SOFTART DEVELOPMENT GROUP

实用编译器构建指南

以 SoftArt Shader Language 为例

空明流转（Ye Wu）

SoftArt Development Group

2009/12/12

本文利用现有的编译器前端或后端的技术和库，以可控制和渐增的方式，从无到有，从小到大，从简单到复杂，从低效到高效的实现一个编译器。在本文里，编译器的构建过程被分解成多个迭代的阶段。其中的大部分阶段，你都能够在理解它之后，在一个小时到一天不等的时间内达到预计的目标。

实用编译器构建指南

Ye Wu，SoftArt Development Group

wuye9036@gmail.com

目录

1. 前言

在你看到这篇文章的时候，你一定在想，编译器啊，真是个麻烦的东西。呃，没有错，在撰写此文的时候，我也一直有着这样的念头。本文不能将你从编译器的苦海里救出来，只是希望它能够让你少游一会儿，以尽可能的少喝水。

与一般的文章不同，本文不会涉及到有关于编译原理的理论细节，NFA，DFA，递归下降，文法，语法制导翻译等等概念，都不会深入的追究。但是希望你能有这方面的概念，这样在讨论到一些算法及其实现的时候，不至于理解的太过费力。

本文重点在于利用现有各式各样的编译器前端或后端技术和库，以可控制和渐增的方式，将我们的编译器从无到有，从小到大，从简单到复杂，从低效到高效的实现出来。本文的写作目标是，我们将编写编译器的任务，分解成多个迭代的阶段，其中的大部分阶段，你都能够在理解它之后，在一个小时到一天不等的时间内达到预计的目标。这样我们才有动力进行下去，不是吗？

唔，自然，这样的文章需要一个完整的实例贯穿前后。我们决定采用 SoftArt 项目内最复杂的一个子项目 SoftArt Shader Language（简称 SASL）作为一个导引，你所看到的行文过程，就是 SASL 的开发过程。SoftArt 开发到哪里，这篇博文就写到哪里，我们对 SVN 发誓，我们所经历的，你都能看得到。

SoftArt 的项目主页 http://code.google.com/p/softart

SoftArt 项目介绍
http://www.cppblog.com/lingjingqiu/archive/2009/12/07/102698.html

换句话说，我们所走的弯路，你一样得走（哈哈，被我们坑的），我们所路过的捷径，一样也会成为你的捷径。祝大家好运。

2. 更新说明

本文是一篇随着项目进行动态更新的文章。DOCX 格式的文章为最新文章，PDF 文章则会不定期更新。本节将维护一个更新列表，用于表明文章的更新情况，并提供每一节和 SVN 代码库里的对应情况，需要完整示例代码的可以去 SVN 中提领对应版本的源代码。

- Rev. 32：新增章节：前言 —— 迷你虚拟机
- Rev.33：新增章节：语法树到 OP CODE
- Rev.35：新增章节：词法分析

3. 本文的使用说明

在阅读本文的时候，我们希望您能有以下的基础：

- 熟悉至少一门高级语言
- 能阅读带模板的 C++（我们的示例代码用此所写）
- 会使用正则表达式
- 了解计算机的工作原理，能阅读汇编
- 对 LL 和 LR 文法有一定的了解
- 掌握基本的数据结构和算法

如果您具备以下条件，那么恭喜您，本文就是为您而写的：

- 需要快速开发编译器
- 对编译原理了解不深
- 对编译速度和编译后的程序在性能上要求不甚苛刻

如果您具有以下特征之一，那么本文可能会给您带来不适，请谅解我们的工作：

- 天然喷
- 紫禁之巅的高手
- 车轮党
- 反模板联盟

4. 迷你虚拟机

哈，我想你一定会问，不是说编译器么，为什么要先说虚拟机呢？唔……用一句话来回答就是，因为它必要而简单。还记得编译原理中描述的一个语言是如何被分析、解释并执行的么？

在词法分析->低级代码生成四个阶段里，每个阶段的输出都作为下一个阶段的输入。这些输出通常都是规范化的，而且都以计算机容易识别的形式保存。依据这些阶段的输出来判断程序运作的正常与否，是需要很多经验的。这显然不是我们所希望的。

那么最后一个阶段，低级代码的执行，便成为了最容易实现，也最容易检验的阶段。在这一阶段，我们只需要设计一个虚拟机便可以达到目的。

唔，虚拟机。是个很可怕的名词，不是么？没关系，对我们的实现来说，虚拟机最大的好处，就在于它可大可小。我们可以实现一个 x86 一样指令集的庞然大物，也可以实现一个只有几条到几十条我们所需要的指令构成的小小玩具。

不过，既然叫虚拟机，那麻雀虽小，五脏俱全。咱们这只小麻雀，看中了谁的五脏呢？一个大家都学过的典范，8086。还是觉得复杂了，是吗？没关系，我们先来看看，8086 的哪些部分是最必要的，我们只要实现了这一部分，就能让虚拟机跑起来。

嗯……我们来看看，8086 里面，要做一个 1+1=2 的运算，需要哪些最基础的设施呢？**通用寄存器，栈，指令寄存器，指令槽**，当然还有最重要的**执行引擎**。好吧，其余的什么段啊，分页啊，中断啊，我们一概不需要。

好，那么我们便用最直截了当的方式，把这些我们需要的设施拼凑成我们的虚拟机。

```
class vm{
    vector<instruction> code_buffer;
    vector<byte> stack;
    int r[16];
    intptr_t eip;
    bool execute(const instruction& ins);
};
```

唔，有了这些基本的设备之后，我们得让它运转起来。我们先设计一个最简单的驱动代码，这个代码依次读入 code_buffer 里 eip 位置上的指令，然后执行这条指令，执行完后，移向下一条，知道指令槽空为止。

```
class vm{
    bool call(){
        eip = 0;
        while ( eip < code_buffer.size() ){
            if( execute(code_buffer[eip]) ){
                ++eip;
            } else {
                break;
            }
        }
    }
};
```

知道这个虚拟机怎么去执行指令后，还得要定义有哪些指令可以执行，每条指令执行后会产生什么效果。这些能够执行的指令的集合，就称为虚拟机的**指令集**。而后者，需要在我们的执行引擎里，编写实际的执行代码来达到每个指令的效果。

什么？你说要实现 x86 的指令集？天哪，那些指令仅仅是读就够你读上半天的。一开始，我们只需要实现最基本的几条指令，告诉我们，这个虚拟机可以 works，这个就 OK 了。我想大家的数学都是从 1+1=2 开始的，这里，我们也用加法运算做为开始。由于咱们这个虚拟机是仿照 x86 的，所以所有的运算都要在寄存器里完成。所以我们只需要两条指令便可以完成运算：读入数据到寄存器的指令和在寄存器里执行加法的指令，咱们的数据类型，只有万年的 32 位有符号整数，简称 int。

明确了我们需要什么指令，就要设计指令的格式了。本着抄袭……啊，不，是借鉴 x86 的精神，我们也将指令设计成三元组的形式：指令码，操作数 1，操作数 2

够用吗？这个你去问 x86，它会告诉你，历史证明，三元组基本够用。当然，其实我觉得四元组更好使……也许我会把它改成 4 元组的。唔，咱们不 YY，先来看三元组的数据结构，简单明了。

```
struct instruction{
    op_code op;
    int arg0;
    int arg1;
};
```

真没啥想头的，不是嘛？嗯，有了指令的数据结构，咱们就可以设计指令了。

在 instruction 里，op_code 是一个指令码的枚举

```
enum op_code{
    op_add,
    op_loadrc
};
```

op_add 执行的是寄存器之间的加法，arg0 是目标寄存器号（实际上就是寄存器数组 r 的下标），arg1 是源寄存器号；该指令将 arg1 中的值加到 arg0 中。

op_loadrc 将一个数值读入到寄存器中。Arg0 是寄存器号，arg1 是数值。

然后在 execute 里面实现这两条指令

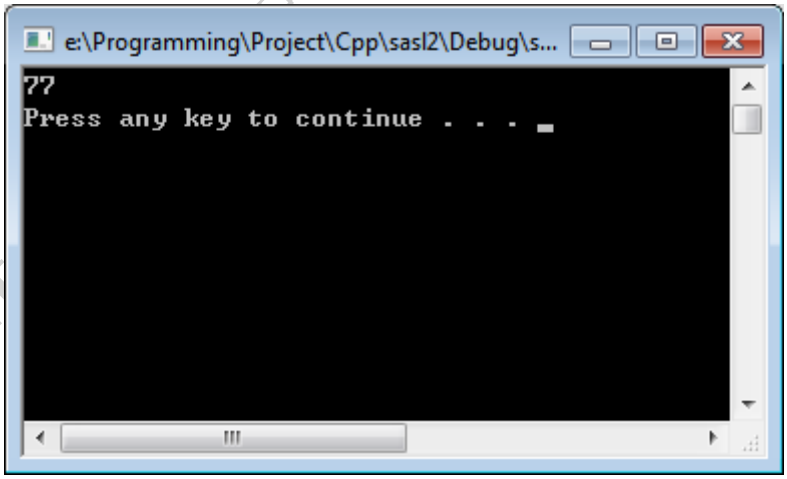
```
bool execute (op_code op, int arg0, int arg1){
    switch (op) {
    case op_add:
        r[arg0] += r[arg1]; break;
    case op_loadrc: {
        r[arg0] = val;  break;
        }
    }
    return true;
}
```

最后，再添上一些辅助性的代码，这个虚拟机就能够正常的工作了。不信？咱们可以试试：

```
int _tmain(int argc, _TCHAR* argv[])
{
    code_generator cg;
    cg
        //读取两个常量
        .op( op_loadrc, r0, 10 )
        .op( op_loadrc, r1, 67 )
        .op( op_add, r0, r1) //执行加法
        ;

    vm machine;
    int result = machine.raw_call( cg.codes() );
    std::cout << result << endl;
    system("pause");
    return 0;
}
```

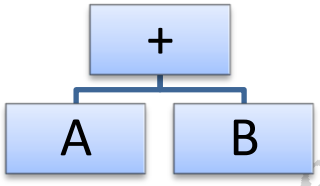
呵呵，Run 起来吧~



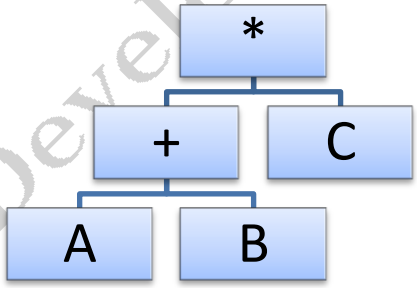
5. 从语法树到 OP CODE（一）

知道咱们的虚拟机能够执行 OP CODE 之后，下一步就要考虑，怎么从语法树里面生成咱们需要的 OP CODE 了。简单来讲，语法树就是将程序的逻辑按照树状组织并保存在内存中的一种形式。有关于更详细的信息，搜“Syntax Tree”，到处都是解释。

一时不明白也没关系，我们来看一个直观的例子。考虑a + b这样一个基本形式的表达式。这个表达式既可以按照我们所写的这样，分为 a，+，b 三个部分串行表示，也可以表示成下图的样子



可能一个表达式你还看不出来树形的优势。要是表达式级联起来，就显示出这种表示的威力了：

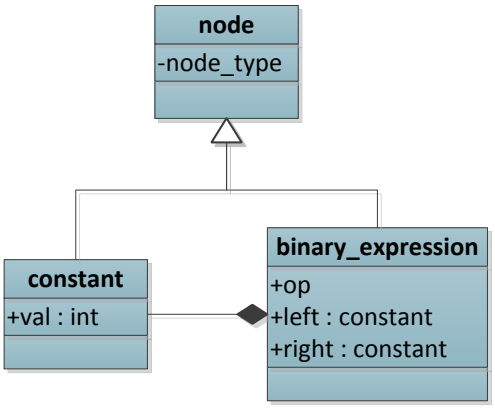


这样一个语法树，可以不借助任何别的手段，保存了表达式的优先级关系。这里的语法树表示的就是(A + B) * C的表达式。同时，在语法树上求值也很方便，后根遍历语法树就可以了。即先算出左右节点的值，再根据当前节点符号求出当前节点值。

王阳明说，知行合一。知道了语法树是什么东西，我们就要开始考虑怎么用了。“怎么用”这个问题可以分成两个部分，第一，语法树怎么实现。第二，语法树怎么生成 op code。啊，先不要把语法树想象的这么复杂。在这里，我们的运算符只有加号，一个加号也只能带两个 int 的值节点，而不能递归的带上一个符号节点。也就是说，这棵树只可能有一种形式而已。

首先来解决语法树怎么实现的问题。在这个问题上，我们只需要把握一点，

语法树是一个天然的 composite 模式。我们用一个 UML 来看看这个只有加法算符的语法树定义：



唔，很简洁，不是么。Node_type 是一个 syntax_node_types 类型的枚举，这个枚举告诉以后的代码生成器这个抽象的 node 究竟是个什么类型，然后代码生成器再还原它原本的类型并生成适当的代码。op 是一个 operators 类型的枚举，表示一个二元运算的操作符。对于本例，只有 operators::add 可用。

在有了基本实现之后，再考虑一下其它需求，例如语法树节点类型之间的可能存在的循环依赖问题，语法树的深浅拷贝问题，等等，最终 SASL 的语法树节点接口是这样的：

```
struct node{
    syntax_node_types type;
    template <typename NodeT> NodeT* clone() const;
    template <typename NodeT> NodeT* deepcopy() const;
protected:
    virtual node* clone_impl() const = 0;
    virtual node* deepcopy_impl() const = 0;
};
```

```
struct binary_expression: public node{
    operators op;
    boost::shared_ptr<constant> left_expr;
    boost::shared_ptr<constant> right_expr;
};
```

```
struct constant: public node{
    int val;
};
```

道理复杂，不过实际上，并没有那么复杂吧？

下面来解决第二个问题：怎么用表达式树产生代码？我不多解释，直接上代码，相信你一定会看明白的：

```
vm_codegen& vm_codegen::emit_expression( const binary_expression& expr ){
    if ( expr.op != operators::add ){ return *this; }
    int c0 = expr.left_expr->val;
    int c1 = expr.right_expr->val;
    ins_.push_back( instruction( op_loadrc, r0, c0 ) );
    ins_.push_back( instruction( op_loadrc, r1, c1 ) );
    ins_.push_back( instruction( op_add, r0, r1 ) );
    return *this;
}
```

然后我们将生成语法树，生成 code，运行 code 的代码补上，运行，OK~

你一定会说，啊，硬性绑定寄存器！太可怕了！如果表达式复杂了该怎么办呢？呵呵。这些都是以后的问题了。以后的问题，就由以后的我们去解决好了。今日事，今日毕，时间不早，咱们还是洗洗睡了。

6. 起源：词法分析（一）

不管你学什么样的外语，大约都是从词汇开始。词，是一个语言里最小的语义单元。编译器阅读你的语言，也是如此。所以第一件事情，就是要把整个文法打散成一个一个的单词。在这里，我们把这些单词叫 token。

怎么进行词法分析，此处就不再赘述，这是一个上下文无关文法的匹配问题。如果需要理解词法分析的原理，或者手工编写词法分析工具，可以参考陈梓翰提供的两篇极好的教程。在 SASL 里，我们不再发明轮子，而选用已有的词法分析工具。

可选的词法分析工具很多，例如出名的 Lex 及其改进 Flex，ANTLR 等。对于 C++而言，这些方法多属于产生式的方法，就是用一段不太靠谱的代码去生成另外一些更不靠谱的代码。更重要的是，这些代码的编译、调试都不方便。所以最终我们还是选择了一个在用 C++实现、并且可以直接在 C++里书写词法和语法的分析器产生工具，它就是 Spirit。

Spirit V1.8 和 V2.1 都是 Boost 库里的一个部分。需要注意的是，Spirit 的 V1 和 V2 是完全不兼容的两个库。在这里，我们选择了 V2 作为我们的词法和语法分析工具。Spirit V2 总共分为 3 个部分，负责语法分析的 Qi，格式化打印的 Karma，和词法分析器 Lex。此外，Spirit 还有一个类似于 boost.mpl 和 boost.lambda 的库 phoenix，这个库也常被用于词法和语法分析中。详细的使用指南和参考，可以参见 Spirit 的文档。

由于 Spirit.Lex 大量运用了 Template Meta-Programming 和编译器推导，因此编译时很容易出错，而且错误信息难于定位；同时 Spirit.Lex 的指南也写得非常简单，它所演示的特性，不足以用来实现一个完整的编译器。因此，这里我们也将给出另外一个快速指南，以展示那些我们在撰写编译器时所用到的技术和特性。

这里我们仍然以 A+B 这样一个简单的表达式为例，其中 A 和 B 都是一个字面值的整数，A+B 之间没有其他空格填充。这样我们就可以把这个“句子”拆分成 A，+，B 三个 token。例如“33+65”就可以被拆分成“33”，“+”，“65”三个 token。对于这样一个表达式，我们只需要下面两个正则就可以完成词法分析：

```
literal_int = “[0-9]+”;
literal_add=”\+”;
```

由于 C++里面“\”是转义符，因此实际上 literal_add 实际上应该写成“\\+”。然后我们需要用 Spirit 来实现。

Spirit 中，首先定义一个 tokens 列表：

```
template <typename BaseLexerT>
struct sasl_tokens : public boost::spirit::lex::lexer< BaseLexerT >
{
    sasl_tokens(){
        littok_int = "[0-9]+";
        optok_add = "[\\+]";

        this->self =
            littok_int
            | optok_add;
    }

    boost::spirit::lex::token_def<> littok_int, optok_add;
};
```

然后，我们利用这个 token 列表生成一个词法分析器 sasl_tokenizer：

```
typedef boost::spirit::lex::lexertl::lexer<> sasl_lexer_base;
typedef sasl_tokens<sasl_lexer_base> sasl_tokenizer;
```

最后来执行一下我们的 tokenizer。在执行之前，我们写一个 callback 函数，这个函数在每分析出一个词之后，都会被调用一下，我们用它来判断我们分出的词正确与否：

```
struct token_printer{
    template <typename TokenT> bool operator()( const TokenT& tok ){
        cout << "token: " << tok.value() << endl;
        return true;
    }
};
```

最后执行一下词法分析：

```
boost::spirit::lex::tokenize(first, last, sasl_tok, token_printer());
```

first，last 是输入字符串的迭代器。如果输入为“55+65”，那么屏幕上就会依次打印出“55”，“+”，“65”的三行。

不过，如果你在“55+65”之间敲入一个空格，例如“55+_65”(‘_’代表空格)这样的，那么词法分析就会失败。因为“_”这个字符，没有合适的词可以匹配。即便是匹配了，空白这个 Token 也没办法用在语法树之中，最终也会导致语法分析失败。而在程序语言里，支持空白符号的过滤掉是必不可少的。所以，下一次，我们就要讲语法，并讨论一下空白符的过滤问题，让我们可以自由写出美观的语句。

7. Exodus：语法分析（一）

在一个魔法世界里，你是一个会魔法的法师。我的意思是，作为一个法师，你什么都会了，也什么都有了，施法材料，法袍，魔杖，法术书。甚至你连成功后的庆祝动作都想好了。你以为你会“魔法”了。只可惜，这里还缺少了一样东西，那就是，魔法的口诀。

而在这里，我们什么都有了。用来分析的 Token，语法树到 OP CODE 的翻译，虚拟机，什么都有了。但是我们还是缺一样口诀，那就是，如何从 Token 到语法树的口诀。

在我们进行词法分析的时候，遵从的是 Spirit 这本颇有难度的《圣经》。不过，我们只浏览了如《使徒行传》般流畅而松散的 Spirit.Lex。在这里，我们依然沿用 Spirit，这是我们编译器前端的原旨。不过现在，我们要讲解的是环环相扣、荡气回肠的《Exodus》——Spirit.Qi。

嘛，这段神叨叨的引子，只是为了强调语法分析的地位而已。在继续阅读本章之前，需要你看的明白 BNF。有关于 BNF 方面的信息，你可以在任何一本讲述编译原理的书籍上找到。

仍然是以一个简单的 A+B 为例。由于我们已经有了词法“literal_int”和“literal_add”，因此 A+B 这样一个表达式，用 BNF 来表示，就是：

```
Expr ::= literal_int literal_add literal_int
```

在 Spirit.Qi 里，语法的表达也类似于 BNF 的形式。只要你设计出语言的 BNF，就很容易的翻译成 Spirit.Qi 支持的语法定义。我们这里，就可以写成：

```
template <typename IteratorT>
struct binary_expression: qi::grammar<IteratorT>{
    template <typename TokenDefT> binary_expression(const TokenDefT& tok):
binary_expression::base_type(start)
    {
        start = (  literal_int >> literal_op >> literal_int );
        literal_int = tok.littok_int;
        literal_op = tok.optok_add;
    }
    qi::rule<IteratorT> literal_op, literal_int, start;
};
```

在 Spirit.Qi 中，一个 Rule 就等于 EBNF 的一个非终结符。一个 Grammar 相当于一组 Rule 的集合，并且可以拥有一个或者多个的起始 Rule 作为入口。本质上我们可以把 Grammar 看成一个 Rule（准确的说，是 Parser，若要了解相关概念，请参阅 Spirit 的自定义 Parser 部分）。等号用于连接非终结符（Rule）及其推导式；使用“>>”（输入流/右位移运算符）连接语法要素之间的连接符号。更多的符号请参阅 Spirit.Qi 文档。

至于为什么不将 Rule 合并到一起，而提供一个 Grammar 的中间层，主要有两方面的考虑，一个是提供了一个抽象层，例如我们可以把 Statement 和 Expression 分开来写，使得层次上更加清晰；还有一个方面在于节省编译时间。因为 Spirit 使用了大量的元编程技术，如果把所有的 Rule 合并到一起编译，会占用大量的编译时间。在使用了 Grammar 之后，可以运用 C++编译器在一个编译过程里对相同的模板特化只进行一次的 Tricks，大大节省了编译时间。

在上一章里，咱们最后还留了一个问题，就是空白符号的处理方法。这里，我们将于空白符号一起，来走一下 Spirit 的语法和词法分析的流程。

首先，我们建立好词法，将源代码字符流组织成更加容易被语法分析识别的 Token 流。

```
template <typename BaseLexerT>
struct sasl_tokens : public boost::spirit::lex::lexer< BaseLexerT > {
    sasl_tokens(){
        this->self.add_pattern("SPACE", "[ \\t\\v\\f]+");

        littok_int = "[0-9]+";
        optok_add = "[\\+]");
        whitetok_space = "{SPACE}";

        this->self = littok_int | optok_add;
        this->self("SKIPPED") = whitetok_space;
    }
    boost::spirit::lex::token_def<>
        littok_int, optok_add, whitetok_space;
};
```

这里，我们将词法分为两组，对语法分析有效的 Tokens 组和无效的空白组，空白组用”Skipped”作为状态以示区别。这里我们需要说明一下，Spirit.LEX 的词法分析的“状态”与词法分析工具“Lex/Flex”中的状态概念是相同的。

在 Lex 类的词法分析工具里，有一个专门的状态。一般而言，这些状态都用字符串表示。Lex 的默认是“INITIAL”，Spirit.Lex 的默认状态是空（如果我没记错的话）。在指定词法的时候，可以告诉词法分析器，此文法在什么状态下，这条词法才发挥作用。词法分析器的状态可以由外部程序自由指定。

我们将表示空白的词法都放在 Skipped 状态下后，我们就可以对每个单词，用 Skipped 状态去匹配。如果发现是在 Skipped 状态下匹配成功的单词，在进入语法分析前就可以先丢弃，进而实现过滤空白符的目的。

考虑表达式“55_+38”（‘_’代表空格），在分析成 Token 流之后，会变成以下的形式：

State	INITIAL	SKIPPED	INITIAL	INITIAL
Token	Literal_int	Literal_ws	Literal_op	Literal_int
Literal	55	_	+	38

然后撰写我们的 Grammar。由于我们需要指定 Skipper 来跳过我们不需要的

Token，因此我们的 Grammar 在模板参数里，也要加入这个 Skipper 的类型参数。

```
template <typename IteratorT, typename LexerT>
struct binary_expression:
    qi::grammar<IteratorT, qi::in_state_skipper<LexerT> >
{
    template <typename TokenDefT>
    binary_expression(const TokenDefT& tok):
        binary_expression::base_type(start)
    {
        start = (  literal_int >> literal_op >> literal_int );
        literal_int = tok.littok_int;
        literal_op = tok.optok_add;
    }
    boost::spirit::qi::in_state_skipper<LexerT> skipper_type;
    qi::rule<IteratorT, skipper_type> literal_op, literal_int, start;
};
```

并在咱们的驱动代码里面，这样写：

```
typedef sasl_tokenizer::iterator_type sasl_token_iterator;
typedef sasl_tokenizer::lexer_def sasl_skipper;

sasl_tokenizer sasl_tok;
binary_expression<sasl_token_iterator, sasl_skipper> g( sasl_tok );

lex::tokenize_and_phrase_parse(
    first,
    last,
    sasl_tok,
    g, qi::in_state("SKIPPED")[sasl_tok.self]
    );
```

喏，看到了指定 skipper 的代码了不？这就提示 parser，遇到了 skipped 状态解析出来的 token，就自己吃了吧，不要拿去匹配了。这样就达到了过滤掉空白符的目的。

不过呢，尽管我们 parse 通过了，但是仍然没有提取出我们想要的信息来。到目前为止，我们还没能让 parser 构造出咱们之前手工构建并传递给 Code Generator 的语法树来。这仍然是横亘在出埃及的我们面前的红海。

下一次，我们将仍然相信 Spirit 这本 Bible，相信它给我们的一章叫“Semantic Action”的启示录。它将告诉我们，如何把 Parser 分析出的结果转化为我们要的语法树，以引领我们走向流 OP CODE 之地。

God bless programmers and p2p sites except gfw’s developers and Cisco. Amen.