编译实习实验班
项目报告

高嗣昂 1400012840

2017-1-14

1 总述

本项目名称为 *Cwft*，是一个语言转换器，可以将 Swift 语言转换为等价的 C++ 语言，确保转换后的文件可以直接编译并运行。

这个转换器可以识别并转换大部分 Swift 中的常用语句和声明，如引用、类、结构体、枚举、函数、switch、(repeat) while、if (else)、typealias、for in 语句等。同时，此转换器还有一些特性，如没有借助 C++11 中引入的 auto 关键字和 for-each 语句、可以进行类型推断、可以进行简单的错误查询等。是一个简单易用，同时又十分强大的语言转换器。

报告剩余部分将从不同方面对此项目进行说明。第二节介绍了项目中用到的语言和工具；第三节介绍项目的整体架构；第四、五节为本报告重点，介绍了转换器的具体实现、特色与难点问题的解决；第六节介绍了如何在机器上安装、运行此转换器；第七节分析了以若干典型 Swift 文件作为输入得到的输出结果；第八节探讨了此转换器未来的改进方向；第九节是对此项目作业的感想与总结；第十节列出了参考文献。

2 工具

本项目总共使用了四种语言或工具：Antlr、Swift、C++、Java。下面将分别进行简要介绍。

Antlr 其全称为"ANother Tool for Language Recognition"，是基于 LL(*) 算法实现的语法解析器生成器。使用 Java 语言编写。作者为 Terence Parr。本项目主要使用 Antlr 生成的 Lexer 和 Parser 来将 Swift 源文件转化为抽象语法树，并生成遍历这棵树所需要的 Visitor 模式的基类。

Swift 是苹果公司开发的一门编程语言，有许多现代语言的特性。Swift 语言还很年轻，在不断地迭代升级中，不同版本的 Swift 所使用的语法可能略微不同。本项目采用的是 Swift 3.0.1 版本的语法作为源编程语言进行输入。

C++ 是我们都熟悉的一门编程语言。本转换器生成的 C++ 目标代码大都不需要 C++11 版本及以后的特性，只有在数组声明的时候所采用的语法在 C++11 以后才有。所以在编译目标文件时要使用 –std=c++11 选项。

Java　用来实现 Visitor 访问抽象语法树时的所需动作。因为 Antlr 本身为 Java 语言编写，所以其生成的访问者接口也需用 Java 语言实现。

3　架构

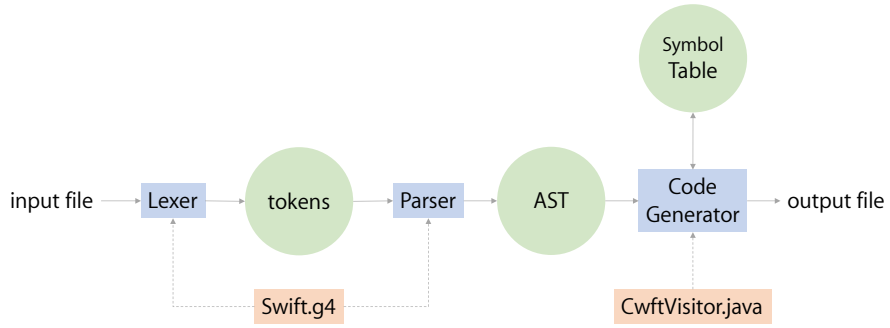

图 1: Architecture

如图 1 所示，本翻译器总体上采用两个步骤来实现功能。第一步是**识别 (recognition)**，第二步是**翻译 (translation)**。识别阶段，是将输入源文件转化为抽象语法树的过程；翻译阶段，是访问这棵抽象语法树并生成目标代码的过程。下面，我们将结合图 1 大致讨论一下工作流程。

识别　在识别阶段，我们首先输入一个源文件，源文件经过一个词法分析器变成字符流，再通过句法分析器生成中间形态的抽象语法树。其中语法分析器与句法分析器均是通过 Swift.g4 这个文件生成的，这是一个使用 Antlr 语法写成的文件，其中包含了解析 Swift 文件所使用的词法与句法。Antlr 会自动根据这个文件生成对应的词法分析器与句法分析器。

翻译　在翻译阶段，我们输入识别阶段得到的抽象语法树，通过一个访问者遍历这棵树。在遍历树的过程中，我们会得到符号表和 Swift 语句对应的代码。若没有发现错误，访问者便会把生成的代码输出；若发现错误，则输出错误信息。这个访问者是由 CwftVisitor.java 这个文件实现的。其主要功能，就是决定在访问一个节点的时候该做些什么事情。

4 具体实现

这一节将讲述翻译器的具体实现方式，即 Swift.g4 和 CwftVisitor.java 两个文件编写的大致思路。首先是识别部分，然后是翻译部分。因为篇幅的限制，Swift 语法部分的完整版内容请见附件。

4.1 识别

识别部分，总体来说就是 Swift 语言的词法和文法构成。因此以下篇幅大部分是讲 Swift 的语言本身，而实现部分内容相对较少。因为一旦获取了正确的词法与文法，借助 Antlr 工具便能很方便地生成词法分析器与语法分析器。

4.1.1 词法

像其他语言一样，Swift 的词法部分也相对较简单。主要分为如下几类：

注释 像 C 语言中一样，注释是由双斜线开始、到回车结束的字符串，在翻译过程中应忽略掉。

关键词（保留字） 是那些有着特殊定义、不能被用来作为其他用途的词，如 **inout**、**class**、**enum** 等。

操作符 即运算符，如 →、=、+= 等。

分隔符 标点符号和括号，如 **，** **：** **{}** 等。

字面量 包括四种基本类型：整数、浮点数、字符串、布尔型，也是作为类型推导的重要部分。如 **123**、**12.3**、**true**、**“123”** 等。

标识符 可作为变量名、类名等。如 **abc**、**a__b__c__** 等。

空格 为了代码格式好看，应该被忽略的字符，包括空格、回车、换行符、制表符。

4.1.2 词法部分实现

根据每一类的词法特征，在 Swift.g4 中编写相应的正则表达式即可。首先，Antlr 要求一个词法单元要以大写字母开头；其次，例如像 “,” 这样的符号也要为其起名 (比如 Comma)，避免在按照单词进行分割的时候出错；最后，要按正确的次序进行排列，如 **else** 既可以被识别为一个关键词，也可以被识别为一个标识符，这时候就要保证关键词的正则表达式排在标识符之前。

例如下面这个例子，生成了浮点数的词法识别模式：

```
1 Floating_point_literal :
2         Decimal_literal Fraction? Exponent?;
3 fragment Fraction : '.' Decimal_literal;
```

上述模式表明，一个浮点数由三部分组成 (后两部分可有可无): 十进制数串、小数部分、指数部分；其中小数部分是一个点加十进制串。关键词 fragment 的意思是在生成解析树的时候生成到浮点数部分就可以了，不用再生成小数子结点。

4.1.3 句法

一个 Swift 文件是由零句、一句或多句话组成的，每句话都是一个**陈述 (statement)**。一个陈述语句可以分为**表达式 (expression)**、**声明 (declaration)**、**循环 (loop statement)**、**分支 (branch statement)**、**控制转移 (control transfer statement)** 五种。下面分别进行简述。

表达式 计算一个表达式会返回数值或产生副作用，是一门语言中最重要的部分。常见的表达式有算术运算、函数调用、访问数组、访问类的成员函数等等。表达式可大致分为前缀表达式、双目运算表达式、原始表达式和后缀表达式四种。

声明 在程序中引入新的变量或常量，定义一个新的类型，定义函数、枚举类型、结构题和类。声明语句通常会导致符号表中内容的变化，也是类型推导的重要依据。

循环 循环语句，包括 for in 和 while。与 C++ 中的 for 形式的循环不同，Swift 中的 for in 形式是对一个可排列的数据结构进行遍历，显得更加抽象。而 C++ 中的 for 更贴近于 Swift 中的 while 语句。

分支 包括 if 与 switch 两种。Swift 中的 switch 语句默认是在每个 case 后面 break 出来的，只有在真正需要 fall through 的时候才显示地指明，这一点和 C++ 正好相反。

控制转移 包括 **break**、**continue**、**fallthrough**、返回语句四种。

4.1.4 句法实现

Antlr v3 及以前的版本是不支持左递归的文法的，这使得语法书写起来很不方便，从 Antlr v4 开始，通过支持 Kleene Closure 表示法实现了对左递归的支持。总体来说，用上下文无关语法来实现对 Swift 语言的识别相对容易。只要正确地分析了每个文法的组成部分有哪些，文法本身没有错误，Antlr 便能自动生成语法分析器。需要注意的一点是，有的非终结符号具有多个生成式（也即中间用 "|" 进行了连接），那么为了方便下一个阶段的翻译过程，需要给每个生成式再分别取一个名字。

下面的代码是为 type 的识别生成的文法模式：

```
type
: Left_mid type Right_mid              # array_type
| Left_mid type Colon type Right_mid   # dictionary_type
| type_identifier                      # basic_type
;
```

我们可以看到，type 有三种不同的类型：基本类型、数组类型和字典类型。每种有不同的生成式，在井号后面是此类生成式的名称。

4.2 翻译

我们可以将翻译过程的流程再进行细化。如图 2 所示，整个翻译过程的输入是一棵抽象语法树，而输出的是可以立即编译、执行的目标代码。我们通过一个访问者去遍历这棵语法树，在遍历的过程中会产生三张符号表和相应的目标代码。目标代码分为两部分，一部分是所有的声明语句，要写在

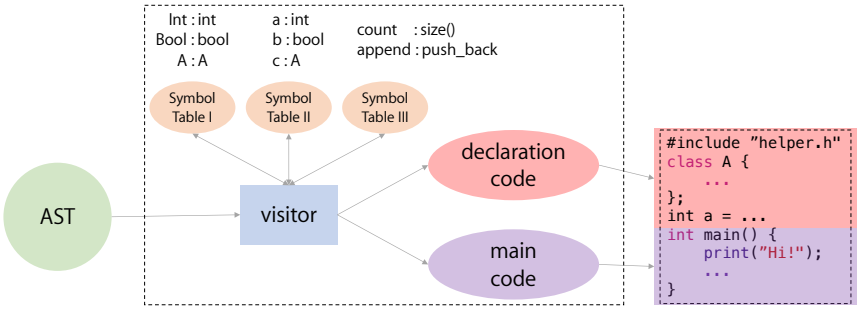

图 2: Translation

C++ 文件中 main 函数之外的地方；而其余所有代码写在 main 函数之中，负责执行。

4.2.1 符号表

符号表 I 负责存储一个类型在 Swift 中的名称和 C++ 中的名称的对应关系。例如整型变量在 Swift 中表示为 Int，而在 C++ 中则表示为 int。在翻译成目标代码的过程中需要换名。此外，新声明的类名等，在 Swift 中和 C++ 中的名称则是一样的。符号表 I 的使用，使我们不必在词法分析时将 Swift 的基本变量名强制性地写进去，而是在词法分析阶段将其识别为一个标识符，而在翻译阶段，通过查找符号表来确定其到底是一个类型名称还是一个变量。使得语法转换器的扩展性很好。

符号表 II 负责存储一个变量对应的 C++ 类型名。譬如 a 为 int，b 为数组等。在类型推导的第一步便是通过这个符号表去确定某个变量的类型。

符号表 III 负责解决在 Swift 和 C++ 中相同功能的成员函数名称不同的问题。比如 Swift 中，向数组中添加一个元素使用 append，而 C++ 使用 push_back。这个时候就需要替换名称。

4.2.2 两种访问模式

Antlr 提供了两种访问抽象语法树的模式和接口，一种是访问者模式，一种是监听者模式。在监听者模式中，Antlr 自动地访问这棵抽象语法树，并且在每进入和离开一个节点（非终结符号）时发送信号。我们可以实现相

应的方法来一步步地完成翻译工作；与之对应的是访问者模式，Antlr 将控制权交给编写程序的人，由编程者决定是否访问、以何种顺序访问一个结点以及其子节点。我们选择了访问者模式，因为它更加自由，并且还有另外一个好处：访问者模式的实现方法是可以有返回值的，通过将生成的代码放在返回值中，避免了显示地维护一个全局的代码区存储代码。下图是一个访问 Prefix_Expression 的函数接口：

```
@Override
public Record visitPrefix_expression(
        SwiftParser.Prefix_expressionContext ctx) {
}
```

4.2.3 实现细节

下面，我们将由简入繁地描述转换细节。

首先，是一些简单的情况下的转换。C++ 和 Swift 中字面量的表示方式，包括整数、浮点数、布尔、字符串，是一样的，因此我们不需要做任何修改。直接将 Swift 中的字面量原封不动搬到 C++ 中即可；Swift 中的条件判断是没有括号的，在 C++ 中，需要在其周围加上括号；Swift 语句中每句话后面的分号是可选的，而 C++ 中强制要求这一点，因此，我们要在每句话的后面添加分号。具体添加分号的时机如下图所示，在访问一个叙述语句的时候，若是一个表达式或控制转移语句，便要在最后加上分号，否则不需要。

```
@Override
public Record visitStatement(
        SwiftParser.StatementContext ctx) {
    // some code...
    if (ctx.expression() != null ||
        ctx.control_transfer_statement() != null) {
        ret.code.add(rec.code.get(0) + ";");
    } else {
        // some code...
    }
}
```

其次，是一些稍微复杂的情况。在声明一个数组或字典时，Swift 的语法是将每个元素中间用逗号隔开，若是字典则键值对中间用冒号隔开，并且首尾需要加上中括号；而在 C++11 以前，并没有这种便捷的初始化数组或字典的方式，在 C++11 中添加了格式略微不同的初始化方式。外部括号是大括号而不是中括号，并且对于字典来说，我们需要将每个键值对用大括号包裹起来。

还有就是在声明的时候，Swift 语言允许一个声明语句中声明多个变量，并且这些变量的类型不同，但在 C++ 中一行的变量肯定是相同类型的 (这里的一行中不含有分号)。因此，我们需要拆分 Swift 一行中的声明，将其中每一项都变成一行独立的声明语句。

对于函数声明，首先，我们需要将函数的返类型，也就是右箭头后面的类型，移动到最前面以符合 C++ 的语法，然后将参数列表中的参数名称和类型换位置。并且，如果函数没有返回值，我们要在最前面生成一个 void。函数声明转换框架如下：

```
@Override
public Record visitFunction_declaration(
        SwiftParser.Function_declarationContext ctx) {
    // some code...
    if (func_type.wrap != 3) {
        ret.code.add(func_type + " " +
                        func_id + func_param);
    } else {
        ret.code.add("void " + func_id +
                        " " + func_param);
    }
    // some code...
}
```

在翻译 switch 语句时，我们需要检查一个 case 中最后一句话是否为 fallthrough，若是，若不是，则需要在翻译的 C++ 代码中对应位置填上 break。

最后是需要较大改动的情况。在翻译 for in 语句的时候，我们需要分成两种情况。如果是像 1..<4 这样在一个范围内递增，那么便相对容易地转换为 C++ 代码；而如果是对一个数组进行迭代就相对复杂。如果我们

采用 C++11 以后带来的 for 语句，即采用 **for(:)** 的形式来实现，那么情况也会比较简单；不过，为了验证类型推断的效果，我们还是采用原始的 for 语句来进行实现。因此，首先我们需要使用一个迭代器来进行遍历，并在循环内加上一句话，即把迭代器指向的值赋给 Swift 中的变量。以 Swift 中的 **for a in b** 为例，首先，我们需要在符号表中查找到 b 所对应的类型，此处假设为 **vector<int>**，那么我们便要创建一个类型为 **int** 的迭代器 (iterator<int>)，在 b 上进行迭代，然后，在循环内部插入代码 **int b = *it**。

5 特色与难点

5.1 返回结构体

得益于访问者模式的优势，我们可以在实现函数接口的同时添加返回值。在翻译过程中，返回值保存了本结点及其子结点所生成的代码，以及对应的类型。代码的保存使用的是一个 **List<String>** 类型的变量，以行为单位进行存储；而类型存储使用的是一个类型为 **Type** 的变量，下面会详细叙述。通过这种方式，每访问一个结点，其返回值便会保存生成的代码以及对应类型。我们需要将返回的代码综合到父节点的代码中，并使用类型进行类型推导。除此之外，每个返回值中还保存了一个布尔值，名为 **isType**，其作用是告知刚刚访问的结点是否为一个类型。如下面的例子：

```
var a = [6]
var b = [Int]()
```

应该生成如下的 C++ 语言代码：

```
vector<int> a = {6};
vector<int> b = vector<int>();
```

我们可以看到，第一个中括号包裹着 6 代表着一个数据，第二个中括号包裹着 Int 则代表是个 int 数组。这时候，我们通过设置 isType 位，在访问 int 结点时 isType 位设置为 true，在访问 [int] 时因为数组中的元素 isType 为 true，加上中括号后 isType 依然为 true，是一个 int 数组。由此生成相应的正确代码。

5.2 类型

我们定义了一个 Type 类型，用来正确地表示一个数据类型。其类定义如下：

```
class Type {
    String basic;
    Type type1;
    Type type2;
    int wrap;
    public String toString();
}
```

basic 是个字符串，如果不是数组或词典类型，那么 basic 字符串就会保存了这个类型的名称；type1 与 type2 则是为了数组与字典服务的，如果一个类型是数组，那么 type1 便是数组中每个元素的类型，如果是一个字典类型，那么 type1 是 key 的类型，而 type2 是 value 的类型；wrap 为 0 代表是基本类型或类、结构体，为 1 代表是数组，为 2 代表是字典，为 3 代表未决类型，为 4 代表是枚举类型。

toString 函数则是返回类型的字符串表示。若是一个基本类型，那么直接返回 basic 值即可；若是一个嵌套类型，如数组、字典，那么需要递归地调用 type1 和 type2 的 toString 函数，再进行适当地拼接，加上如 vector 等关键字，最终返回字符串表示。

5.3 类型推导

自动类型推导是 Swift 语言的特色之一，在声明变量的时候不需要显示地指出变量类型是什么，而是编译器通过字面量的归类，以及表达式的类型推导等方式自动地得出结果的类型。而 C++ 语言在 C++11 之前是不支持这一点的，在 C++11 之后加入了 auto 关键字后也能做到。本翻译器在不使用 auto 关键字的前提下，实现了类型推导。

实现类型推导需要两步，首先要在恰当的时候将变量和所对应的类型加入到符号表之中，其次还要在需要的时候进行查询。加入符号表的过程包括：翻译程序初始化时 (加入内置函数的类型，如 count 对应 int)、typealias(新定义的 type 加入符号表 I)、enum 声明 (将枚举类型和每个枚举量都加入符号表)、结构体和类的声明、变量和常量声明、函数声明 (包括函数名和参

数) 等。而查询的过程包括访问函数调用、变量访问、数组访问等。例如下面这个简单示例：

```
var array = [1,2,3,4,5,6]
var item = array[1]
for i in array {

}
```

在访问第一行等号右半部分时，通过字面量归类以及数组包装得知返回的类型为 int 数组，将 array-vector<int> 这个键值对加入到符号表当中并输出相应代码；在第二行右半部分，先从符号表中查找到 array 的类型为 vector<int>，因为是数组访问，取 array 类型的 type1 也即 int 作为 item 的类型。同样地，在对 array 进行遍历时，首先推断出生成的迭代器指向 int 类型的数据，然后在循环内生成的 i 也为 int 类型。

6 实验条件及流程

6.1 环境配置

Swift 3.0.1 为了能够写出没有错误的 Swift 源代码并运行得到预期的结果，需要能够编译运行 Swift 语言。若是 macOS 系统，下载安装 XCode 即可；ubuntu 系统没有尝试，不过网络上似乎有安装 Swift 的教程。Windows 系统暂不能使用 Swift。

GCC 为了能够编译运行得到的 C++ 源代码，需要 C++ 等编译工具并支持 C++11 标准。

Java Antlr 工具和访问者模式的编写均是基于 Java 实现的，因此机器必须配置了可以运行 Java 的环境。

Antlr 此工具既有工具链的形式也有 Java 库的形式，这里为了自动生成目标代码，使用的方法是在 Java 代码中 import 这个 Java 库。具体安装方法请见官网。

6.2 运行流程

首先，要使用 Antlr 工具，通过 Swift.g4 文件（其中包含了词法结构和语法结构）生成词法分析器和语法分析器，其中-visitor 是令其生成访问者模式的接口。

```
$ java -jar /usr/local/lib/antlr-4.5.3-complete.jar
        -no-listener -visitor Swift.g4
```

其次，编译所有的 java 文件，其中 Master.java 是用来控制整个程序行为的文件。

```
$ javac Master.java Swift*.java
```

然后，使用 test.swift 源程序作为 Master 输出，产生输出，输入到 res.cpp 中。

```
$ java Master test.swift > res.cpp
```

最后，编译运行结果代码，注意编译选项要加上-std=c++11。

```
$ g++ -std=c++11 res.cpp
$ ./a.out
```

7 实验结果

为了测试翻译器结果的正确性，我们编写了若干测试文件，现选取其中重点部分进行说明，完整生成结果请见附件。

首先，翻译器可正确地将用 Swift 文件写的快速排序代码翻译成 C++ 语言。如 Swift 中的这个段落：

```
let pivot = array[0]
for x in array {
```

会被翻译成：

```
const int pivot = array[0];
for (vector<int>::iterator it = array.begin();
        it != array.end(); ++it)
```

```
{
        int x = *it;
```

可见其正确地处理了类型推导，以及 for in 语句的翻译问题。

class 和结构体也可以正确翻译。如：

```
class A {
    var a: Int = 5
    var b: Double = 6
    func f(d:Int)->Int {
        return a+b
    }
}
var c = A()
print(c.f(d:2))
```

会被翻译成：

```
#include "helper.cpp"
class A
{
public:
        int a = 5;
        double b = 6;
        int f(int d)
        {
                return a + b;
        }
};
A c = A();
int main() {
        print(c.f(2));
        return 0;
}
```

此外还有枚举类型。对枚举类型的声明平淡无奇，但是在使用 switch 语句进行分类的时候，情况略有不同：

```
enum A {
    case a, b, c
}
var cc = A.a
switch cc {
case .a:
    print("haha")
case A.b:
    print("hehe")
default:
    print("hoho")
}
```

其转换结果为：

```
#include "helper.cpp"
enum A
{
    a,b,c,
};
A cc = a;
int main() {
    switch (cc)
    {
        case a:
            print("haha");
            break;
        case b:
            print("hehe");
            break;
        default:
```

```
17              print("hoho");
18      }
19      return 0;
20  }
```

注意到在 switch 语句中进行分类讨论时，Swift 中的.a 和 A.b 均被替换成了 a，并且在所有没有 fallthrough 的地方均加上了 break。

此外，此翻译器还会忽略所有的注释，正确处理引用情况，并在多重的 for in 中得到正确结果。翻译器还会处理一些简单的错误情况，如：

```
1  var a = 1
2  var a = "a string"
```

此时便不会输出任何翻译结果，而是 “Error: a already defined!” 的错误信息。

8 未来展望

范型 在 Swift 中，声明一个范型的方法很容易，就是在声明的变量名后面加上中括号，里面是范型名称：

```
1  func swapTwoValues<T>(_ a: inout T, _ b: inout T)
```

在 C++ 中声明范型类似，只是需要在这句话前加上 template 的声明。因此转换的实现应该不会很难。

optional optional 是 Swift 中一个比较特殊的类型，它可以与一个其他类型 T 绑定，属于这个类型的变量要么类型是 T，要么类型是空。在 C++ 中或许可以通过 enum 的方式来实现。

tuple 相当于一个动态的结构体。一个 tuple 就是由其他类型所组成的类型。例如，在函数返回值的时候，我们希望返回的是一个点，其有横坐标和纵坐标。那么我们只需要将两个值用小括号括起来直接返回即可。在 C++ 中或许可以通过结构体的方式进行实现。

9 总结与思考

在制作翻译器的过程中，我遇到了一些困难，也学习到了很多的新知识，在这里总结如下：

尽早开始 一个学期的课程很多，在课程项目布置后应该尽早开始规划、尽早动手。随时都会有新的作业、新的任务布置下来。尽早开始，可以确保我们有足够多的富余时间来处理各种各样的突发情况，也不会在 deadline 来临之际才开始紧赶慢赶。

由简入难 编写一个转换器是一个复杂的工作。比如第一步，编写词法、语法分析器，Antlr 有整整一本书来讲述过程，将这本书全部看完、每一个小细节全部掌握才开始编程是不可能的，也会很快浇灭我们的热情。因此我们应该从最简单的事例入手，再逐渐向代码中添加困难的部分。譬如可以先完成声明部分的语法识别和翻译，再去看其余部分。

程序先跑起来再说 作为有追求的程序员，我们都希望程序写的又快又好，然而这是很难做到的，并且——说难听点——是“没用”的。真正重要的让程序先跑起来，这是从零到一的转变，后面的优化过程反而没有那么重要。过于纠结于代码的效率和整洁性，会严重拖慢开发进度，甚至可能导致项目的夭折。

语言很相似 现代语言，其实基本的组成部分是很类似的。譬如表达式、循环、分支等，有时几乎都不需要什么转换直接复制过去都可以。所以，醉心于学习一门又一门的新语言，掌握那些语法糖，其实没有很大的作用，相反，专心地钻研一门语言，掌握其特性，那么再上手其他语言是十分轻松的事情。

很难一次成功 在真实的程序开发环境中，一个应用不是一口气写完的，肯定是在解决了很多 bug，打了很多补丁后才“看上去没问题”，因此我们不能指望写程序的过程会一帆风顺，总会有新的 bug 在等着我们。

参考文献

[1] *The Definitive ANTLR 4 Reference*, Terence Parr, 2nd edition, 2013.

[2] *The Swift Programming Language (Swift 3.0.1)*, Apple Inc..

[3] *Let's Build a Compiler*, Jack Crenshaw.