别是bl ock、statement、expression。

越上层越抽象，越底层越具体。Ex：var->

整体采用递归调用的模式进行语法分析。每一个层级都拥有自己的哈希表，将不同tokentype和对应处理函数关联起来。

错误信息使用全局的error函数以及emiterror、emitwarning函数生成。

简单的处理函数如下：

这个函数检查是否是一个类。

注：st是新符号。参数kscopeglobal表示存放在全局变量的符号表。

Assert是C++用宏实现的断言。这里用于检查符号是否成功被创建，否则会触发abort。

后续

bool MIRParser::ParseMIRForClass() {

  MIRSymbol \*st = GlobalTables::GetGsymTable().CreateSymbol(kScopeGlobal);

  ASSERT(st != nullptr, "st nullptr check");

  st->SetStorageClass(kScInvalid);

  st->SetSKind(kStJavaClass);

  if (!ParseJavaClassInterface(\*st, true)) {

    return false;

  }

  if (!GlobalTables::GetGsymTable().AddToStringSymbolMap(\*st)) {

    Error("duplicate symbol name used in javaclass at ");

    return false;

  }

  return true;

}

稍复杂一些的例子：

parseStmtBlock用于处理函数体。即{}内的语句。

在当前module中创建一个新的语法树结点blk。后续识别出新的结点就作为子节点加入进去。

使用while循环完成主体部分的识别工作，不断使用lexer获取下一个记号，根据tokentype取用哈希表中函数处理之。直至遇到右大括号，完成处理。

bool MIRParser::ParseStmtBlock(BlockNodePtr &blk) {

  if (lexer.GetTokenKind() != kTkLbrace) {

    Error("expect { for func body but get ");

    return false;

  }

  blk = mod.CurFuncCodeMemPool()->New<BlockNode>();

  MIRFunction \*fn = mod.CurFunction();

  paramCurrFuncForParseStmtBlock = fn;

  lexer.NextToken();

  // Insert \_mcount for PI.

  if (mod.GetWithProfileInfo()) {

    StmtNode \*stmtt = nullptr;

    if (!ParseStmtCallMcount(stmtt)) {

      return false;

    }

    blk->AddStatement(stmtt);

  }

  while (true) {

    TokenKind stmtTk = lexer.GetTokenKind();

    // calculate the mpl file line number mplNum here to get accurate result

    uint32 mplNum = lexer.GetLineNum();

    if (IsStatement(stmtTk)) {

      ParseStmtBlockForSeenComment(blk, mplNum);

      StmtNode \*stmt = nullptr;

      if (!ParseStatement(stmt)) {

        Error("ParseStmtBlock failed when parsing a statement");

        return false;

      }

      if (stmt != nullptr) {  // stmt is nullptr if it is a LOC

        SetSrcPos(stmt, mplNum);

        blk->AddStatement(stmt);

      }

    } else {

      std::map<TokenKind, FuncPtrParseStmtBlock>::iterator itFuncPtr = funcPtrMapForParseStmtBlock.find(stmtTk);

      if (itFuncPtr == funcPtrMapForParseStmtBlock.end()) {

        if (stmtTk == kTkRbrace) {

          ParseStmtBlockForSeenComment(blk, mplNum);

          lexer.NextToken();

          return true;

        } else {

          Error("expect } or var or statement for func body but get ");

          return false;

        }

      } else {

        if (!(this->\*(itFuncPtr->second))()) {

          return false;

        }

      }

    }

  }

}

Parser调用关系梳理：

bool ParseMIR(std::ifstream&);  // the main entry point

ParseMIR->ParseMIRForFunc()->ParseFunction->ParseStmtBlock->ParseStmt

Parse作为整个词法分析的入口，最终在driver中被调用。

if (theParser.ParseMIR())

Phase说明。

Phase是方舟使用的代码优化机制。

分两类，module和mefun。

创建之后在def中注册，即可在代码中使用以优化代码。