SASL开发者指南

# 设计概述

## 代码路径与项目结构

SASL代码路径在<salvia\_root>/sasl下。

图 1 SASL代码目录结构

SASL包含的工程包括：

* Common
  + 在SASL共享的接口
  + Token的定义
  + Diagnostic信息的定义和实现
* Parser
  + Tokenizer
  + Lexer
  + Parser
* Semantic
  + Semantic Analyzer
  + Symbols
  + Type system and conversation
* Code\_generator
  + APIs
  + LLVM based code generator
    - Common Code generator
    - Code generator service（LLVM的封装）
    - SISD Specified Code Generator
    - SIMD Specified Code Generator
  + SASL VM（Deprecated）
* Driver
  + Static and DLL Driver
* Host
  + Shader Host（用于管线集成）

## 第三方依赖

SASL使用了boost（1.44或以上）和LLVM（3.0或以上）。如果直接使用SALVIA编译，则遵照SALVIA编译文档说明即可。

对SALVIA的其他库，SASL对eflib有实现依赖，对salviar(salvia renderer)有部分头文件依赖（主要集中在salviar的shader\_abi.h和abi\_info中）。

## 静态结构

SASL是传统的多遍编译器。

图 2 多遍编译器的工作流程

### 词法分析

SASL的词法分析分为两步，第一步将所有的代码处理成一组word，第二部将word处理成token流。这样做的原因是为了将preprocessor控制在word的处理阶段。静态结构如下：



图 3 词法分析阶段类图

Code\_source和lex\_context是两个纯接口类。当调用了lexer.tokenize()后，Lexer将循环调用code\_source中的next来获得下一个word，并在lex\_context接口中获得文件、行列数等token的位置信息（严格说lex\_context是code\_source实现行列统计的一个算法），并返回一个token\_t的序列，这是语法分析时候需要的token流。

在driver中提供了一个driver\_code\_source实现了这两个接口，并且driver\_code\_source使用了Boost.Wave，提供了与C99/C++0x相同能力的preprocessor功能。

SASL的Lexer基于Boost.Spirit.Lexer开发，重新进行了封装便于第三方代码对词法进行定制。之所以没有使用Boost.Qi，主要是出自于编译器时间的考虑。

### 语法分析

#### 基于Combinator的语法定义和语法分析器生成

SASL的语法分析采用了基于Combinator的语法分析器构造工具。VCZH有一组很好的文章<http://www.cppblog.com/vczh/archive/2008/06/06/52318.html>，《构造可配置的语法分析器》，讲述了Combinator的机制。简单的将，Combinator就是个用Compositor模式拼起来的递归下降法解析器。

Combinator的代码在generator.h和generator.cpp中。语法在grammar.h与grammar.cpp中。

SASL提供了一系列combinator的操作符，这些操作符实际上是由多个类型Parser来实现的（图 4）：‘=’为Rule Definition，类似于EBNF的‘::=’；‘|’表示分支结构，会生成一个A | B这样的规则，会生成一个Selector Parser；‘>>’表示顺序结构；‘>’为强制顺序（如果不满足就报错）；‘\*’为重复；‘-’为不匹配规则。利用STERM(token)可指定term作为语法要素。



图 4 Parsers（部分）

因此，最终的语法可能用以下类似于EBNF的形式定义语法：

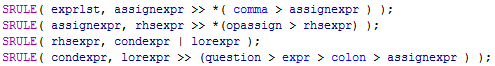


图 5 类EBNF的语法定义

#### Parser的分析结果

每一个Parser都会生成一个Attribute作为返回。不同类型的Parser生成的Attribute类型也不相同。



图 6 Attributes（部分）

每个attribute都会有一个rid（rule id），用于标示Attribute是哪个Rule生成的。如果attribute不是由Rule生成，而是由中间parser生成，例如，如果attribute是由A >> (B | C)中的(B|C)这个Combinator表达式生成的selector\_attribute，那么这个attribute的rid恒为-1。

所有的attribute会构成一个树状结构，叶节点为terminal/token，根节点由根rule（SASL中为program）生成。这棵树我们称之为parse tree（分析树），它是语法分析的直接结果。

#### AST及其生成

# 操作符与内建函数开发指南