编译原理实验报告

——PCAT 编译器 plang

1 概述

在《编译》的 Lab1 与 Lab2 中，已经完成了对 PCAT 语言的词法分析、语法分析，并生成了抽象语法树供后续模块使用。在本次实验中，将实现 PCAT 编译器 plang，利用语法分析器产生的抽象语法树进行进一步的处理，完成中间码生成，实现编译器前端；并进一步生成汇编代码，最终生成可执行文件。

本实验已经完整的完成了一个简单编译器所包含的全部功能，因此主要包含但不限于以下特性：

- 抽象语法树生成；
- 类型检查、复合类型系统；
- 函数框帧；
- 中间表示生成；
- 汇编生成、可执行文件生成。

在后续部分，本文档将从使用方法、系统设计，以及实现三个角度来介绍 plang 编译器。

2 plang 使用方法

Plang 是本实验中实现的 PCAT 编译器，它接受 PCAT 语言的源代码，并可输出抽象语法树、中间表示、汇编代码，以及二进制可执行文件。Plang 依赖于编译器基础设施 LLVM，以及基于 LLVM 的 C 编译器前端 clang，前者用于实现编译器基础框架，后者用于把汇编指令编译并链接为可执行文件。下面将介绍 plang 的编译与使用方法。

2.1 使用方法

Plang 的基本命令行格式如下：

```
plang [options] <input file>
```

其中[options]为可选参数，<input file>为输入源代码的路径。如果省略路径，plant 将从标准输入中读取代码。在默认条件下，plang 将输出源码编译后得到的可执行文件。例如，若执行：

```
plang demo.pcat
```

将会生成一个可执行文件“demo”。

下面介绍 plang 的可选参数。与绝大多数 GNU 系列编译器一样，plang 提供了一系列可选参数，使用“-help”可以令 plang 显示可选参数及其说明。

- -S：输出汇编代码，文件后缀为“.S”；
- -c：输出 object 文件，文件后缀为“.o”；
- -emit-llvm：输出 LLVM 中间表示，文件后缀为“.ll”；
- -emit-llvm-stdout：向标准输出流（stdout）中输出 LLVM 中间表示；
- -help：显示命令说明；
- -o=<filename>：指定输出的二进制可执行文件的名称，若此选项未指定则与输入文件名去掉后缀保持一致，此选项同时会影响其他选项的输出文件名；
- -show-ast：向标准输出流（stdout）中输出抽象语法树；
- -version：显示 plang 的版本信息。

以下是一个样例，plang 接受源码文件“fib.pcat”，并生成可执行文件“fibonacci”，汇编代码文件“fibonacci.S”，LLVM 中间表示文件“fibonacci.ll”：

```
plang -o=fibonacci -S -emit-llvm fib.pcat
```

2.2 编译方法

在编译 plang 之前，需要确认系统中已经正确安装了如下组件：

- 支持 C++11 的编译器，例如最新版本的 clang、GCC；
- 较新版本的 LLVM，包括其头文件与动态链接库；
- 可以使用的 clang 编译器。

这是由于 plang 使用了 C++11 的特性编写，并基于 LLVM 实现。在组件完整的情况下，也许根据 LLVM 的版本不同，需要对 Makefile 文件稍加修改，使编译器能寻找到 LLVM 的安装目录即可。Makefile 文件的结构非常简单，因此不再赘述具体的修改方法。确保以上步骤正确无误后，只需执行如下命令即可编译 plang：

```
make plang
```

3. 系统设计

Plang 基于 LLVM 开发，因此其设计框架在一定程度上由 LLVM 的特性决定。因此在这一部分将首先简要介绍 LLVM 的部分特性，然后介绍 plang 的框架。

3.1 编译器基础设施 LLVM

LLVM 全称为底层虚拟机（Low Level Virtual Machine），但它并不是一个如同 VMWare 的虚拟机程序。LLVM 是为了完成编译器模块化开发与全阶段优化而建立的框架，因而被称为编译器基础设施。它的基础是精心设计的一套底层中间表示指令，被称为 LLVM-IR。

与其他编译器不同的一点是，LLVM-IR 经过了非常仔细的设计，既足够底层，又保留了足够多的信息。由于 LLVM-IR 的底层特性，任何一种计算机语言都可以产生此种形式的中间代码，而不失任何灵活性；而由于保留的信息足够丰富，使得全阶段优化、语言翻译，以及高级语言特性的实现成为可能。基于这套中间表示，LLVM 系统提供了完善的类型系统、指令系统、优化系统、汇编系统等模块，可以供任何编译器开发者使用。

首先是类型系统。LLVM 提供的类型系统包含基本类型、函数类型、复合类型等常见类型。其中基本类型包括整数与实数，并可定义其位数或精度；函数类型由参数表与返回值两个部分组成，表示一个函数的定义；复合类型包含向量类型、数组类型、结构体类型，通过这一类型可以构造出任意的复杂类型。

指令系统是 LLVM 中另一个十分重要的模块。它提供了对 LLVM-IR 的抽象，利用指令系统，可以由开发者按照自己的需求自由生成 LLVM-IR。同时，指令系统也提供了对栈空间与堆空间的管理，plang 的全部变量都保存在指令系统分配的栈空间内。

最后是优化系统与汇编系统。当完成了 LLVM-IR 的生成后，就可以调用这两个模块进行最后的处理，生成出汇编指令，并进一步生成可执行文件了。LLVM 允许在优化阶段与汇编阶段中插入自己的插件，以控制优化与汇编生成的种种细节，这也为开发者提供了极大的灵活性。

3.2 plang 系统框架

Plang 主要由四个模块组成，分别为词法分析器、语法分析器、中间指令生成器，以及驱动程序。其中词法分析器接受源码流，输出 Token 序列；语法分析器接受 Token 序列，输出抽象语法树；中间指令生成器接受抽象语法树，输出 LLVM-IR；驱动程序是系统的控制部分，它负责协调前端组件，并调用 LLVM 生成汇编代码与二进制可执行文件。其结构如图 3.1 所示。

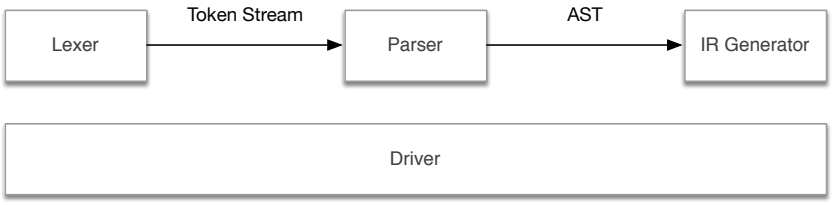

图 3.1 plang 系统结构

词法分析器与语法分析器在 Lab1 与 Lab2 中已经详细介绍过，本次实验主要完成后两部分。

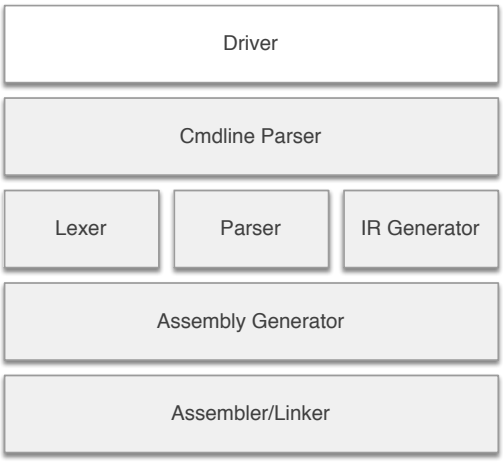

图 3.2 驱动程序结构

首先是驱动程序，其详细结构如图 3.2 所示。在执行程序时，驱动程序会分析命令行参数，然后读取文件，调用分析器与生成器进行 IR 与汇编代码的生成，最终调用汇编器与链接器生成本机机器指令，即二进制可执行文件。

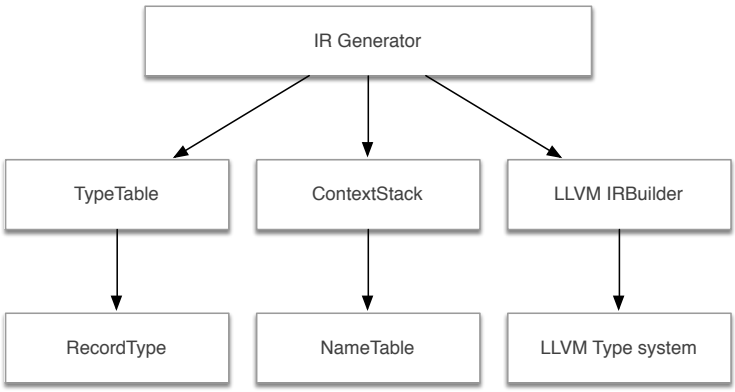

图 3.3 中间指令生成器结构

接下来是中间表示生成器，其详细结构如图 3.3 所示。中间表示生成器是本次实验的核心，它内部包含类型表、上下文栈、LLVM-IR 子系统三个子部分。其中类型表提供了对语言类型的记录功能，上下文栈提供了对作用域中符号名的查找支持，LLVM-IR 子系统提供了 IR 生成的必要辅助工具。而生成器的本身接受 PCAT 的抽象语法树，深度遍历整棵树，并在此过程中生成 IR 代码。

4 实现原理

在此部分将介绍 plang 的核心部分原理及实现。在低 3 节中已简要介绍了 plang 的系统设计，系统中最为关键的部分为 IR 生成与汇编、链接功能，下面主要介绍这两部分的实现。

4.1 NameTable

在 plang 中定义了符号表类型 NameTable。NameTable 实质上是一个类，其中包含了一个名称、值对象组成的字典，NameTable 所对应作用域的名称，以及过程实例对象。

作用域是指一个范围，在此范围之中的变量是有效的，具体到 PCAT 语言中，每个作用域都要么对应一个过程（Procedure），要么是全局范围的变量。这是因为一个过程中包含的局部变量声明在过程之外是不可见的，但在过程之则可见，因此一个过程定义了一个作用域，作用域的名称通常也与过程名一致。

NameTable 第二个重要的元素是过程实例对象。过程实例对象实际上是一个 LLVM 函数对象（llvm::Function），表示在 LLVM 类型系统中的一个函数。在 LLVM 类型系统中函数是一种基本类型，它作为中间代码的容器被创建出来，并插入到一个模块之中。当希望插入 IR 时，必须指定代码插入的位置，因此就需要指定一个函数作为 IR 的插入点。在函数中插

入代码时，也可以通过函数对象访问到其参数表中的值对象，以此实现参数传递。NameTable 的过程字段保存了此作用域对应的 LLVM 函数对象，以便后续代码生成时可以用于查询 IR 的插入点。实际上，NameTable 的过程对象这一字段间接地实现了继承属性。

NameTable 的第三个重要元素是名称、值对象字典。这个字典记录的没一个条目都表示一个变量（对于最外层过程而言为全局变量，对于内部过程而言为局部变量），其关键字类型是字符串，值类型是 LLVM 值对象。关键字即为变量的名称，在后续生成中，当需要查找变量的值或指针，则可以通过此字典实现查询。

需要进一步介绍的是 LLVM 值对象（llvm::Value），它表示 LLVM 中间代码中可以出现的任意一个值，如立即数、变量、中间变量、函数等，与三元表达式中出现的变量是类似的。在 NameTable 中，即可保存一个名字、值对象对，以便此后根据变量名来查询变量的值。

4.2 NameTableStack

NameTableStack 就是用 NameTable 组成的一个栈结构。在此栈之中，每一层都为一个 NameTable，在 IR 翻译的开始阶段，栈中只有一个 NameTable，作为全局变量的存放位置而出现；当翻译进行过程中，由于作用域是允许嵌套的，每当进入一个新的作用域，就会向栈中压入一个新的 NameTable 对象，在此作用域中出现的变量声明与函数参数均存入栈顶的 NameTable，以此类推。

当在任意一个作用域中查询一个变量的值时，只需要从栈顶开始向下搜索变量，若能够找到，则返回找到的第一个值对象；如果不能找到，则说明此变量不存在，可以抛出异常并进行错误处理。

4.3 TypeTable

与 NameTable 对应的另一个数据结构是 TypeTable，用于保存 PCAT 中全部已知的类型信息。TypeTable 包含一个字符串、类型对象字典，与一个字符串、StructDef 对象字典。前者注册了类型名字与类型对象的对应关系，其中类型对象是 LLVM 类型系统中的类型对象（llvm::Type），可以表示 IR 中的基本类型与更为复杂的复合类型。在 TypeTable 初始化时需要向此字典中添加 PCAT 基本类型 INTEGER、REAL、BOOLEAN，分别对应 LLVM 的基本类型 llvm::IntegerType、llvm::FPType、llvm::IntegerType<1>。

任何代码中定义的类型，都将首先利用 LLVM 类型系统构造出对应的类型对象，然后插入到 TypeTable 的类型字典之中。然而对于记录类型（RECORD），除了插入到类型字典之中，还需要记录下记录中每个字段在记录中的索引。为了记录额外的记录类型信息，需要创建 StructDef 对象。

StructDef 对象中保存了一个记录体中的必要信息。它包含了记录名，记录体中全部字段在记录体中的索引顺序与类型。在 LLVM 中，可以根据记录体中各个字段的类型创建一个复合结构体，当访问一个结构体变量中的字段时，就可以根据字段的索引号来获得指向字段值的指针，并利用指针来访问或修改结构体内容。因此，在定义记录体时需要同时创建 StructDef 对象，并合理的设置后存入 TypeTable，供后续翻译工作时查看。

4.4 IRGeneratorContext

这个类是实现 LLVM-IR 生成的核心类，它保存了一个 LLVM 中间码生成所用到的全部上下文，并提供一个方法 GenerateIR，当调用此方法，传入抽象语法树后，该类就会遍历抽象语法树，在此过程之中构造 LLVM-IR。

生成 LLVM-IR 的过程中主要是用到了 llvm::IRBuilder 类与 LLVM 类型系统。利用 IRBuilder 可以产生指定的指令，如一元、二元四则运算，条件跳转，存取指令，偏移量计算指令等等，与三元表达式非常类似。翻译的过程中需要根据不同的语句选择不同的指令，为此，plang 把抽象语法树分为不同的机组类型，针对不同的组调用不同的方法来生成对应的 LLVM 指令。

针对不同的语句翻译方式各不一样，下面将介绍几种主要语句的翻译方法，详情仍然是需要参见源码，因为在此报告中介绍全部的翻译模式将要耗费太多的篇幅。

4.4.1 表达式翻译

多数 AST 节点的翻译的结果为一个 LLVM 值对象，它表示一个运算结果，通常在 plang 执行时并不确定（常数例外）。利用值对象与值对象通过运算符组合，就可以得到新的值对象，这就是运算符的翻译模式。在 LLVM 中，使用 IRBuilder 的 CreateOP 系列函数可以生成此类指令。

在翻译表达式的过程中会遇到读取变量的情况。在 LLVM 中，变量是保存在堆或栈中的一片内存空间，在分配空间时获取了指向空间的指针，存入 NameTable 之中，当需要读

取一个变量时，从 NameTable 中查找指针，并使用 LLVM 的 load 指令获取变量的值。同理，在给变量赋值时，也需要通过某种手段获取变量的指针，使用 LLVM 的 store 指令把值赋予变量。

另一类较为复杂的表达式有还有数组索引与记录体索引。在 plang 中，数组是堆中连续的一片空间，数组变量的值为指向此空间头部的指针。在已知数组大小的情况下，可以很轻易的利用偏移量，算出数组所访问元素的地址。最后利用计算到的元素地址就可以进行存取操作。记录体的索引与数组类似，可以利用字段名称从 TypeTable 中查询到字段的索引，并利用索引号计算偏移量，最终得到可以访问字段值的指针。

最后还有一类表达式，这类表达式用于初始化数组或是记录体。这类表达式的翻译最为复杂。对于结构体的初始化，需要在内存中为记录分配一段空间，然后根据每个字段的初值表达式来计算初始值对象，并计算出字段指针，进行赋值，最终把字段中的值赋予变量。而初始化数组更加困难，这是由于数组中赋值的长度是在运行时确定的。对于一组数组初值语句，plang 首先迭代每一个子句，计算全部表达式的值，并把值相加，针对每一个子句循环指定次数（这也是由次数表达式的值决定的），为数组赋予初值。

表达式的翻译代码位于 IRGenContext::GenerateExpression(ast*)。

4.4.2 语句的翻译

PCAT 的语句包含许多种类型，如赋值、条件、循环、返回、输入输出等。针对每一种不同的语句，需要产生不同的 LLVM 指令。

赋值语句已经在表达式的翻译中提及过，即采用 LLVM 的 store 指令对一个指针进行存储操作。对于后续的语句翻译，需要使用到 LLVM 的基本块概念（llvm::BasicBlock）。一个基本快是指有唯一入口与唯一出口的一段中间码段，在同一函数之中包含一个或多个基本块，基本块之间需要用跳转指令连接。在同一函数之内，允许从任何位置跳转至一个基本块的头，而达到基本块的尾部后，就必须跳转至其他基本块。如此就可以保证一个程序中不会出现冗余的代码段。

条件语句、循环语句的翻译都利用了基本块来处理逻辑流。对于条件语句，每一段内部语句块都隶属于一个基本块，每一个条件判断页数以一个基本块。可以按照如表 4.1 所示实现条件语句的翻译。

```
if.cond1:
   %cond = <condition value>
   br cond %cond, if.true1, if.else1
if.true1:
   <true statements>
```

```
        br if.exit
    if.else1:
        %cond1 = <condition value1>
        br cond %cond, if.true2, if.else2
    if.true2:
        <true statements>
        br if.ext
    if.else2:
        ....
    if.exit:
```

表 4.1 条件语句的翻译

循环语句的翻译与条件语句类似，但也许会更加复杂一些。其中 FOR 循环需要额外注意。按照 PCAT 官方参考文档的定义，其起始值应不大于终止值，同时循环计数器应取初始值至终止值的左闭又开区间。但在本实验的测试数据中，这两点均不成立，因此针对 FOR 循环的翻译过程中，首先需要对起始值与终止值的大小进行判断，根据其大小来分别计算循环计数器是否超出循环范围。

与解释执行的编译器不同，plang 实现的是本机指令的生成，因此不可能直接向语言中注册 C 函数供交互操作调用。为了实现输入输出语句，plang 对 C 语言的 scanf 与 printf 进行了封装，实现了__pcat_readreal()、__pcat_writereal()、__pcat_writeint()、__pcat_writestr() 等函数。输入输出语句实质上是对这类函数的调用，来实现与标准输入输出的数据交换。这四个函数实现的代码位于 IRGenContext::CreateInternal()系列函数中。

4.4.3 过程翻译

PCAT 中的过程（PROCEDURE）与 C 语言中的函数类似，定义了一个可以供外部语句调用的程序，它可以接受一组参数，并可选地返回一个值。过程与 LLVM 中的函数对象（llvm::Function）对应，为了创建函数体，需要列举输入参数表与返回参数类型，利用这些参数构造 LLVM 函数原型（llvm::FunctionType），然后再根据函数原型创建函数对象，并插入到主模块之中。

定义了一个函数后，就需要为函数创建一个上下文，即向 NameTableStack 中压入一个新的 NameTable，并根据参数与声明列表来向 NameTable 中添加变量声明。然而在压入新的 NameTable 之前，需要先把函数注册到当前的符号表之中，以便此后可以在作用域中对过程进行调用。

函数创建后，IR 的插入位置也改变了。因此需要为函数创建一个基本块，并把插入点设为新的基本块之上。此后即可继续进行变量声明等翻译工作。完成了声明部分与语句体部

分，即函数结束后，弹出当前上下文，然后回到上一层上下文之中，恢复 IR 的插入位置到此前的基本块之上。至此，过程的翻译基本结束。

然后在此之中还需要处理一些特别的问题。例如，基本块要求只有一个入口与一个出口（条件双跳转也算为一个出口），然而在翻译的过程中不可避免，也许会在一个基本快中插入多个出口。为此，在进行过程翻译与语句翻译时，需要时刻记录当前基本块是否已经产生了出口。若产生了出口，那么此后生成的任何语句都是无意义的，只需要丢弃，并向用户发出警告即可。

4.4.4 声明的翻译

变量的声明基本在前文已经有所记叙。在 LLVM-IR 中，变量保存于堆或栈之上，在分配空间时会得到一个指向分配空间的指针，在 plang 中就是用指针来维护变量的存取访问。因此，每声明一个变量，都会在 NameTable 中保存它的名字与指针地址（同样也为一个值对象）。当访问变量时，通过指针的 load 指令来获取值，赋值时，通过指针的 store 指令来存储值。

详细的翻译模式，见 IRGenContext::GenerateVarDeclares(ast*, bool)。

5 总结

本报告至此已经完成了对 PCAT 编译器 plang 的介绍。PCAT 是 Pascal 的简化版语言，本实验中基于 LLVM 实现了一套完成的 PCAT 编译器，其中涉及了词法分析、语法分析、抽象语法树生成、语义分析、中间代码生成、汇编与链接等技术，最终生成二进制可执行文件。这与解释执行相比有着两点优势：性能极佳，并且二进制文件的生成允许 PCAT 产生或利用库程序，提供了很强的复用性。最终经过测试，plang 是非常稳定并高效的编译器。

•