|  |
| --- |
| 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](mailto:wuye9036@gmail.com)

目录

[1. 前言 1](#_Toc248682675)

[2. 更新说明 2](#_Toc248682676)

[3. 本文的使用说明 3](#_Toc248682677)

[4. 迷你虚拟机 4](#_Toc248682678)

[5. 从语法树到OP CODE（一） 8](#_Toc248682679)

[6. 起源：词法分析（一） 11](#_Toc248682680)

[7. Exodus：语法分析（一） 14](#_Toc248682681)

# 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。

# 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

# 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

# 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() );

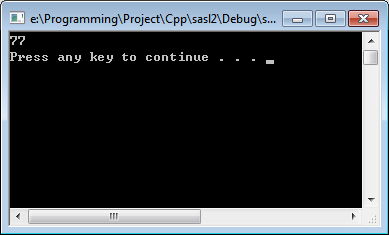
std::cout << result << endl;

system("pause");

return 0;

}

呵呵，Run起来吧~



# 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。

# 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](http://www.cppblog.com/vczh/)提供的[两篇极好的教程](http://www.cppblog.com/vczh/archive/2008/05/22/50763.html)。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。

# 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.**

# 语义：从分析树到语法树（一）

在1-7章里，我们已经建立了一个编译器所需要的绝大部分环节：词法分析、语法分析、代码生成、代码执行。前两个阶段，将会生成分析树（Parse Tree）后两个阶段，则是用语法树生成的。我们多希望语法分析后的分析树，直接就能用作语法树啊！

从结构上看，分析树和语法树几乎是如出一辙的。只可惜，如果我们再仔细的观察会发现，从分析树到语法树有一条深深的鸿沟。是的，你猜得没错，这条鸿沟，就是**语义**。只要有了语义，我们就可以将我们的分析树，变成可以产生代码的语法树。

本质上讲，语法树（Syntax Tree）是含有语义的。仍然用A+B这个表达式的语法树来举例子。在这里，A和B都是一个Int32的常量，例如我们这里A是5，B是10。这个语法树里面A节点，它具有以下的语义：

* A是一个常量。
* A是一个整型值。
* A是5。

这些语义信息在语法树里面都具备，而在语法分析之后的分析树里面，只有“5”这样一个字符串。所以实际上，从分析树到语法树的建立，还需要经历一个附加语义的过程。

在一个常见的编译流程里，语义分析可以分为两个部分。其中一部分会跟随在词法和语法分析中，用于解析一些最基本的语义。例如输入的是不是关键字啦，是不是字面量啦，是不是运算符啦一类的信息，这些语义信息还可能用来指导后一阶段的语法分析。

还有一个部分就是例如类型推导、符号设置、函数签名分析一类的语义分析。这些分析的结果通常并不影响语法树的结构，而放在语法分析阶段又会增加分析的复杂度。这一类的语义分析，通常是在语法树建立好之后，再来对语法树进行进一步的分析，将语法树上的语义信息补完。

在SASL里，我们将语法树的建立分成三个步骤。

第一步，在词法分析和语法分析的同时，进行简单的语义解析。包括字面值、操作符、关键字的提取等。这些一方面是语义，一方面也是为了语法分析服务的。

第二步，我们将语法分析得出来的分析树，转换成我们需要的语法树的形式。我们的语法树上，拥有一些属性。通过这些属性可以给语法树节点上附加产生代码所必须的语义。

第三步，遍历语法树，填充语义，执行一些准备工作。

这样，我们就建立了一颗可以被代码生成工具所识别的语法树。