# Vczh Library++ 语法分析器开发指南

陈梓瀚

# 前言

在日常的开发工作中我们总是时不时需要写一些语法分析器。语法分析器不一定指的是一门语言的编译器前端，也有可能仅仅是一个自己设计格式的配置文件的读写程序，或者是一门用来简化我们开发的 DSL（领域专用语言）。我们可以选择使用 XML，不过因为 XML 的噪音实在是太多，所以自己写语法分析器在有些情况下是必要的，特别是那种经常需要修改的文件，使用 XML 有时候会增加我们的负担，除非我们专门为此开发一个编辑器程序。

这篇文章将紧密结合一个带函数的四则运算计算器的例子（Documentation\Samples\ExpressionCalculator\ExpressionCalculator.sln）来说明如何使用 Vczh Library++提供的工具来大幅度简化我们的语法分析器的开发，并最终给出一个可以编译的例子。虽然这个例子实在是老掉牙了，不过开发一个四则运算计算器可以覆盖大部分开发语法分析的过程中会遇到的问题，所以也不失为一个好的例子。

# 制定语法

我们需要对带函数的四则运算计算器下一个定义，这样我们才可以有目的地完成这个任务。我们对四则运算式子是很熟悉的，一个四则运算式子包含加减乘除、括号和数字。我们还可以支持负号：-a，其实是(0-a)的简写形式。那么什么是支持函数呢？这里我们只考虑单参数函数的情况，譬如说三角函数和对数指数等等。譬如说下面的式子就是满足定义的带函数的四则运算式子：

```
sin(1+2) + cos(3*-4)
```

Vczh Library++使用语法的角度来对待一个字符串，因此我们可以把上面的定义转换成语法。一个语法用来表示字符串的一个子集。我们可以通过语法来表达什么样的字符串是满足规定的，什么样的字符串是不满足规定的。不过一个具有现实意义的语法总是会有一些局限性的，譬如说你很难用上下文无关的文法来表达一个字符串：a…ab…bc…c，其中三种字母的数量都相等。幸好在绝大多数情况下我们都不需要去面对这些高难度的问题，因此可以用一些简单的规则来处理：

```
RULE = EXPRESSION
```

RULE 是这个规则的名字，而 EXPRESSION 是这个规则的定义。语法可以由一条规则组成，也可以由很多条规则组成。当所有的规则都列出来之后，那么每一个规则的名字都是一个字

符串的集合。大部分情况下你需要指定一个“总入口”来代表整个语法。

举个例子，假设我们判断一个字符串是不是无符号整数。一个无符号整数只能由数字字符组成。于是我们可以先用一条规则来代表“数字字符”。这里我们可以使用“|”来代表“或”，那么下面的规则就表示 DIGIT 是’0’或’1’或…或’9’：

**DIGIT = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’**

那么，无符号整数就是“很多数字字符”：

**INTEGER = DIGIT | INTEGER DIGIT**

无符号整数 INTEGER 要么是一个数字字符，要么就是一个合法的无符号整数后面再加上一个数字字符。无符号整数加上一个数字字符仍然是一个无符号整数。

现在可以来检验一下。譬如说“1”是一个无符号整数，那么从 INTEGER 开始，分析“1”所走的路径就是

**INTEGER**
**= DIGIT** **(INTEGER = DIGIT)**
**= ‘1’** **(DIGIT = ‘1’)**

字符串“123”显然也应该是一个无符号整数。“123”是**一些**数字字符组成的，因此走的路径跟单个字符稍微有些不同。这里将会交替使用 INTEGER 的两条路径来模拟循环：

**INTEGER**
**= INTEGER DIGIT** **(INTEGER = INTEGER DIGIT)**
**= INTEGER DIGIT DIGIT** **(INTEGER = INTEGER DIGIT)**
**= DIGIT DIGIT DIGIT** **(INTEGER = DIGIT)**
**= ‘1’ DIGIT DIGIT** **(DIGIT = ‘1’)**
**= ‘1’ ‘2’ DIGIT** **(DIGIT = ‘2’)**
**= ‘1’ ‘2’ ‘3’** **(DIGIT = ‘3’)**

在使用 INTEGER 分析“123”的时候，我们可以交替使用 INTEGER = DIGIT 和 INTEGER = INTEGER DIGIT 这两条规则来将一个 INTEGER 替换成恰好三个 DIGIT，然后再将 DIGIT 替换成’1’、’2’和’3’三个字符，从而确信“123”满足 INTEGER 的定义，因此“123”是一个无符号整数。

替换的过程并不是唯一的，我们完全可以使用另一种顺序来将 INTEGER 替换成“123”：

**INTEGER**
**= INTEGER DIGIT** **(INTEGER = INTEGER DIGIT)**
**= INTEGER ‘3’** **(DIGIT = ‘3’)**
**= INTEGER DIGIT ‘3’** **(INTEGER = INTEGER DIGIT)**
**= INTEGER ‘2’ ‘3’** **(DIGIT = ‘2’)**
**= DIGIT ‘2’ ‘3’** **(INTEGER = DIGIT)**
**= ‘1’ ‘2’ ‘3’** **(DIGIT = ‘1’)**

这正是语法的一个特点：替换顺序与结果无关。

现在我们将这个例子再深入一点，如何用语法规则来描述一个逗号分隔的无符号整数列表呢？逗号分隔的无符号整数列表可以是一个整数“123”，也可以使多个整数“1,23,456”。这也是重复的一种，只是跟 INTEGER 的那种重复有所区别——多了一个逗号。根据上面的描述可以知道，逗号分隔的无符号整数列表有两种情况，第一种是单独的一个整数，第二种是

一个已经完成的列表后面跟着一个逗号和一个整数。那么事情就变得简单了。假设我们使用 LIST 来代表这个列表，那么根据上面的描述我们可以用类似的技巧来描述它：

```
LIST = INTEGER | LIST ',' INTEGER
```

用 LIST 来分析一个数字列表的过程与用 INTEGER 分析一个无符号整数是相似的。因为篇幅问题，这里指展示使用 LIST 处理“1,23,456”的其中一种方法：

```
LIST
= LIST ',' INTEGER                          (LIST = LIST ',' INTEGER)
= LIST ',' INTEGER ',' INTEGER              (LIST = LIST ',' INTEGER)
= INTEGER ',' INTEGER ',' INTEGER           (LIST = INTEGER)
= DIGIT ',' INTEGER ',' INTEGER             (INTEGER = DIGIT)
= '1' ',' INTEGER ',' INTEGER               (DIGIT = '1')
= '1' ',' INTEGER DIGIT ',' INTEGER         (INTEGER = INTEGER DIGIT)
= '1' ',' DIGIT DIGIT ',' INTEGER           (INTEGER = DIGIT)
= '1' ',' '2' DIGIT ',' INTEGER             (DIGIT = '2')
= '1' ',' '2' '3' ',' INTEGER               (DIGIT = '3')
= '1' ',' '2' '3' ',' INTEGER DIGIT         (INTEGER = INTEGER DIGIT)
= '1' ',' '2' '3' ',' INTEGER DIGIT DIGIT   (INTEGER = INTEGER DIGIT)
= '1' ',' '2' '3' ',' DIGIT DIGIT DIGIT     (INTEGER = DIGIT)
= '1' ',' '2' '3' ',' '4' DIGIT DIGIT       (DIGIT = '4')
= '1' ',' '2' '3' ',' '4' '5' DIGIT         (DIGIT = '5')
= '1' ',' '2' '3' ',' '4' '5' '6'           (DIGIT = '6')
```

在开发实际的语法分析器的时候，我们总是需要考虑空格的问题。人们用空格让一个具有严格限制的字符串变得更加易读，譬如说将“1,23,456”变成“1,  23,  456”会让密密麻麻的一对字符变得非常容易看懂。空格也不是乱家的，有些地方可以加空格，有些地方不能加空格。

在上面这个例子里面，如果要支持空格，那么空格除了不能插在 INTEGER 中间，应该可以放在任何的地方。这个时候就带来麻烦了，带空格的语法不是太好写。如果我们让 LIST 支持空格，那会把 LIST 变成下面这个样子：

```
SPACES = <EMPTY> | SPACES ' '
LIST = SPACES INTEGER SPACES | LIST ',' SPACES INTEGER SPACES
```

这里<EMPTY>代表空字符串，所以 SPACES 就是没有空格、一个空格或者很多空格了。因此我们必须在 LIST 里面所有可以加入空格的地方写空格，这会让我们的语法膨胀得很厉害。因此我们必须使用一种方法来让我们免除空格带来的困扰。

# 词法分析

引入词法分析的目的是让我们的语法更加简洁。我们可以将处理空格、注释和分割字符串的工作与语法分析完全分开，那么代码写起来就会更加容易，维护起来也会更加简单了。我们总是倾向于让我们的程序越来越容易理解和维护。

词法分析的目标是将输入的字符串适当分割并抛弃处理掉没有用的部分。“适当分割”

一般来说没有一个明确的规则，应该根据具体情况而定，越方便越好。在大部分情况下我们仅把输入的字符串简单的划分为符号、数字、操作符、字符串、空格和注释等等的简单部分。这些划分一般代表“插入空格会改变意义”。比如说“1234”变成“12 34”之后，就从一个整数变成两个整数了。字符串的情况有点特别，虽然字符串中间插入一个空格还是一个字符串，但是插入空格后的字符串已经不是插入空格前的字符串了，因为内容已经发生了变化。与此同时，在一个整数列表里面，往逗号后面插入一个空格不会影响这个列表所要表达的意义，因此将字符串转换成“整数列表”的工作一般划分在语法分析而不是词法分析里。

处理词法分析的方法一般是使用正则表达式。Vczh Library++提供了一个使用正则表达式来开发词法分析器的类库。关于正则表达式的语法请参考 Documentation\Chinese\Vczh Library++\Regex\Regex.htm#Grammar ，关于这个词法分析器类的内容请参考 Documentation\Chinese\Vczh Library++\Regex\Regex.htm#RegexToken。

在使用 Vczh Library++进行词法分析的开发之前需要掌握正则表达式的简单用法。这里我们假设读者对正则表达式已经入门了。精通是没有必要的，因为词法分析使用到的正则表达式的内容十分简答。我们回到之前的“带函数的四则运算计算器”。经过简单的整理，我们知道一个带函数的四则运算计算器由数字、函数名、操作符和符号组成。

加号与减号的优先级一样，对于语法分析来说他们其实没有区别。乘号与除号也类似。当语法分析结束，语义分析开始的时候，加号与减号的区别才会出现。因此在词法分析里面我们可以把他们当成同样的东西来对待，因此有：

| | |
|---|---|
| BLANK = \s+ | ：空格 |
| ADD = \+\|\- | ：加减号 |
| MUL = *\|/ | ：乘除号 |
| NUMBER = \d+(.\d+)? | ：数字 |
| ID = [a-zA-Z_]\w* | ：函数名 |
| OPEN = \( | ：开括号 |
| CLOSE = \) | ：闭括号 |

我们把分类后的结果叫**记号类型**。一个字符串可以被分成很多记号，每一个**记号**属于一个**记号类型**。如果一个记号不属于任何记号类型的话（譬如问号“?”），那么遇到了词法分析的错误。这个时候我们需要报告错误了。

Vczh Library++有一个简单的方法让我们是用正则表达式表达记号类型，并使用他们来构造词法分析器：

```
List<WString> patterns;
const int BLANK         = patterns.Add(L"/s+");
const int ADD           = patterns.Add(L"/+|-");
const int MUL           = patterns.Add(L"/*|//");
const int NUMBER        = patterns.Add(L"/d+(./d+)?");
const int ID            = patterns.Add(L"[a-zA-Z_]/w*");
const int OPEN          = patterns.Add(L"/(");
const int CLOSE         = patterns.Add(L"/)");
```

```
RegexLexer lexer(patterns.Wrap());
```

为了方便书写正则表达式，Vczh Library++同时支持两种转义符：“\”和“/”。因为 C++使用了“\”作为字符串的转义符，所以在这里我们可以使用“/”，这样写起来会比较清晰。

构造词法分析器的方法很简单，我们将所有正则表达式放到一个字符串列表 List<WString>，然后交给词法分析器 RegexLexer，我们就得到了一个词法分析器了。在分析字符串的时候，每一个记号的类型其实就是该记号的正则表达式描述在字符串列表中的位置。如果发生错误的话，记号类型会变成-1。因为列表的 Add 函数返回添加的元素在列表中的位置，因此就可以使用上面的写法来简单地构造一个词法分析器了。

我们可以用一种简单的方法来使用这个词法分析器。RegexLexer 输出的记号存放在 RegexToken 类型里面，我们可以使用任何容器来存放记号，在这里我们仍然使用 RegexToken。

RegexToken 的定义如下：

```
class RegexToken
{
public:
    int                 start;
    int                 length;
    int                 token;
    const wchar_t*      reading;
    int                 lineIndex;
    int                 lineStart;
    int                 codeIndex;
};
```

RegexToken 记录了一个记号在输入的字符串中的位置、所在的行和在该行内的位置、记号类型和指向该位置的指针。这些信息可以用来做很多事情，譬如在产生错误信息的时候可以精确指定错误发生的位置。

在这里我们需要过滤空格，也就是过滤掉 BLANK 记号，因此我们需要写一个过滤函数：

```
bool IsNotBlank(RegexToken token)
{
    return token.token!=0;
}
```

我们知道 BLANK 就是 0，因此这里直接以 0 代替。有了这个函数之后，我们就可以将输入切割成记好了：

```
List<RegexToken> tokens;
CopyFrom(tokens.Wrap(), lexer.Parse(L"(1 + 2) * abs(-3 - 4)")>>Where(IsNotBlank));
```

执行了这段代码之后，我们就将字符串切割成记号了。这里只用了 15 行就完成了词法分析器的定义并使用词法分析器来分析一个字符串的任务了。

**注意**：如果将一个字符指针传入 lexer.Parse 的话，在获得记号列表之后将这个字符指针删除，那么所有记号中的 reading 将全部变成野指针。lexer.Parse 的参数是 WString 类型，所

以这个例子在执行之后，临时的字符串对象会被删除，因此记号列表中的所有 reading 成员将全部变成野指针。**因此在实践过程中最好先使用一个 WString 变量去保存输入的字符串，然后将这个变量传入 lexer.Parse，之后所有 reading 成员将指向这个变量内部的一个有效指针。**

这个时候我们就可以使用 tokens 里面的信息来做处理了。不过 Vczh Library++还提供了语法分析器的类库，让我们可以不用亲自遍历这些记号。

# 带函数四则运算式子的语法

到了这里，我们可以把数字、函数名和符号当成已经存在的东西来看待了，而且再也不需要考虑空格的问题了。于是我们可以仔细组织带函数四则运算式子的语法：

**FACTOR = NUMBER**
**FACTOR = '-' FACTOR**
**FACTOR = '(' EXP ')'**
**FACTOR = ID '(' EXP ')'**
**TERM = FACTOR | TERM MUL FACTOR**
**EXP = TERM | EXP ADD TERM**

语法的设计直接反映了我们的思考过程。这是一个带有递归的语法。当我们考虑下面的式子的时候

**1*(2+2)*3+4*5*sin(6)+7*8*9**

我们首先使用加减法将式子分割为三个部分

**1*(2+2)*3 + 4*5*sin(6) + 7*8*9**

然后使用乘除法将式子分割为九个部分，然后我们发现(2+2)和 sin(6)他们是一个整体。不过整体仍然是由部分构成的，因此内部还包含表达式。所以不难看出，这里的 FACTOR 代表“整体”，TERM 代表乘除法构成的“第二层表达式”，EXP 代表加减法构成的“第一层表达式”。这个语法同时还代表“先乘除后加减”的计算原则。

但是这里还有一个问题，我们观察一下 EXP = TERM | EXP ADD TERM 这条规则。我们不难发现他们其实是独立的两条规则的组合：

**EXP = TERM**
**EXP = EXP ADD TERM**

第二条 EXP 的规则仍然从 EXP 开始，这种递归称为**左递归**。左递归直接处理起来比较困难，因为你分析到 EXP 的时候很容易陷入一个死循环，因此我们需要拆开它们。

我们引入扩展规则的机制来解决这个问题。如果我们想表达一个循环的话，我们不得不专门为它建立一条规则并命名：

LIST = ITEM | LIST ITEM

如果我们可以简化成 LIST = ITEM+的话，就不需要专门为它起一个名字 LIST 了，而可以直接在各个地方使用 ITEM+。跟正则表达式一样，我们使用+和*来代表循环。因此 EXP 就可以被改写成 EXP = TERM ( ADD TERM)*了。注意(与'('的区别，'('代表一个字符，而(跟平常一样用来规定优先级，譬如这里代表重复“*”的范围。于是我们可以重新组织文法：

**FACTOR = NUMBER**
**FACTOR = '-' FACTOR**
**FACTOR = '(' EXP ')'**
**FACTOR = ID '(' EXP ')'**
**TERM = FACTOR (MUL FACTOR)***
**EXP = TERM (ADD TERM)***

# 语法类型与 C++表达

Vczh Library++允许我们直接把语法在 C++的框架下表达出来，因此我们不得不对语法的表达形式做一点修改使之可以满足 C++的要求。所以这里我们需要做两件事情，第一件事情是规则的类型，第二件事情是如何用 C++语句来表达规则。

规则的类型含义比较复杂，一个规则的类型不值取决于它自身，还取决于它的产出。如果我们用语法规则来将记号直接转换成计算结果，那么一般来说规则的类型就是计算结果的类型，譬如说数字。如果我们用语法规则来讲记号转换成四则运算式子的语法树，那么规则的类型就是语法树节点的指针。

如果我们把规则看成一个函数的话，那应该会更加容易理解。一个语法规则将输入的记号列表转换成我们需要的结果，所以规则的类型至少包含两个部分，一个是输入记号的类型，一个是输出类型。Vczh Library++专门为规则定义了一个模板类，而且这里 FACTOR、TERM 和 EXP 将会作为 C++的变量直接声明出来。在这里我们希望语法规则能直接将输入转换成计算结果，结果的类型是 double，输入的类型是 RegexToken，因此我们可以这么声明三个规则的名字：

```
Rule<TokenInput<RegexToken>, double> factor, term, exp;
```

TokenInput 是输入的其中一种表达形式，它可以将一个指针和长度转换成符合 Rule 输入的类型。Vczh Library++还同时提供了 StringInput<T>和 EnumerableInput<T>，但是我们已经将记号保存在 List<RegexToken>里面了，因此使用 TokenInput<T>是最合适的。

StringInput<T>也好，EnumerableInput<T>也好，TokenInput<T>也好，其实都是一个迭代器。Vczh Library++的语法分析器类库为迭代器规定了一个接口，这三种迭代器都是在那个接口的框架下实现的。我们可以简单的把一个把 TokenInput<RegexToken>套在 List<RegexToken>上：

```
TokenInput<RegexToken> input(&tokens[0], tokens.Count());
```

TokenInput 在内部只保存了一个指针、长度和当前位置，所以是一个相当轻量级的类，可以到处复制并且不会有多少性能上的损失。不过 TokenInput<T>的生命周期不应该比 List<T>长，不然 TokenInput<T>指向的对象会因为已经被释放掉而发生问题。同样的道理，在 TokenInput<T>已经被套在 List<T>上的时候，List<T>最好不要被修改。

现在输入的类型已经清楚了，可以开始研究输出的类型了。上面的 factor 的声明是 Rule<TokenInput<RegexToken>, double>，因此 factor 可以看成是一个输入迭代器 TokenInput<RegexToken>，修改迭代器位置并输出 double 作为结果的函数。不过其实返回的实际类型是 ParsingResult<double>，因为一个规则在分析一个迭代器输入的时候可能会产生

错误，这个时候不能修改输入迭代器的位置，而且还要返回错误的标志。因此这里使用 ParsingResult<double>，它能告诉你成功还是失败，而且成功的话会带有一个真正的 double 类型的返回值，并且修改迭代器的位置，让它指向跳过一个 factor 后的位置以便继续分析。

规则有许多种组合方法。假设有规则：

Rule<I, A> a, a2;

Rule<I, B> b;

那么可以组合出以下各种新的规则：

a+b：类型 Rule<I, ParsingPair<A, B>>，代表 a 和 b 应该按顺序出现。

*a：类型 Rule<I, ParsingList<A>>，代表 a 应该连续出现 0 或多次。

+a：类型 Rule<I, ParsingList<A>>，代表 a 应该连续出现 1 或多次。

a|a2：类型 Rule<I, A>，代表要么是 a，要么是 a2。这里 a 和 a2 类型应该一致。

a>>b：类型 Rule<I, B>，代表 a 和 b 应该按顺序出现，并且抛弃 a 只保留 b 的结果。

a<<b：类型 Rule<I, A>，代表 a 和 b 应该按顺序出现，并且抛弃 b 只保留 a 的结果。

opt(a)：类型 Rule<I, A>，代表 a 应该出现 0 或 1 次。

还有另外两种组合方法，分别用于转换分析结果和进行错误恢复。在这里先介绍转换分析结果的组合方法。下面举 EXP 的例子：

**EXP = TERM (ADD TERM)***

写成 C++应该是：

**EXP = TERM + *(tk(ADD) + TERM);**

这里 ADD 的类型是 const int，因此我们需要一个函数把它转换成一个规则。这里使用 tk 函数。tk 函数将一个 int 转换成 Rule<TokenInput<RegexToken>, RegexToken>，勇于匹配一个输入是不是 ADD 类型的记号。于是我们可以慢慢解开这个规则的最终类型。这里我们不关心输入类型，只关心输出类型，因为所有的规则的类型都是 Rule<TokenInput<RegexToken>, T>。根据上文，我们知道 TERM 与 EXP 的类型一样，都是返回 double。

tk(ADD) ：T == RegexToken

tk(ADD) + TERM ：T == ParsingPair<RegexToken, double>

*(tk(ADD) + TERM) ：T == ParsingList<ParsingPair<RegexToken, double>>

TERM + *(tk(ADD) + TERM) ：T == ParsingPair<double, ParsingList<ParsingPair<RegexToken, double>>>

这里问题就来了，EXP 的类型跟 TERM + *(tk(ADD) + TERM)类型不一样，那必然需要一个函数来帮我们做转换。假如我们已经有了一个函数：

double Operator(const ParsingPair<double, ParsingList<ParsingPair<RegexToken, double>>>& input)

这个函数勇于将输入的那一大串东西，经过计算最终转换成一个 double 类型的结果，那么我们就可以使用这个 Operator 函数最终将 EXP 和 TERM + *(tk(ADD) + TERM)连起来：

**EXP = (TERM + *(tk(ADD) + TERM))[Operator];**

ParsingPair<double, ParsingList<ParsingPair<RegexToken, double>>>的内容实际上是一个操作数，加上一个操作符连着操作数的列表。于是当我们真的需要把它转成一个 double 的时候，就要去遍历所有“操作符连着操作数”的列表，最后将计算结果全部累加到第一个操作数身上。记得我们之前表达 EXP 的方法跟现在不一样吗？以前是

**EXP = TERM | EXP ADD TERM**

因为 Vczh Library++无法处理左递归，才需要我们手动拆解成

**EXP = TERM (ADD TERM)***

于是为了让我们处理起来更简单，Vczh Library++提供了一个 lrec 函数，让我们可以享受左递归带来的方便。

lrec 把类型 ParsingPair<T, ParsingList<U>>通过一个函数 T(const T&, const U&)转换成 T。这就意味着一个输入 T U U U … ，加上一个把一个 T 跟 U 加起来变成 T 的函数，最终把整个序列处理成 T：

T U U U

=> T U U

=> T U

=> T

如果把他们套到我们的 EXP 上面，就可以做下面的计算

TERM (ADD TERM) (ADD TERM) (ADD TERM)　　**1+2+3+4**

=> TERM (ADD TERM) (ADD TERM)　　**3+3+4**

=> TERM (ADD TERM)　　**6+4**

=> TERM　　**10**

这个转换函数跟处理 EXP ADD TERM 是一样的！

因此，只要有了 lrec 函数，我们可以把

**EXP = TERM | (EXP + tk(ADD) + TERM) [F1]**

这种 Vczh Library++不支持的左递归语法表示处理成

**EXP = lrec(TERM + *(tk(ADD) + TERM), F2)**

其中 F1 的类型是 double (const ParsingPair<ParsingPair<double, RegexToken>, double>&)

而 F2 的类型是 double (const double&, const ParsingPair<RegexToken, double>&)

**我们不会因为需要拆解左递归而带来任何不便！**

# 实现

现在开始进入激动人心的时刻了，我们可以借助 Vczh Library++来实现一个带函数四则运算式子的计算器了。现在回顾一下我们的语法：

**FACTOR = NUMBER**

**FACTOR = '-' FACTOR**

**FACTOR = '(' EXP ')'**

**FACTOR = ID '(' EXP ')'**

**TERM = FACTOR (MUL FACTOR)***

**EXP = TERM (ADD TERM)***

把它转换成 C++就应该是：

```
Rule<TokenInput<RegexToken>, double> factor, term, exp;
factor    = tk(NUMBER) [Convert]
          | (tk(L"-") >> factor)[Negative]
          | (tk(L"(") >> exp << tk(L")"))
```

```
            | (tk(ID) + (tk(L"(") >> exp << tk(L")")))[Call]
            ;
term        = lrec(factor + *(tk(MUL) + factor), Operator);
exp         = lrec(term + *(tk(ADD) + term), Operator);
```

让我们来逐个阅读规则，并分析出结果转换函数 Convert、Negative、Call 和 Operator 的类型。

第一个是 tk(NUMBER)[Convert]。这个规则将一个数字记号转换为一个真正的数字。因为 tk(NUMBER)的类型是 RegexToken，因此 Convert 的类型是 **double (const RegexToken&)**。

第二个是(tk(L"-") >> factor)[Negative]。tk(L"-")的类型是 RegexToken，factor 的类型是 double，所以 RegexToken>>double 其实就是 double。因此 Negative 的类型是 **double (const double&)**。

第三个是 tk(L"(") >> exp << tk(L")")。实际上分析了两个括号和 exp 之后，括号被丢掉了，剩下 exp 的类型是 double。因此这一行规则不需要任何转换函数。

第四个是(tk(ID) + (tk(L"(") >> exp << tk(L")")))[Call]，我们很容易知道 Call 的类型是 **double(const ParsingPair<RegexToken, double>&)**。

最后一个是 Operator，这个之前已经讨论过了，类型是 **double (const double&, const ParsingPair<RegexToken, double>&)**。

知道了这个之后，我们就可以实现这些函数了：

```
double Convert(const RegexToken& input)
{
    return wtof(WString(input.reading, input.length));
}

double Negative(const double& input)
{
    return -input;
}

double Operator(const double& left, const ParsingPair<RegexToken, double>& right)
{
    switch(*right.First().reading)
    {
    case L'+':
        return left+right.Second();
    case L'-':
        return left-right.Second();
    case L'*':
```

```
            return left*right.Second();
        case L'/':
            return left/right.Second();
        default:
            return 0;
        }
}

double Call(const ParsingPair<RegexToken, double>& input)
{
    WString name(input.First().reading, input.First().length);
    double parameter=input.Second();

    if(name==L"sin")
    {
        return sin(parameter);
    }
    else if(name==L"cos")
    {
        return cos(parameter);
    }
    else if(name==L"tan")
    {
        return tan(parameter);
    }
    else if(name==L"cot")
    {
        return 1/tan(parameter);
    }
    else if(name==L"sec")
    {
        return 1/cos(parameter);
    }
    else if(name==L"csc")
    {
        return 1/sin(parameter);
    }
    else if(name==L"exp")
    {
        return exp(parameter);
    }
    else if(name==L"ln")
    {
        return log(parameter);
```

```
    }
    else if(name==L"abs")
    {
        return abs(parameter);
    }
    else if(name==L"sqrt")
    {
        return sqrt(parameter);
    }
    else if(name==L"sqr")
    {
        return parameter*parameter;
    }
    else
    {
        throw Exception(L"Function "+name+L" not exists.");
    }
}
```

然后我们就可以用这些函数来构造一个语法分析器了：

```
List<WString> patterns;
const int BLANK         = patterns.Add(L"/s+");
const int ADD           = patterns.Add(L"/+|-");
const int MUL           = patterns.Add(L"/*|//");
const int NUMBER        = patterns.Add(L"/d+(./d+)?");
const int ID            = patterns.Add(L"[a-zA-Z_]/w*");
const int OPEN          = patterns.Add(L"/(");
const int CLOSE         = patterns.Add(L"/)");
RegexLexer lexer(patterns.Wrap());

Rule<TokenInput<RegexToken>, double> factor, term, exp;

factor    = tk(NUMBER)[Convert]
          | (tk(L"-") >> factor)[Negative]
          | (tk(L"(") >> exp << tk(L")"))
          | (tk(ID) + (tk(L"(") >> exp << tk(L")")))[Call]
          ;
term      = lrec(factor + *(tk(MUL) + factor), Operator);
exp       = lrec(term + *(tk(ADD) + term), Operator);

WString line=Console::Read();
List<RegexToken> tokens;
CopyFrom(tokens.Wrap(), lexer.Parse(line)>>Where(IsNotBlank));
TokenInput<RegexToken> input(&tokens[0], tokens.Count());
```

```
double result=exp.ParseFull(input, false);
Console::WriteLine(L"Result is "+ftow(result));
```

是不是很容易写出来呢？不仅 exp 可以用来做分析，其实任何的 Rule<I, T>都有一个 ParseFull 函数用来分析输入的记号列表。

# 错误恢复和定制错误信息

Vczh Library++对语法分析提供了强大的错误处理的支持。我们可以自由定制在语法规则的任意一点发生错误的时候应该采取的处理方法。我们可以

- 记录一个错误并控制错误信息的文字内容
- 决定恢复或者不恢复（构造一个假的分析结果）
- 为了恢复错误，决定当前的迭代器应该跳过多少个记号

还是以那个例子为基础，对于 tk(NUMBER)[Convert]，如果我们想在输入的迭代器所指向的位置不是一个数字的时候，想让分析立刻失败（分析器会自动尝试接下来的三个同一等级的规则，如果都失败，那么会采用这里的分析结果），那么可以将系统为这个错误自动生成的错误信息清除并使用我们自己的信息，然后返回一个值告诉系统说我不仅要自己定制错误信息，而且还不准备恢复：

```
ParsingResult<RegexToken> NeedExpression(TokenInput<RegexToken>& input,
Types<TokenInput<RegexToken>>::GlobalInfo& globalInfo)
    {
        globalInfo.errors.Clear();
        globalInfo.errors.Add(new CombinatorError<TokenInput<RegexToken>>(L"Here needs an
expression.", input));
        return ParsingResult<RegexToken>();
    }
```

于是我们可以在这个地方使用这个错误处理函数：

```
tk(NUMBER)(NeedExpression)[Convert]
```

Vczh Library++使用中括号插入结果转换函数，用小括号插入错误处理函数。因此我们可以挑选所有需要定制错误的地方，写出这些函数然后应用在规则上：

```
ParsingResult<RegexToken> NeedOpenBrace(TokenInput<RegexToken>& input,
Types<TokenInput<RegexToken>>::GlobalInfo& globalInfo)
    {
        globalInfo.errors.Clear();
        globalInfo.errors.Add(new CombinatorError<TokenInput<RegexToken>>(L"Here needs a
\"(\".", input));
        return ParsingResult<RegexToken>();
    }

ParsingResult<RegexToken> NeedCloseBrace(TokenInput<RegexToken>& input,
Types<TokenInput<RegexToken>>::GlobalInfo& globalInfo)
    {
```

```
        globalInfo.errors.Clear();
        globalInfo.errors.Add(new CombinatorError<TokenInput<RegexToken>>(L"Here needs an \")\".", input));
        return ParsingResult<RegexToken>();
    }

    ParsingResult<RegexToken> NeedOperator(TokenInput<RegexToken>& input, Types<TokenInput<RegexToken>>::GlobalInfo& globalInfo)
    {
        globalInfo.errors.Clear();
        globalInfo.errors.Add(new CombinatorError<TokenInput<RegexToken>>(L"Here needs an operator.", input));
        return ParsingResult<RegexToken>();
    }

    factor    = tk(NUMBER)(NeedExpression)[Convert]
              | (tk(L"-") >> factor)[Negative]
              | (tk(L"(") >> exp << tk(L")")(NeedCloseBrace))
              | (tk(ID) + (tk(L"(")(NeedOpenBrace) >> exp << tk(L")")(NeedCloseBrace)))[Call]
              ;
    term      = lrec(factor + *(tk(MUL)(NeedOperator) + factor), Operator);
    exp       = lrec(term + *(tk(ADD)(NeedOperator) + term), Operator);
```

并不是所有的地方都需要我们亲自处理错误，我们只需要在需要自己定制错误消息的地方写上错误处理函数就好了。我们有一些简单的原则来寻找需要处理错误的地方。

首先，一个规则的非第一分支的第一个记号不需要处理错误。这个很好处理，我们看 factor，一共有四个分支。首先 tk(NUMBER)是第一分支的第一个记号，而 tk(L"-")、tk(L"(") 和 tk(ID)是非第一分支的的第一个记号。因为只要第一个分支处理了错误，那么非第一分支全部在第一个记号就失败的话，那么结果显然是采取第一个分支的错误结果。

第二，大部分错误都集中在**记号规则**上。记号规则说的是 tk 函数产生的规则。因为绝大多数错误信息都是在描述“这里需要 XXX 但是却没出现”，因此只需要在第一个原则所说的不需要错误信息的地方以外的所有记号规则出现的地方都写上自己的错误处理就可以了。

第三，因为第一和第二个原则，因此所有非记号规则能产生的所有错误都被我们定制过了，因此非记号规则不需要任何错误处理，**除非我们想定制能提供更多有用信息的错误信息，或者执行我们自己的错误恢复以便尽可能在错误产生的时候继续分析并产生多条有用的错误信息**。

因此根据这三条原则，再加上我们这个例子只需要第一个错误信息，因此选中了那 6 个标记了红色的地方进行错误处理并输出我们自己的错误信息。

# 捕捉错误

最后的问题就是如何捕捉错误了。每一个 Rule<I, T>都提供了一个 Parse 函数和 ParseFull 函数。Parse 函数用于在输入的迭代器中寻找一个满足语法要求的最长前缀或者在遇到错误的时候给出有意义的错误列表。ParseFull 则假定迭代器中的完整内容满足语法要求，然后进行分析或者在遇到错误的时候给出有意义的错误列表。

Vczh Library++内部有一套用于将所有用户自定义的错误恢复机制所产生的错有可恢复错误挑选并组合起来的算法。因此在捕捉到错误的时候，第一个错误总是处于一个尽可能元的位置，而且基本上都是有意义的。Parse 和 ParseFull 函数都直接返回我们需要的分析结果，或者在遇到错误的时候抛出一个 CombinatorException<I>类型的异常。

Parse 和 ParseFull 的参数和结果如下：

```
template<typename I, typename O>
class Rule
{
    O Parse(const I& input, bool allowError, I* remain=0)const;
    O ParseFull(const I& input, bool allowError)const;
};
```

input 参数是输入的迭代器。一般来说输入的迭代器的当前位置是第一个记号的位置，当然你也可以自己读了几个记号之后再传给 Parse。 allowError 为 true 的时候，如果分析出了错误但是所有错误都被用户自定义的错误恢复函数恢复了，也会返回分析结果而不会抛出异常。allowError 为 false 的时候，只要有错误出现就会抛出异常。remain 参数仅在 Parse 函数中有用，在分析结束之后，如果传入的指针不是空，那么对象会被修改为分析结束后迭代器的状态。

如果分析出现错误并且需要被处理的话，那么 Parse 和 ParseFUll 都会抛出一个 CombinatorException<I>的异常。CombinatorException<I>的定义如下：

```
template<typename I>
class CombinatorException : public Exception
{
    const I& GetInput()const;
    const typename Types<I>::GlobalInfo& GetGlobalInfo()const;
};
```

GetInput 返回迭代器的当前状态。在所有错误都被恢复的时候，迭代器的当前状态是分析结束的时候迭代器的位置。一旦出现了没有被恢复的错误，那么迭代器的当前状态是 Parse 或者 ParseFull 输入的迭代器状态。GetGlobalInfo 返回的对象有 errorList 与 candidateErrorList 两个列表，分别是错误和备选错误。他们的元素类型都是 Ptr<CombinatorError<I>>。

CombinatorError<I>的定义如下：

```
template<typename I>
class CombinatorError : public Exception
{
```

```
public:
    typedef typename Types<I>::Input          InputType;

    const InputType& GetPosition();
};
```

而 Exception 的定义如下：

```
class Exception : public Object
{
public:
    const WString& Message()const;
};
```

我们可以通过 Message()函数获得错误信息的文字内容，然后通过 GetPosition()函数获得错误发生的时候迭代器的状态。于是我们不仅可以知道出现了多少错误，还能知道这些错误时分别在什么地方出现的。

于是让我们来看一看带函数的四则运算计算器应该如何处理用户输入的表达式在分析过程中产生的错误：

```
Console::Write(L"\r\nexpression>");
WString line=Console::Read();
if(line==L"")
{
    break;
}

try
{
    List<RegexToken> tokens;
    CopyFrom(tokens.Wrap(), lexer.Parse(line)>>Where(IsNotBlank));
    for(int i=0;i<tokens.Count();i++)
    {
        if(tokens[i].token==-1)
        {
            throw Exception(L"Syntax error. Unknown token: \""+WString(tokens[i].reading,
tokens[i].length)+L"\".");
        }
    }
    if(tokens.Count()==0)
    {
        throw Exception(L"Syntax error. Expression cannot be empty.");
    }
    try
    {
        TokenInput<RegexToken> input(&tokens[0], tokens.Count());
        double result=exp.ParseFull(input, false);
```

```
            Console::WriteLine(L"Result is "+ftow(result));
        }
        catch(const CombinatorException<TokenInput<RegexToken>>& e)
        {
            Ptr<CombinatorError<TokenInput<RegexToken>>>
error=e.GetGlobalInfo().errors.Get(0);
            const TokenInput<RegexToken>& position=error->GetPosition();
            if(position.Available())
            {
                throw Exception(L"Syntax error. "+error->Message()+L" First occurs at
\""+WString(position.Current().reading)+L"\".");
            }
            else
            {
                throw Exception(L"Syntax error. Expression is not complete.");
            }
        }
    }
    catch(const Exception& e)
    {
        Console::SetColor(true, false, false, true);
        Console::WriteLine(e.Message());
        Console::SetColor(true, true, true, false);
    }
```

# 结束

使用 Vczh Library++开发语法分析器的指南就到此结束了。如果在阅读过程中有什么疑问的话可以使用如下方法来找到我：

电子邮件：vczh@163.com

博客：http://www.cppblog.com/vczh

Vczh Library++项目主页：http://vlpp.codeplex.com