AI编译器系列

LLVM架构和原理

ZOMI

Talk Overview

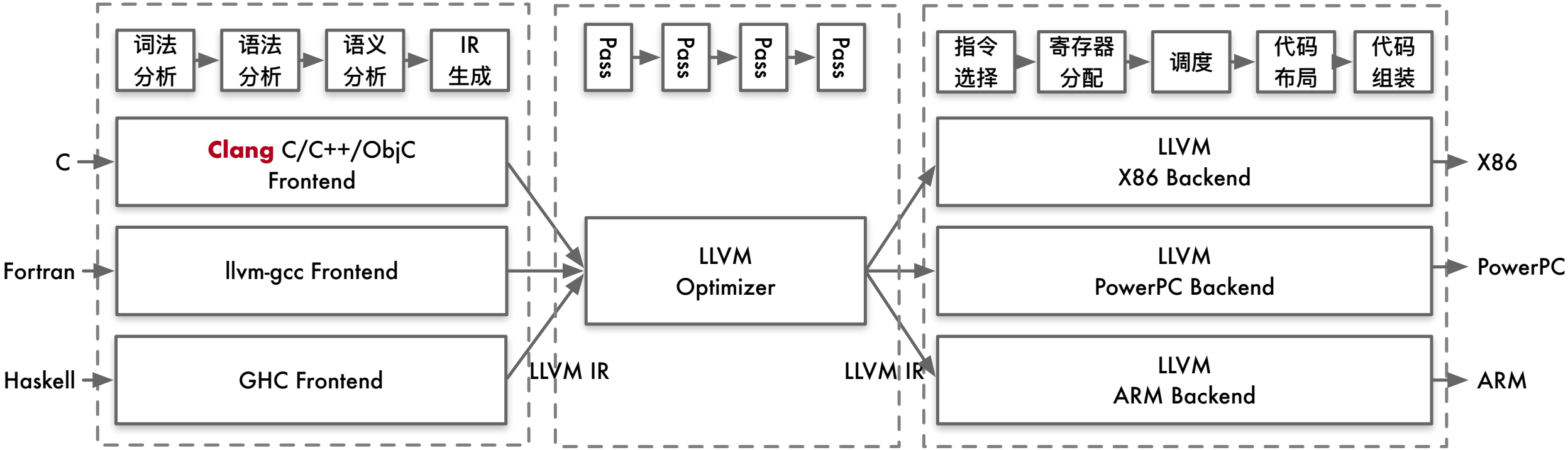
1. **传统编译器**
   - History of Compiler - 编译器的发展
   - GCC process and principle – GCC 编译过程和原理
   - LLVM/Clang process and principle – LLVM 架构和原理
2. **AI编译器**
   - History of AI Compiler – AI编译器的发展
   - Base Common architecture – AI编译器的通用架构
   - Different and challenge of the future – 与传统编译器的区别，未来的挑战与思考

Talk Overview

LLVM/Clang process and principle – LLVM 架构和原理

- LLVM 项目发展历史
- LLVM 基本设计原则和架构
- LLVM 中间表示 LLVM IR
- LLVM 前端过程
- LLVM 中间优化
- LLVM 后端生成
- 基于 LLVM 项目

LLVM Architecture

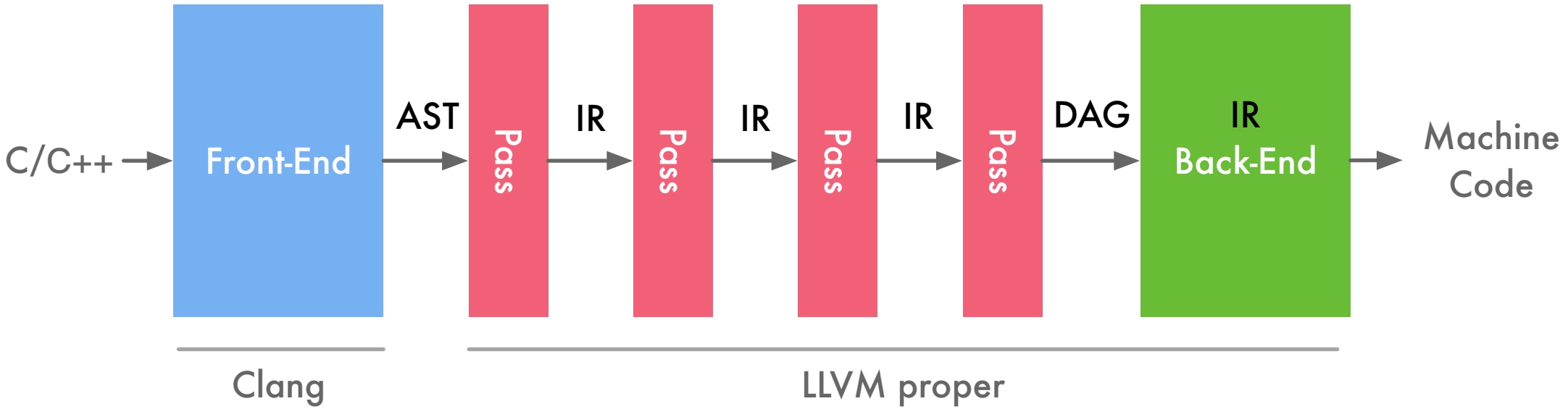


LLVM IR

中间表达

LLVM IR

这并不意味着LLVM使用单一 IR 表示方式， 在编译不同阶段会采用不同的数据结构：



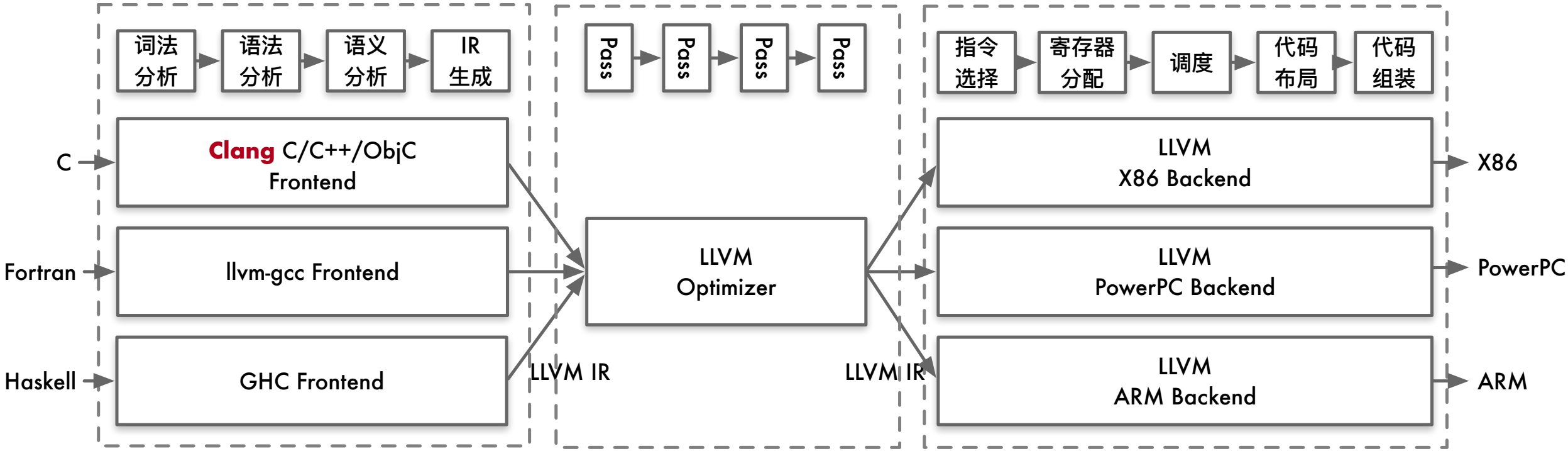
LLVM IR 内存模型

Module	Module类聚合了整个翻译单元用到的所有数据，它是LLVM术语中的“module”的同义词。它声明了Module::iterator typedef，作为遍历这个模块中的函数的简便方法。你可以用begin()和end()方法获取这些迭代器。
Function	Function类包含有关函数定义和声明的所有对象。对于声明来说（用isDeclaration()检查它是否为声明），它仅包含函数原型。无论定义或者声明，它都包含函数参数的列表，可通过getArgumentList()方法或者arg_begin()和arg_end()这对方法访问它。你可以通过Function::arg_iterator typedef遍历它们。如果Function对象代表函数定义，你可以通过这样的语句遍历它的内容：for (Function::iterator i = function.begin(), e = function.end(); i != e; ++i)，你将遍历它的基本块。
BasicBlock	BasicBlock类封装了LLVM指令序列，可通过begin()/end()访问它们。你可以利用getTerminator()方法直接访问它的最后一条指令，你还可以用一些辅助函数遍历CFG，例如通过getSinglePredecessor()访问前驱基本块，当一个基本块有单一前驱时。然而，如果它有多个前驱基本块，就需要自己遍历前驱列表，这也不难，你只要逐个遍历基本块，查看它们的终结指令的目标基本块。
Instruction	Instruction类表示LLVM IR的运算原子，一个单一的指令。利用一些方法可获得高层级的断言，例如isAssociative()，isCommutative()，isIdempotent()，和isTerminator()，但是它的精确的功能可通过getOpcode()获知，它返回llvm::Instruction枚举的一个成员，代表了LLVM IR opcode。可通过op_begin()和op_end()这对方法访问它的操作数，它从User超类继承得到。

LLVM 前端

LLVM Architecture

编译器前端将源代码变换为编译器的中间表示 LLVM IR，它处于代码生成之前，后者是针对具体目标的。



Lexical analysis 词法分析

- 前端的第一个步骤处理源代码的文本输入，将语言结构分解为一组单词和标记，去除注释、空白、制表符等。每个单词或者标记必须属于语言子集，语言的保留字被变换为编译器内部表示。

```
int 'int'        [StartOfLine]  Loc=<hello.c:5:1>
identifier 'main'        [LeadingSpace] Loc=<hello.c:5:5>
l_paren '('              Loc=<hello.c:5:9>
void 'void'              Loc=<hello.c:5:10>
r_paren ')'              Loc=<hello.c:5:14>
l_brace '{'              Loc=<hello.c:5:15>
identifier 'printf'      [StartOfLine] [LeadingSpace]   Loc=<hello.c:6:5>
l_paren '('              Loc=<hello.c:6:11>
l_paren '('              Loc=<hello.c:6:12 <Spelling=hello.c:3:19>>
string_literal '"hello world\n"'               Loc=<hello.c:6:12 <Spelling=hello.c:3:20>>
r_paren ')'              Loc=<hello.c:6:12 <Spelling=hello.c:3:35>>
r_paren ')'              Loc=<hello.c:6:21>
semi ';'                 Loc=<hello.c:6:22>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<hello.c:7:5>
numeric_constant '0'     [LeadingSpace] Loc=<hello.c:7:12>
semi ';'                 Loc=<hello.c:7:13>
r_brace '}'      [StartOfLine]  Loc=<hello.c:8:1>
eof ''         Loc=<hello.c:8:2>
```

$ clang -cc1 -dump-tokens hello.c

Syntactic analysis 语法分析

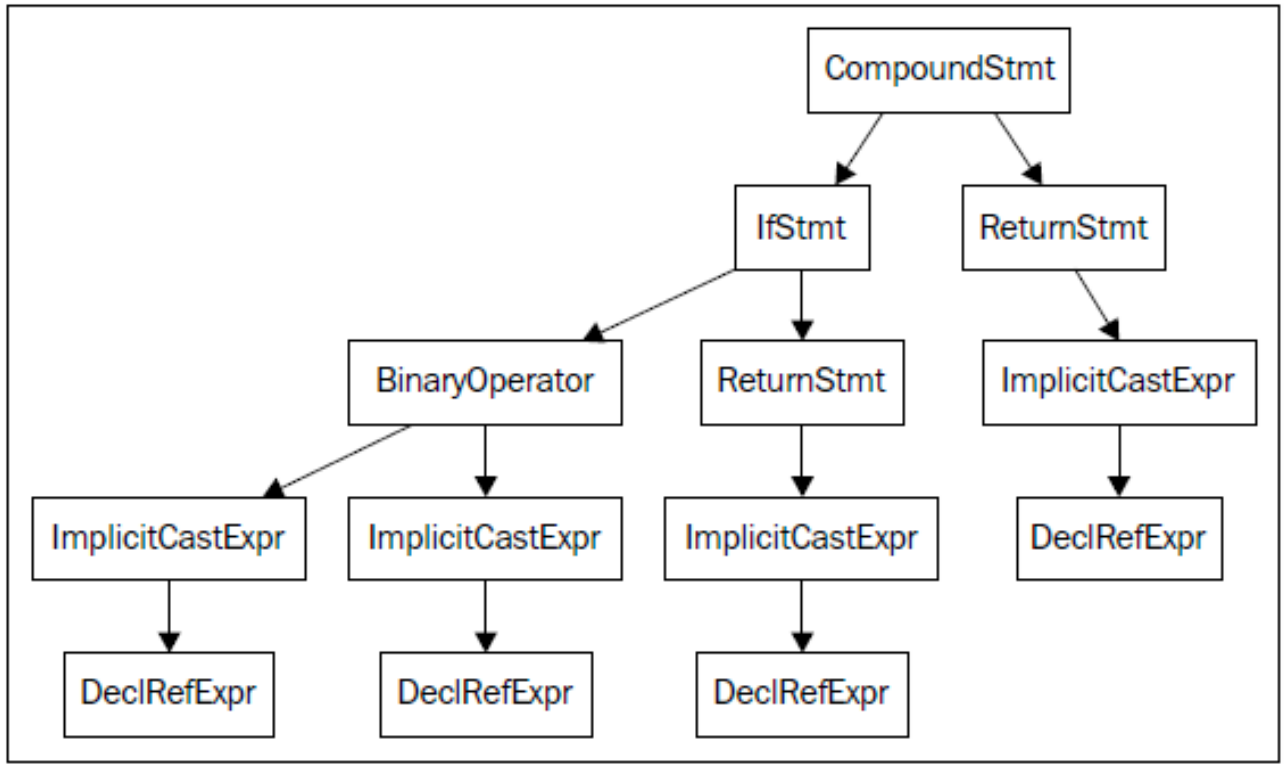
- 分组标记以形成表达式、语句、函数体等。检查一组标记是否有意义，考虑代码物理布局，未分析代码的意思，就像英语中的语法分析，不关心你说了什么，只考虑句子是否正确，并输出语法树（AST）。

```
|-ParmVarDecl 0x7fde088fb320 <col:52> col:58 'size_t':'unsigned long'
|-ParmVarDecl 0x7fde088fb3a0 <line:62:7, col:18> col:30 'const char *restrict'
`-ParmVarDecl 0x7fde088fb418 <col:32> col:39 'struct __va_list_tag *':'struct __va_list_tag *'
-FunctionDecl 0x7fde088cb270 <line:70:1, line:71:40> line:70:12 __vsnprintf_chk 'int (char *restrict, size_t, int, size_t, const ch
|-ParmVarDecl 0x7fde088fb650 <col:29, col:34> col:46 'char *restrict'
|-ParmVarDecl 0x7fde088fb6c8 <col:48> col:54 'size_t':'unsigned long'
|-ParmVarDecl 0x7fde088fb748 <col:56> col:59 'int'
|-ParmVarDecl 0x7fde088cb000 <col:61> col:67 'size_t':'unsigned long'
|-ParmVarDecl 0x7fde088cb080 <line:71:8, col:19> col:31 'const char *restrict'
`-ParmVarDecl 0x7fde088cb0f8 <col:33> col:40 'struct __va_list_tag *':'struct __va_list_tag *'
-FunctionDecl 0x7fde088cb3e0 <hello.c:5:1, line:8:1> line:5:5 main 'int (void)'
-CompoundStmt 0x7fde088cb608 <col:15, line:8:1>
  |-CallExpr 0x7fde088cb580 <line:6:5, col:21> 'int'
  | |-ImplicitCastExpr 0x7fde088cb568 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
  | | `-DeclRefExpr 0x7fde088cb480 <col:5> 'int (const char *, ...)' Function 0x7fde088e9548 'printf' 'int (const char *, ...)'
  | `-ImplicitCastExpr 0x7fde088cb5c0 <line:3:19, col:35> 'const char *' <NoOp>
  |   `-ImplicitCastExpr 0x7fde088cb5a8 <col:19, col:35> 'char *' <ArrayToPointerDecay>
  |     `-ParenExpr 0x7fde088cb500 <col:19, col:35> 'char [13]' lvalue
  |       `-StringLiteral 0x7fde088cb4d8 <col:20> 'char [13]' lvalue "hello world\n"
  `-ReturnStmt 0x7fde088cb5f8 <line:7:5, col:12>
    `-IntegerLiteral 0x7fde088cb5d8 <col:12> 'int' 0
```


$ clang -fsyntax-only -Xclang -ast-dump hello.c

Syntactic analysis 语法分析

- 分组标记以形成表达式、语句、函数体等。检查一组标记是否有意义，考虑代码物理布局，未分析代码的意思，就像英语中的语法分析，不关心你说了什么，只考虑句子是否正确，并输出语法树（AST）。



$ clang -fsyntax-only -Xclang -ast-dump hello.c

Semantic analysis 语义分析

- 借助符号表检验代码没有违背语言类型系统。符号表存储标识符和其各自的类型之间的映射，以及其它内容。类型检查的一种直觉的方法是，在解析之后，遍历AST的同时从符号表收集关于类型的信息。

```
3    #define HELLOWORD ("hello world\n")
4
5    int a[4];
6    int a[5];
7
```

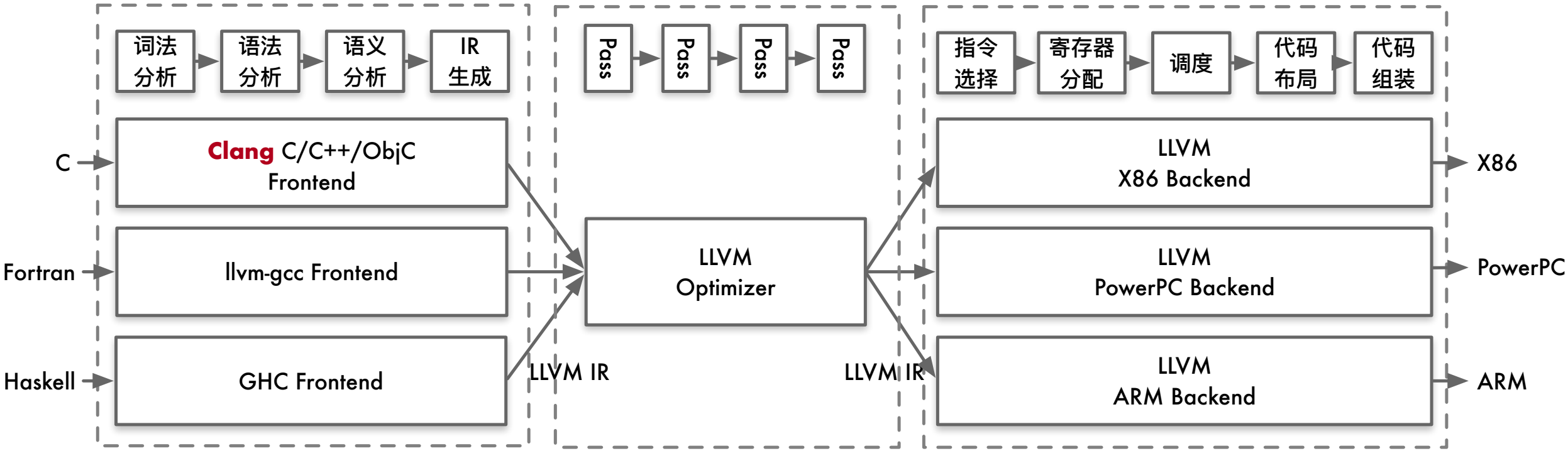
```
hello.c:6:5: error: redefinition of 'a' with a different type: 'int [5]' vs 'int [4]'
int a[5];
    ^
hello.c:5:5: note: previous definition is here
int a[4];
    ^
1 error generated.
```

$ clang -c hello.c

LLVM 优化层

LLVM Architecture

目标无关优化，理解优化操作，实际上就是理解 IR 如何在 pass 流水线中被修改，这需要知道每个 pass 执行的修改，还有各个 pass 是以什么顺序被执行。



Finding Pass

优化通常由分析 Pass 和转换 Pass 组成。

- 分析 Pass：负责发掘性质和优化机会；
- 转换 Pass：生成必需的数据结构，后续为后者所用；

```
===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0003 seconds (0.0003 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  ---Instr---  --- Name ---
   0.0001 ( 72.7%)   0.0000 ( 68.2%)   0.0002 ( 71.6%)   0.0002 ( 70.1%)     481094  BitcodeWriterPass
   0.0000 ( 11.7%)   0.0000 ( 13.6%)   0.0000 ( 12.2%)   0.0000 ( 12.7%)     127722  VerifierPass
   0.0000 (  5.4%)   0.0000 ( 12.1%)   0.0000 (  7.0%)   0.0000 (  7.3%)      78056  RequireAnalysisPass<llvm::DominatorTreeAnalysis, llvm::Function>
   0.0000 (  7.8%)   0.0000 (  3.0%)   0.0000 (  6.6%)   0.0000 (  6.8%)      56681  VerifierAnalysis
   0.0000 (  2.4%)   0.0000 (  3.0%)   0.0000 (  2.6%)   0.0000 (  3.2%)      33040  DominatorTreeAnalysis
   0.0002 (100.0%)   0.0001 (100.0%)   0.0003 (100.0%)   0.0003 (100.0%)     776593  Total
===-------------------------------------------------------------------------===
                                LLVM IR Parsing
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0003 seconds (0.0003 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  ---Instr---  --- Name ---
   0.0003 (100.0%)   0.0001 (100.0%)   0.0003 (100.0%)   0.0003 (100.0%)     869304  Parse IR
   0.0003 (100.0%)   0.0001 (100.0%)   0.0003 (100.0%)   0.0003 (100.0%)     869304  Total
```

$ opt hello.bc -instcount -time-passes -domtree -o hello-tmp.bc -stats

$ opt hello.bc -instcount -time-passes -domtree -o hello-tmp.bc -stats

Finding Pass

https://llvm.org/docs/Passes.html

优化通常由分析 Pass 和转换 Pass 组成。

- **分析 Pass**：负责发掘性质和优化机会；
- **转换 Pass**：生成必需的数据结构，后续为后者所用；

- -adce: Aggressive Dead Code Elimination

积极的死代码消除。此pass类似于DCE，但它假定值是死的，除非得到其他证明。这类似于SCCP，除了用于值的活动性。

- -constmerge: Merge Duplicate Global Constants

将重复的全局常量合并到一个共享的常量中。这是有用的，一些passes在程序中插入许多字符串常量，不管现有字符串是否可用。

Understand Pass Relation

在转换Pass和分析Pass之间，有两种主要的依赖类型：

- **显式依赖**：转换Pass需要一种分析，则Pass管理器自动地安排它所依赖的分析Pass在它之前运行；

```
DominatorTree &DT = getAnalysis<DominatorTree>(Func);
```

- **隐式依赖**：转换或者分析Pass要求IR代码运用特定表达式。需要手动地以正确的顺序把这个Pass加到Pass队列中，通过命令行工具（clang或者opt）或者Pass管理器。

Pass API

Pass类是实现优化的主要资源。然而，我们从不直接使用它，而是通过清楚的子类使用它。当实现一个Pass时，你应该选择适合你的Pass的最佳粒度，适合此粒度的最佳子类，例如基于函数、模块、循环、强联通区域，等等。常见的这些子类如下：

- ModulePass
- FunctionPass
- BasicBlockPass