语法自扩展编译系统详细设计文档

一、写作目的

本文是可扩展编译系统的详细描述，主要介绍本编译系统的语言特性、设计特点、具体实现方法、各个组件基本功能、扩展接口设计等等。

本文适合本系统的开发人员阅读。

二、语言特性

1.跨平台

我们由于后端是采用LLVM支持的，带有跨平台的功能，而且既可以解释执行（自带JIT），也可以编译到对应平台。 目前我们使用的LLVM版本是3.6版，支持的主要平台为：

- x86
- amd64
- ARM
- PowerPC

2.语法的可扩展性

我们自行开发了一款脚本驱动的LALR语法分析器，这款语法分析器可以在运行中加载新的语法配置文件，生成新的分析表，然后对文法进行分析。可以在运行中通过接口控制，是这款语法分析器最大的特点，并且由于是使用lua作为其核心脚本引擎，lua的原生库也可以使用。其主要的扩展接口如下：

- include 包含一个软件包
- add_bnf 添加一条bnf范式
- add_bnf_file 添加一个新的语法描述文件
- load_default_bnf 加载默认的语法文件
- load_empty_bnf 加载空的语法解析环境
- parse_file 编译一个目标文件
- print_bnf 打印当前bnf范式集合
- print 打印一个编译时变量
- 其他lua标准函数

3.设计支持的编程模型

过程式编程

经典C风格程序，并且也可以调用其他C库函数。这部分原生支持函数的部分，但一些重要的部分也会拆到外部来实现。

函数式编程

由lisp风格的函数调用实现，但暂时并不支持很多函数式特性，而是作为外部库和扩展语言特性来实现。

面向对象编程

对象模型并不是内置的，而是几乎全部由扩展来实现，我们会实现一个对象模型作为其标准库的一部分，而这部分也像上两部分一样，是可以重开发和允许用户自行扩展的。

模块管道式模型

增强版面向对象模型，在对象的层次上新增了管道式接口，用来处理一个对象的全部异步数据传入传出请求，方便并行化程序的编写。

其他编程模型

尽量实现成库的形式，让语法自动扩展

4.基础语言元素

表

这是仿Lisp的表结构，是唯一的元语言类型，所有的代码，在语法解析后，都会被翻译成元表。例如如下的翻译：

```
class Main {

    static void hello(int k, int g) {
        print("hello world");
    }

    static void main() {
        hello(1,2);
    }

}
```

翻译后：

```
Node
    (String class
    String Main
    Node
        (Node
            (String function
            String void
            String hello
            Node
                (Node
                    (String set
                    String int
                    String k)

                Node
                    (String set
                    String int
                    String g)
                )

            Node
                (Node
                    (Node
                        (String call
                        String print
                        String hello world)))
            Node
                (String static)

        Node
            (String function
            String void
            String main
            Node ()
            Node
                (Node
                    (Node
                        (String hello
                        Int 1
                        Int 2)))

            Node
                (String static)
            )
    )
```

表也能单独的被程序识别，是基础元编程的重要部分。

宏

宏是编译时的函数，所有的描述符都是宏，例如上面例子中的：class， function 等等，均是宏。 宏是翻译的规则，将对应格式的内容填充到宏函数中。

不是宏的部分，直接作为元数据直接向下传递 函数调用要翻译为call宏

宏的C定义方式是，写一个编译器插件的so，然后编写一条函数，形式如下：

```
Value* MacroDemo(Content* list);
```

然后调用接口将这个函数指针注册即可。 C实现的宏就是在解析这个语法树，完成基础的翻译工作，将Content*包含的语法树翻译成LLVM指定的组件形式，最后拼接在一起。 同时，为了解析一个语法树，有时需要调用其他的宏，解析其内部节点，这时会由系统递归地先完成内部节点的翻译，然后最后再翻译这条。

例如，我们在矩阵运算时，频繁使用了双重for语句，假若我们可以自定义一个宏，能够展开成如下双重for语句：

```
for (int i = 0; i < 10; ++i) {
    for (int j = 0; j < 10; ++j) {
        // do some work
    }
}
```

对应的宏语句如下设计：

```
for_matirx (i, j, 0, 10) {
    // do some work
}
```

其宏定义部分是这样的：

```
defmacro for_matirx (list, code) {
    for (int list[0] = list[2]; list[0] < list[3]; ++list[0])
        for (int list[1] = list[2]; list[1] < list[3]; ++list[1])
            code
}
```

5.基本语法

语法都是EBNF决定的，只要能将良好表示的类C风格代码，转换为S表达式的正确格式，就完成了语法解析的基本任务。