**设计和实现：**

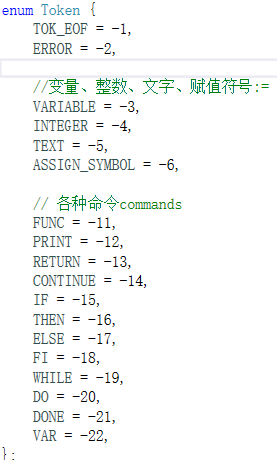
**1.设计实现词法分析，并输出词法信息，至少包括（单词，term类型）**

**1.1词法分析的设计和实现：**

1. 首先分析VSL，找到其中有意义的字符

包括变量名（VARIABLE）、整数（INTEGER）、文字（TEXT）、赋值符号（ASSIGN\_SYMBOL）和各种需要使用的命令，各种命令如下：

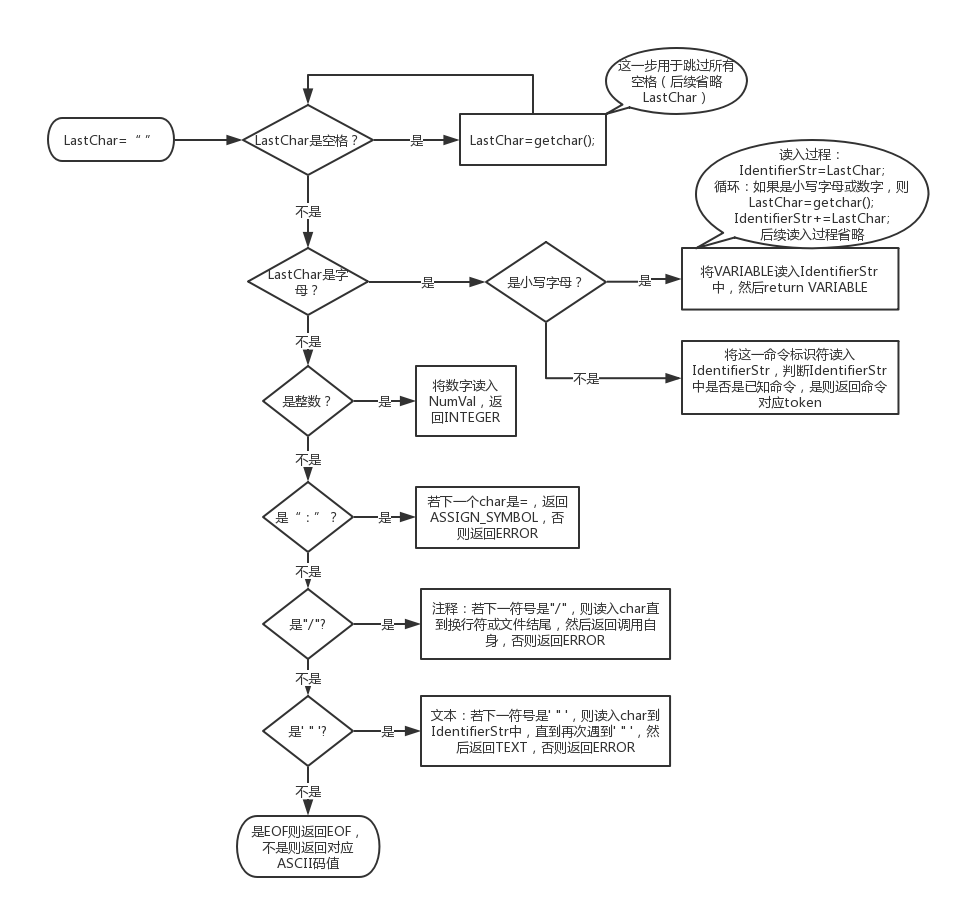
FUNC, PRINT, RETURN, CONTINUE, IF, THEN, ELSE, FI, WHILE, DO, DONE, VAR.

1. 设置每个符号对应的枚举类型token（对应VSL中的所有可能出现的有意义的字符） 
2. 创建两个全局变量，分别用于存储读入的标识符名称和整数值



1. 创建gettok()函数，逐字符读入代码，并进行识别，如果读取的是上述已知字符（或字符串），则返回对应的token数值，如果是未知字符则返回该字符对应的ASCII码值，gettok()的实现思路如下：

注意：VSL规定变量标识符由小写字母和数字组成，且由小写字母开头，所有的命令都是大写字母组成，可以用于区分



1. 调用gettok()即可获得对应字符或字符串的token，词法分析完成。

**1.2输出词法信息**

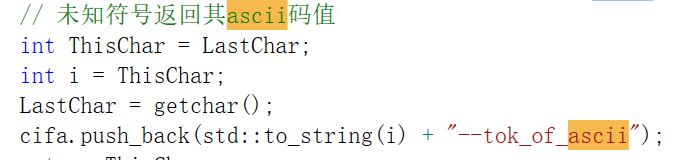
1. 创建一个全局字符串数组用于存储待输出的词法信息



1. 在识别字符时，如果识别到的是已知字符，将字符-字符类型存入字符串数组中；



如果不是已知字符，将字符的ascii码存入字符串数组中

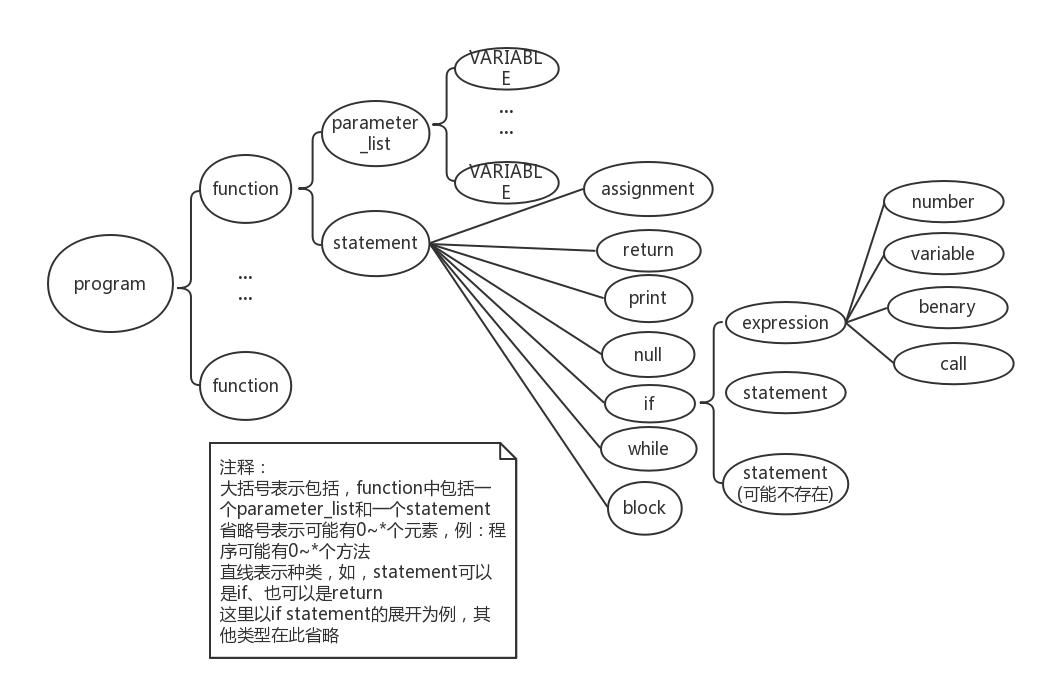


1. 最后在生成目标文件时，将字符串数组中的内容输出到lexer.txt文件中

**2.设计实现语法分析，并输出语法分析树**

**2.1设计实现语法分析**

1. **分析VSL语法结构**



1. **设计抽象语法分析树AST**

LLVM已经给出了表达式的抽象语法树：包括基类ExprAST和继承该基类的子类NumberExprAST、VariableExprAST、BinaryExprAST、CallExprAST。

还有function相关的抽象语法树FunctionAST、PrototypeAST。

可以总结出，抽象语法树都应该包含如下内容：

**a.它应存储的内容**，可以是字符串、数字或数组，也可以是其他自定义类,如FunctionAST中包含了ExprAST类的指针；（这些内容以私有属性的方式存在类中）

**b.一个公有的构造函数**

**c.一个公有中间代码生成函数（后需要用到）**

因此，根据上图和VSL的语法要求以及LLVM提供的帮助，设计抽象语法树AST。

首先是，创建statement的抽象语法树的基类StatAST，让它继承ExprAST（因为FunctionAST中包含的是一个ExprAST类的body对象，而且后面遇到了点bug，继承ExprAST后可以比较好的解决）

然后创建StatAST的一系列子类：

**NullStatAST：**不需要存储内容

**ReturnStatAST：**存储1个ExprAST指针（std::unique\_ptr<ExprAST>，下同）

**AssignStatAST：**存储1个ExprAST指针，和1个字符串（指定被赋值的对象）

**IfStatAST：**存储3个ExprAST指针（分别代表条件表达式、then\_statement、else\_statement），两个不同的构造函数，分别对应有无else的情况;

**WhileStatAST：**存储2个ExprAST指针（分别代表条件表达式、while\_statement）

**BlockAST：**存储1个StatAST类指针的数组（std::vector<std::unique\_ptr<StatAST>>）

和1个DeclExprAST类（后面详述）指针的数组（std::vector<std::unique\_ptr<DeclExprAST>>），分别用来存储一系列declaration和一系列statement。

**PrintStatAST：**存储一个PrintItemAST类指针的数组（std::vector<std::unique\_ptr<PrintItemAST>> Items）

实现了所有的statement类型的AST后，我们发现还有declaration和printitem两类抽象语法树需要用到。

declaration是一系列变量的声明：

**DeclExprAST：**包含1个 字符串和ExprAST指针组成的对儿的数组（std::vector<std::pair<std::string, std::unique\_ptr<ExprAST>>>）， 继承StatAST;

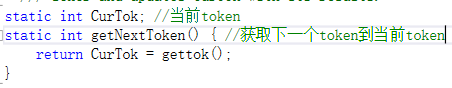
printitem则分为两种，分别是表达式类型打印项ExpPrintItem和文本类型打印项TextPrintItem；所以构造基类PrintItemAST，和继承他的两个子类ExpPrintItemAST、TextPrintItemAST

**ExpPrintItemAST：**存储1个ExprAST指针

**TextPrintItemAST：**存储1个字符串Text

1. **实现语法分析器Parser**

首先，定义全局变量CurTok和函数getNextToken()，用于存储和获取token



创建2个用于打印错误信息的函数以及1个获取二元运算符优先级问题：

std::unique\_ptr<ExprAST> LogError(const char \*Str)

std::unique\_ptr<PrototypeAST> LogErrorP(const char \*Str)

static std::map<char, int> BinopPrecedence;

LLVM已经给出了分析表达式Expression和分析Function的一系列函数，这里不再一一列举，我们只需简单修改就能使用，同时可以从中分析其实现思路

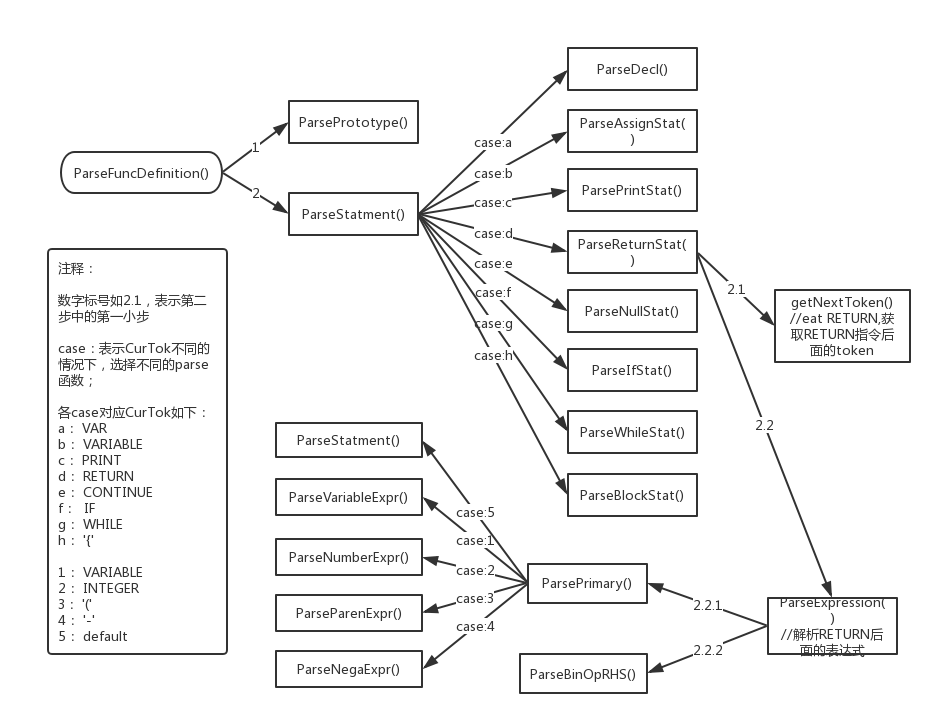
具体思路如下：

* 利用getNextToken()函数来获取下一个CurTok，用来跳过不需要存储的命令标识符，注意判断命令是否正确
* 根据VSL语法结构，生成当前AST需要存储的内容，可以通过获取全局变量：IdentifierStr、NumVal中的值，也可以调用不同的Parse函数，

如VSL中：return statement的语法结构：RETURN expression跳过RETURN后可以调用ParseExpression()函数获取ExprAST

* 根据前面的结果，生成对应AST并返回。

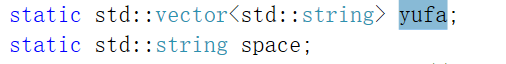
下图展示了整个语法分析器的调用结构（以ParseReturnStat()的具体实现为例）。



根据实现思路和上图结构，实现图中的所有parse函数，就实现了语法分析器Parser。

**2.2输出语法分析树**

1. 创建一个全局字符串数组用于存储待输出的语法信息以及一个存储空格的字符串



1. 语法分析器读取到相关的语法信息，并调用相关的parse函数时，函数开始时将语法信息（向后移位数+语法类型）存入字符串数字中



1. 当语法分析器读取到该语法的下一个tok时，将space压入字符串数组

1. 当该语法分析结束时，字符串space向前缩进一个字符，语法分析器调用中间代码生成。



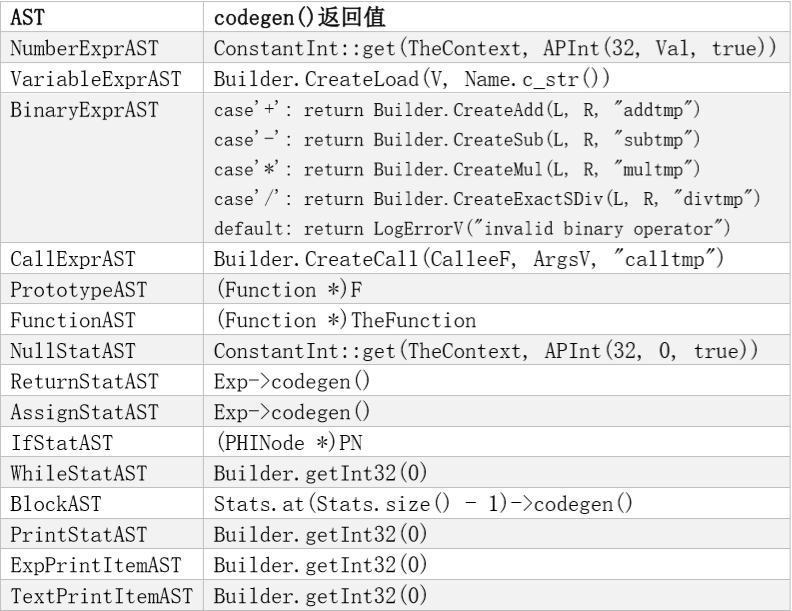
1. 最后在生成目标文件时，将字符串数组中的内容输出到parser.txt文件中

**3.实现代码生成，并输出对应的LLVM IR代码**

**3.1设计实现代码生成**

中间代码生成的函数调用结构和之前的语法分析器的调用结构完全相同（把Parse函数换为对应AST的codegen()函数即可），因此，我们只需要实现每个AST的codegen()函数即可。

1. 根据LLVM的需求创建相关的全局变量（如LLVMContext TheContext）和打印错误信息的函数Value \*LogErrorV(const char \*Str)
2. 在每个AST类中定义虚拟代码生成（codegen）方法，每个方法都返回一个LLVM Value\*对象



上图是所有AST对应codegen()函数应有的的返回值。LLVM文档中已经给出了部分codegen函数的代码，我们只需做适当修改即可。剩余部分设计和实现如下：

**ReturnStatAST：**较为简单，只需调用Exp的codegen()方法并返回即可；

**AssignStatAST：**调用Builder.CreateStore创建存储后，返回Exp->codegen()；

**IfStatAST**：我们通过调用CreateCondBr函数来产生跳转指令，从而实现布尔表达式的语义。在使用函数CreateCondBr(CondV, ThenBB, ElseBB)时，有这么几个约定：

1. 当CondV为真时，跳往ThenBB基本块；
2. 当CondV为假时，跳往ElseBB基本块；
3. 紧随“函数CreateCondBr所生成的跳转指令”之后的基本块为ThenBB。

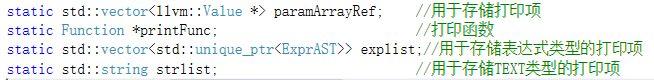
对于“if(exp) stmt”，我们需要创建ThenBB基本块；对于“if(exp) stmt else stmt”语句，我们需要创建ThenBB和ElseBB这2个基本块。因此我们要为then和else创建块，并在函数末尾插入'then'块。'Then'、‘Else’的Codegen可以更改对应的块，为PHI更新ThenBB或者ElseBB。最后返回ifStatAST的PHI节点。

**WhileStatAST：**对于while控制流语句和if语句基本相同，WhileStatAST分为LoopBB, AfterBB以及一个控制循环表达式Exp

**BlockAST：**遍历调用其中所有元素的codegen()方法并返回最后一个statement的codegen()，如果statement为空则返回Builder.getInt32(0);

**PrintStatAST**的中间代码生成则较为复杂，这里的方法是编译的时候，利用动态链接库，直接链接到C库中的printf函数。具体实现：

首先创建如下全局变量，作用如注释：



然后创建打印函数的声明方法，在PrintStatAST的codegen()最前面调用实现声明，然后就可以以合适的参数调用.

(参考网站：https://stackoverflow.com/questions/35526075/llvm-how-to-implement-print-function-in-my-language)

需要注意的是，如果需要打印多个字符串和表达式，那么

每当遇到TEXT类型的打印项，则直接将字符串压入strlist中（在TextPrintItemAST的codegen()函数中实现）；

如果遇到表达式类型的，则将"%d"压入strlist中，将Exp压入explist中（在ExpPrintItemAST的codegen()函数中实现）；

最后在PrintStatAST的codegen()函数中遍历调用每个PrintItemAST的codegen()函数，然后将strlist、explist中每个元素的codegen()压入paramArrayRef，最后以paramArrayRef为参数，调用已经声明的打印函数即可

Builder.CreateCall(printFunc, paramArrayRef, "printcall");

**3.2输出对应的llvm IR代码：**

在HandleFuncDefinition()，调用FnAST->codegen()->print(errs());即可

**4.增加JIT支持和支持优化**

//姚连杰TODO

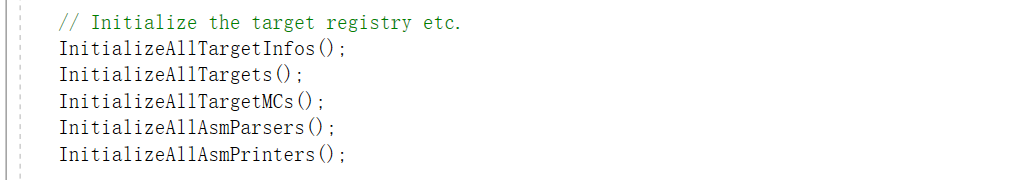
**5.利用LLVM后端，生成本机机器代码**

**5.1选择目标**

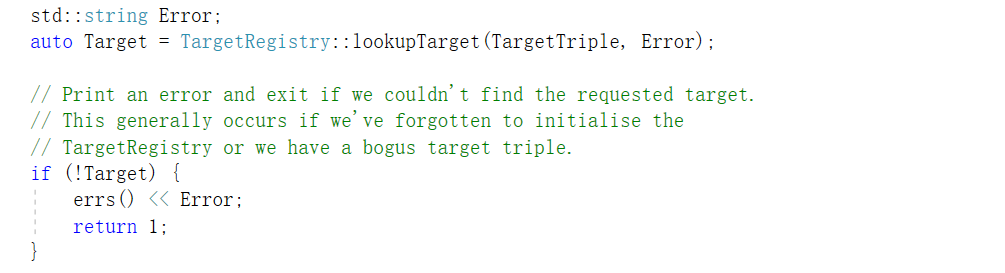
首先，我们要知道LLVM是支持交叉编译的。我们以当前的计算机作为目标，因为LLVM提供sys::getDefaultTargetTriple，利用它可以返回当前计算机的目标三元组，所以我们不需要硬编码目标三元组来定位当前机器。



我们初始化所有的目标来生成目标代码。

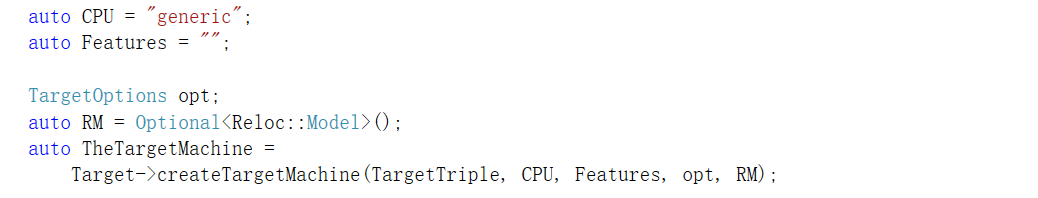


我们使用返回的目标三元组来获得Target



**5.2 TargetMachine类**

TargetMachine类为我们提供了完整的目标机器描述。我们可以利用它来针对某一特定功能和特定的CPU。现阶段，我们选择使用通用的CPU。



**5.3配置模块**

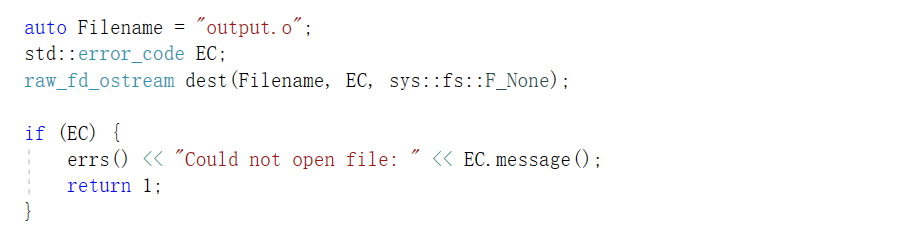
为了实现进一步优化，我们通过配置模块来指定目标和数据布局。（不是必须的）





**5.4生成目标代码**

定义我们要将文件写入的位置



定义了一个生成目标代码的pass，然后我们运行该pass

