Chapter 6

関数型プログラミング

6.1 関数型プログラミングとは？

ものごとの見方や考え方の枠組みという意味の「パラダイム」という言葉があります。プログラミングの世界にも、プログラムってものをどう捉えるのか、どう設計するのか、という考え方…つまりパラダイムが、たくさん提唱されてきました。

例えば「構造化プログラミング」。プログラムはまとまった処理ごとにいくつかの単位に分けて、メインの処理ではそれらサブルーチンを呼び出す形で記述する方法です。あるいは、「オブジェクト指向」。これは、状態と振る舞いがセットになった「オブジェクト」同士の相互作用としてプログラムというものを表現しよう、というもの。「ジェネリックプログラミング」なるパラダイムもありました。アルゴリズムの、扱うデータの種類に依存しない共通の構造に着目することで、非常に再利用性の高いプログラムを実現しようという考え方です。

さて、そんな中の一つに、「関数型プログラミング」と呼ばれるパラダイムがあります。近年、Haskell や Objective Caml など強力な関数型言語の人気が高まってくるにつれて、特に注目を浴びています。関数型プログラミングの特徴は、主に次の二点。

- 純粋な関数（入力を受け取って出力を返す以外のことは何もしない関数）の組み合わせのみでプログラムを記述する。
- 関数そのものを値として扱う。つまり、高階関数（関数を引数や返値とする関数）を積極的に利用する。

前者は、C++ とは決定的に異なる特徴です。例えば純粋な関数型言語には、変数への代入が存在しません。

```
x = 4
```

この C++ の式の入力は、変数 x と値 4 です。出力、つまり式の値は 4 です。ところがこの式は式の値 4 を返す計算の他に、変数 x の持つ値を 4 に書き換えるという副作用を伴ってしまいます。関数型パラダイムでの変数は、このようには使えず、単に式や値に名前をつけるという意味だけを持っています。


```
let x = 3.0 + 0.14 in   -- この式に名前をつけておいて、
  x * x                 -- あとで使うことはできる。
                        -- でも、変数 x の値を書き換える構文はない。
```

▲Src. 6-1　関数型言語 Haskell のプログラム例

またこれに伴って、for ループや while ループがなくなります。なにせ副作用が起こせないので、ループ変数を書き換えることも、終了条件になっている変数をいじることも許されないのです。

```
int sum = 0;
for(int i=0; i<=100; ++i)
  sum += i;
```

▲Src. 6-2　0 から 100 の和を計算する C++ プログラム

さあ困った…と思いきや、これは関数の再帰呼び出しを使って副作用なしで綺麗に記述できます。実際、全てのループは再帰関数に変換できてしまいます。

```
let sumf 0 = 0                    -- 関数 sumf の定義。0 を入れると 0 を返し、
    sumf n = n + sumf (n-1) in    -- 他の数 n なら自分を再帰的に呼んで計算
  sumf 100
```

▲Src. 6-3　0 から 100 の和を計算する Haskell プログラム

と、ここまでは関数型プログラミングの制限ばっかりを挙げてきました。しかしもちろん、敢えてこんな制限を課すからには理由があります。具体的には、次のような利点が得られると言われています。

- 一度定めた変数の値は決して変化しないので、定義さえ見ればその変数が把握できる。つまりプログラムの見通しがよくなって、バグが減る。（C++ でも、出来るだけ const を活用しようとよく言われていますね。）
- 副作用がないので、プログラムの実行順序の自由度が高い。つまり、コンパイラによる実行順序の最適化が期待できる。

特に強烈なのが、後者です。「いつ計算してもいいんだから、本当に値が必要になるギリギリまで式の実行を遅らせよう」という遅延評価方式を採用した言語なら、例えば要素が無限に続くリストなんてものを非常に簡潔に記述することができます。しかも実行が巧みに遅延されるため、「その無限リストの全要素に関数を適用する」という式を、無限ループに陥ることなく扱うことができてしまいます。

```
> let lst = 1:[2,3,4] in lst
          -- ':' はリストの先頭に要素をつなげる普通の演算子
[1,2,3,4]
> let lst = [1,2,3,4] in map (* 2) lst
          -- map関数は、全要素に関数を適用してリストを変換する
          -- std::transform みたいなものです
[2,4,6,8]
> let lst = 1 : map (+ 1) lst in map (* 2) lst
          -- 1ずつ増えていく無限リストの全要素を2倍したリスト
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,
50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86,88,90,92,94,
96,98,100,102,104,106,108,110,112,114,116,118,120,122,124,126,128,13
0,132,134,136,138,140,142,144,146,148,150,152,...
```

▲Src. 6-4　Haskellによる無限リストの例

この節の頭であげた関数型プログラミングの特徴はもう一つありました。それは「関数そのものを値として扱う」こと。これは実は、C++標準ライブラリのSTLにも頻繁に現れてくる考え方です。

```
bool bigger_first( int x, int y ) { return x > y; }
...
vector<int> v;
std::sort( v.begin(), v.end(), &bigger_first );
```

▲Src. 6-5　C++での高階関数の例

std::sort関数は、要素と要素の比較をする関数を受け取って、処理をしています。このように、関数をただ呼び出すだけではなく、他の関数へ渡したり戻したりできる値と捉えることで、STLのような汎用性の高いコンポーネントが生まれています。

でも、じゃあC++も関数型言語と言えるのかと言うと、それには大きな疑問が残ります。プログラマが定義した関数へのポインタを渡したり返値としたりすることはできますが、プログラム実行中に新しい関数を生成してそれを返す、といった柔軟な記述ができないからです。

例えば、「数と数の大小を比較する関数」と「文字列を数に直す関数」が既に定義されているとき、「文字列を数として比較する関数」が欲しいと思ったら、そういう関数をプログラマが書く必要があります。


```
bool cmp_as_int( string x, string y )
{
    return bigger_first( toInt(x), toInt(y) );
}
```

例えば、「2乗する関数」と「足し算する関数」が既に定義されているとき、「2乗同士を足し算する関数」が欲しいと思ったら、そういう関数をプログラマが書く必要があります。

```
double calc_norm( double x, double y )
{
    return add( square(x), square(y) );
}
```

でもこれって、よく見ると全く同じ形をしています。もし実行中に新しい関数を作ることが出来たなら、こんな風に関数と関数を組み合わせる関数を作ってしまえそう。

```
compose( f, g )
{
    new_function(x,y) { return f(g(x), g(y)); }
    return new_function;
}
calc_norm  = compose( add, square );
cmp_as_int = compose( bigger_first, toInt );
```

▲Src. 6-6　C++でこんな風に書けたら嬉しい擬似コード

```
let compose f g = ¥x y -> f (g x) (g y) in ...
```

▲Src. 6-7　Haskellでの関数合成コード

このようなプログラムが書ける表現力も、関数型プログラミングの特徴となっています。しかしこれはC++では表現できない。残念なことです。

……いや、でも、本当にC++はこれらの機能を実現できないのでしょうか？Boostライブラリは、関数型プログラミングの特徴の一つ、「関数を値として扱う」という考え方を強力にサポートします。

6.2 高階関数

6.2.1 bind

C++の標準ライブラリ「STL」に用意されたアルゴリズムは、パラメータとして関数を取れる設計にすることで、汎用性を高めています。例えば、要素の列から条件を満たすものを探すfind_ifアルゴリズムは、条件を満たすなら真、満たさないなら偽を返す関数をパラメタに渡して、さまざまな種類の検索を実現できます。

```
string str;
string::iterator i;
i = find_if( str.begin(), str.end(), &std::isupper ); // 大文字を探す
i = find_if( str.begin(), str.end(), &std::isdigit ); // 数字を探す
```

また、関数だけでなく「関数オブジェクト」も使うことができます。これはoperator()を実装したオブジェクトで、あたかも本物の関数であるかのように、()を使って呼び出せるものです。関数と違って関数オブジェクトは内部状態を持てるため、検索条件を動的に変化させることも可能でした。

```
template<typename Type>
  struct less_than : public unary_function<Type, bool>
  {
      Type base;
      less_than( const Type& b ) : base(b) {}
      bool operator()( const Type& x ) const { return x < base; }
  };

vector<int> v;
int N = 100;
find( v.begin(), v.end(), less_than<int>(N) ); // Nより小さいものを探す
```

このアプローチは柔軟性が高く素晴らしいのですが、しかし残念なことに、「検索条件用の関数を書くのが面倒くさい」という、しょうもないけれど重大な欠点があります。ちょっとずつ違う条件でvectorから値を探すコードを書くたびに一個ずつ関数や関数オブジェクトを書いていく、というのは何だか遠回り。

そこでBoost.Bindライブラリでは、「既存の関数をベースに新しい関数をその場で合成してしまう」機能を提供します。いや、正確には、新しい「関数オブジェクト」を合成します。早速例をみてみましょう。

```
#include <iostream>
#include <algorithm>
#include <boost/bind.hpp>
using namespace std;
using namespace boost;

bool less_int( int x, int y )
{
    return x < y;
}

int main()
{
    int v[5] = {500,100,50,10,5};
    int N = 100;

    int* i = find_if( v, v+5, bind(&less_int, _1, N) );

    cout << *i << endl;
    return 0;
}
```

▲Src. 6-8　bind1.cpp

実行結果

```
> bind1
50
```

ポイントは、bind()関数です。ここでは、二つの整数を比較するless_intという2引数関数をもとに、Nより小さいかどうか？を返す1引数関数を作ってみました。bindによって、less_intの第二引数にNを縛りつけて固定して、残りの一個しか引数を取らない新しい関数を作ったわけです。つまり、

```
bind( &less_int, _1, N )( x )
```

こう書くのと

```
less_int( x, N )
```

こう書くのが同じ意味になります。_1という変数(これ、かなり変わった名前ですが、れっきとした普通の変数です)は、この位置にあとで第一引数を入れる、とbindに伝える役割を果たしています。

このように、後で値を入れる場所(placeholder)と、引数を特定の値に束縛(bind)、という二つの指定を組み合わせるのがbindの基本パターンです。STLにもstd::bind1stとstd::bind2ndという同じような働きの関数がありますが、これらは2引数関数にしか使うことができません。Boost.Bindでは、もっと幅広い指定にも対応しています。

```
#include <iostream>
#include <boost/bind.hpp>
using namespace std;
using namespace boost;

void testfunc( const char* s1, const char* s2,
               const char* s3, const char* s4 )
{
    cout << s1 << ' ' << s2 << ' '
         << s3 << ' ' << s4 << endl;
}

int main()
{
    // _1, _2 と空けておくと、あとで第一引数と第二引数が入ります
    bind( &testfunc, _1, _2, "three", "four" )( "いち", "に" );
    bind( &testfunc, "one", _1, "three", _2 ) ( "に", "よん" );

    // 関数の引数の順序を入れ替えるのにも使えます
    bind( &testfunc, _4, _3, _2, _1 )( "A", "B", "C", "D" );

    // 一つの引数を増殖させても構いません
    bind( &testfunc, _1, _1, "hoge", _2 )( "FUGA", "PIYO" );

    // _1や_2を使わずに_3を使うと、第一/第二引数は無視されます
    bind( &testfunc, _3, _3, _3, _3 )( "(T_T)", "m(_ _)m", "(^^)" );

    return 0;
}
```

▲Src. 6-9 bindの動作(bind2.cpp)

実行結果

```
> bind2
いち に three four
one に three よん
D C B A
FUGA FUGA hoge PIYO
(^^) (^^) (^^) (^^)
```

bindへメンバ関数へのポインタを渡すと、自動的に、thisポインタになる変数を第一引数に取る関数と見なしてbind処理がおこなわれます。これは例えば、コンテナに入ってるオブジェクト全部のメンバ関数を一気に呼ぶようなコードを書くときの関数オブジェクトとして便利です。

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <boost/bind.hpp>
using namespace std;
using namespace boost;

int main()
{
    vector<string> v;
    v.push_back( "abcde" );
    v.push_back( "fghijklmn" );
    v.push_back( "" );
    v.push_back( "opqr" );
    v.push_back( "stuvwxyz" );

    // 空文字列を取り除く
    //   bind(&string::empty, _1)( s ) == s.empty()
    vector<string>::iterator end =
        remove_if( v.begin(), v.end(), bind(&string::empty, _1) );

    // 各文字列の、[2]から2文字ずつ取り出す変換
    //   bind(&string::substr, _1, 2, 2)( s ) == s.substr(2,2)
    transform(
        v.begin(), end, v.begin(), bind(&string::substr, _1, 2, 2) );

    cout << v[0] << ' ' << v[1] << ' ' << v[2] << ' ' << v[3] << endl;
```

```
    return 0;
}
```

▲Src. 6-10　bind3.cpp

実行結果

```
> bind3
cd hi qr uv
```

この機能は、STLのstd::mem_funやstd::mem_fun_refの拡張になっています。メンバ「関数」だけでなくメンバ「変数」も、thisポインタとなるオブジェクトを受け取る1引数関数と解釈して、bindの対象にすることができます。

```
struct Point { int x, y; };
Point pt = {3, 4}:
assert( bind(&Point::x, _1)( pt ) == 3 );
assert( bind(&Point::y, _1)( pt ) == 4 );
```

また、ここまでは一つの関数に値をいくつか束縛する、という使い方を紹介してきましたが、既にある関数から別の関数を作る方法はこれだけではありません。複数の関数を合成して一つにしてしまう、なんてことも考えられます。例えば

- 平方根を求める、double sqrt(double x)
- 足し算をする、double add(double x, double y)
- 掛け算をする、double mult(double x, double y)

の3つの関数が既に存在したとします。すると$\sqrt{x^2+y^2}$を計算する関数は、この3つを組み合わせれば、

- sqrt(add(mult(x,x),mult(y,y)))

と書けます。このように関数を多段に呼び出す合成処理は、bindを多段重ねにすることで、実現されます。

```
#include <iostream>
#include <functional>
#include <cmath>
#include <boost/bind.hpp>
using namespace std;
using namespace boost;
```



```
int main()
{
       double x = 3.0, y = 4.0;

    // bind内に更にbindを書くと、二つの関数の合成として
    // 処理されます。この例では、sqrtと足し算と掛け算を
    // 合成して、sqrt( x*x + y*y ) を計算しています。
    //
      // plus<double>()やmultiplies<double>()はSTLの
      // 関数オブジェクト。sqrtは標準ライブラリのsqrt関数です
      cout <<
              bind( &sqrt,
                bind( plus<double>(),
                  bind( multiplies<double>(), _1, _1 ),
                  bind( multiplies<double>(), _2, _2 )
              )( x, y )
      << endl;

    return 0;
}
```

▲Src. 6-11　bind4.cpp

これを用いると、例えば文字列のvectorを長さの短い順にソートする処理が、新しい比較関数を自分で定義することなく、こんな風に記述できます。

```
vector<string> v;

// f(s1,s2) == s1.length() < s2.length() となるような
// 関数fを合成して、sortのパラメタとして渡しています
sort( v.begin(), v.end(),
    bind(less<size_t>(), bind(&string::length,_1),
                         bind(&string::length,_2)) );
```

6.2.2 リファレンス

bind関数

必要なヘッダファイル	#include <boost/bind.hpp>
名前空間	boost

```
template<class R>
  unspecified bind( R (*f)() );
template<class R, class B1, class A1>
  unspecified bind( R (*f)(B1), A1 a1 );
template<class R, class B1, class B2, class A1, class A2>
  unspecified bind( R (*f)(B1, B2), A1 a1, A2 a2 );
```

グローバル関数や静的メンバ関数へのポインタを受け取り、引数を束縛します。仕様では3引数以上の関数を扱うバージョンはかならずしも提供されないとなっていますが、version 1.31時点では、全ての環境で9引数版まで利用可能なようです。

```
template<class F>
  unspecified bind( F f );
template<class F, class A1>
  unspecified bind( F f, A1 a1 );
template<class F, class A1, class A2>
  unspecified bind( F f, A1 a1, A2 a2 );

template<class R, class F>
  unspecified bind( F f );
template<class R, class F, class A1>
  unspecified bind( F f, A1 a1 );
template<class R, class F, class A1, class A2>
  unspecified bind( F f, A1 a1, A2 a2 );
```

関数オブジェクト(operator()の定義されたオブジェクト)へ引数を束縛します。

一般の関数オブジェクトの場合は、bind<int>(f, …)のように返値型をユーザが指定しなければなりません。ただし、関数オブジェクト内にresult_type型が定義されていれば、この指定は省略することができます。C++標準ライブラリやBoostで定義される関数オブジェクトには、基本的にresult_typeが定義されています。9引数版まで定義があります。

```
template<class R, class T, class A1>
  unspecified bind( R (T::*f)(), A1 a1 );
template<class R, class T, class A1>
  unspecified bind( R (T::*f)() const, A1 a1 );
template<class R, class T, class B1, class A1, class A2>
  unspecified bind( R (T::*f)(B1), A1 a1, A2 a2 );
template<class R, class T, class B1, class A1, class A2>
  unspecified bind( R (T::*f)(B1) const, A1 a1, A2 a2 );
```

メンバ関数へのポインタを、明示的に第一引数にthisポインタを取る関数とみなして引数を束縛します。thisの分を含めて9引数版まで定義があります。

```
template<class R, class T, class A1>
  unspecified bind( R T::*f, A1 a1 );
```

メンバ変数へのポインタを、thisポインタを引数として取る関数と見なして、その他の関数と同じように引数を束縛します。

bind関数の返す関数オブジェクト

bind関数の返値は、それ自体、何らかの引数を受け取って値を返す関数オブジェクトとなっています。

```
BF = bind( f, p1, p2, ..., pN );
BF( arg1, arg2, ..., argM );
```

このBFの呼び出し結果は、以下のような疑似プログラムで定義されています。まず、bind関数に渡された疑似引数指定と実際にBFに渡された引数の情報を合わせて、最終的に使うパラメータを計算します。

```
real_arg1 = calc_arg( p1, arg1, …, argM );
real_arg2 = calc_arg( p2, arg1, …, argM );
real_argN = calc_arg( pN, arg1, …, argM );
```

ただし、calc_arg(p, arg1, ..., argM)は次の値になります。

- pが_1のとき、arg1
- pが_2のとき、arg2
- pがboost::ref型もしくはboost::cref型(Chapter8参照)のオブジェクトの時、p.get()。これによって参照渡しを実現します
- pが別のbind関数の返値のとき、p(arg1, …, argM)
- それ以外の場合、pそのもの

次に、このreal_argたちを使って関数を呼び出します。この結果が、

```
BF( arg1, arg2, ..., argM )
```

という式の値となります。real_argの使い方は、次のようになっています。

fが普通の関数へのポインタや関数オブジェクトの時、

```
f( real_arg1, …, real_argN );
```

fがクラスメンバへのポインタの時、real_arg1がそのクラス型の参照なら

```
(real_arg1.*f)                    // メンバ変数へのポインタの場合
(real_arg1.*f)( real_arg2, … ) // メンバ関数へのポインタの場合
```

real_arg1がそのクラス型のポインタなら、

```
(real_arg1->*f)                   // メンバ変数へのポインタの場合
(real_arg1->*f)( real_arg2, … ) // メンバ関数へのポインタの場合
```

real_arg1がそのクラス型へのスマートポインタ(Chapter1参照)なら、

```
(get_pointer(real_arg1)->*f)      // メンバ変数へのポインタの場合
(get_pointer(real_arg1)->*f)( real_arg2, … )
                                  // メンバ関数へのポインタの場合
```


placeholder

必要なヘッダファイル	#include <boost/bind.hpp>
名前空間	boost

```
unspecified_placeholder_type_1  _1;
unspecified_placeholder_type_2  _2;
unspecified_placeholder_type_3  _3;
unspecified_placeholder_type_4  _4;
unspecified_placeholder_type_5  _5;
unspecified_placeholder_type_6  _6;
unspecified_placeholder_type_7  _7;
unspecified_placeholder_type_8  _8;
unspecified_placeholder_type_9  _9;
```

bind関数に渡したときに「あとで引数が入る場所」として扱われる、Boost.Bindライブラリにとって特殊な型の変数です。_1を渡すとそこは第一引数が入る場所になり、_2を渡すとそこは第二引数が入る場所になります。

環境によっては_1, _2, _3までしか提供されない可能性があるとされていますが、version 1.31の時点では、全ての環境で_1から_9までが使用できるようです。

この_1, _2, ...は、同じ名前がBoost.LambdaやBoost.MPLでも使われていますが、それぞれ別物です。混用しないようご注意下さい。

protect関数

必要なヘッダファイル	#include <boost/bind/protect.hpp>
名前空間	boost

```
unspecified protect( bind_type );
```

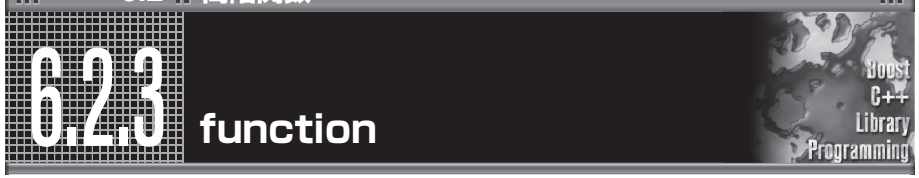
bindの結果を別のbindに渡すと関数の合成として処理されてしまいますが、この動作を禁止するためのサポート関数です。protect(bind(...))をbindに渡すと、一つ目のbindの結果作られた関数オブジェクトを、値として二つ目のbindに渡すという意味になります。

apply<RetType>クラステンプレート

必要なヘッダファイル	#include <boost/bind/apply.hpp>
名前空間	boost

```
template<typename RetType>
  class apply;
```

「関数適用」を行う関数オブジェクトです。apply<R>()(f, x)は、f(x)と同じ動作をします。



前節で紹介したbind関数では、新しく作り出した関数オブジェクトは、すぐにアルゴリズム関数へ渡して使い捨てていました。これを何か他の変数に入れて保持しておくことはできないでしょうか？変数として持つことができれば、後で何回も呼び出す必要がある場合…例えばイベントのコールバック関数として利用するなど、用途が広がります。

従来は、関数を変数に保存するには、関数ポインタを使いました。しかしこれは、bindの返値には使うことができません。bindが返すのは、あたかも関数のように振る舞うオブジェクトではありますが、本当の関数ではないからです。

```
// 失敗例。bindの返値は関数ポインタではなく、オブジェクト
typedef bool (*compare_func_type)(const string&, const string&)
compare_func_type f = bind( less<size_t>(),
                            bind(&string::length,_1),
                            bind(&string::length,_2) );
```

▲Src. 6-12 失敗例

では、bindの返値の型を調べてその型の変数をつくる、という案はどうでしょう。これもうまくありません。というのは、実際に上記のbindの返値型を調べてみると、こんな型であることが判明するのです。

```
boost::_bi::bind_t<boost::_bi::unspecified,std::less<size_t>,boost::
_bi::list2<boost::_bi::bind_t<size_t,boost::_mfi::cmf0<size_t,string
>,boost::_bi::list1<boost::arg<1> > >,boost::_bi::bind_t<size_t,boos
t::_mfi::cmf0<size_t,string>,boost::_bi::list1<boost::arg<2> > > > >
```

引数の情報を全て効率的に保持するために非常に複雑な型になっているわけですが……こんな型の名前をソースコードに書いて変数を宣言するのは、ちょっぴり気が引けますね。しかももっと悪いことに、少しでも違う組み合わせでbindを書くと、たとえ同じ「stringとstringを取ってboolを返す関数」であったとしても、微妙に違う別の型のオブジェクトになってしまいます。これでは、変数を宣言しておいて実行時に色々違う関数オブジェクトを格納して動作を切り替える、というよ

うなことが実現できません。

つまり、関数や関数オブジェクトを変数に持とうと思ったら、実際の細かい型のことは忘れて「stringとstringを取ってboolを返すことができる」ものを何でも保持してくれる変数が必要です。そんな言わば万能関数ポインタ的な働きをするのが、Boostのfunctionクラステンプレートです。

```
#include <iostream>
#include <iomanip>
#include <functional>
#include <string>
#include <boost/bind.hpp>
#include <boost/function.hpp>
using namespace std;
using namespace boost;

bool is_substr_of( const string& sub, const string& all )
{
    // allの中にsubが含まれているか？
    return all.find( sub ) != string::npos;
}

int main()
{
    cout << boolalpha;

    // stringとstringをうけとってboolを返す
    //「関数っぽいもの」は何でも格納できる変数f
    function<bool (const string&, const string&)> f;

    // 関数ポインタを入れて呼び出してみる
    f = &is_substr_of;
    cout << f( "a", "abc" )  << endl;
    cout << f( "a", "bbb" )  << endl;
    cout << f( "ccc", "dd" ) << endl;
    cout << "--------------" << endl;

    // bindの結果関数オブジェクトも入る
    f = bind( less<size_t>(),
            bind(&string::length,_1),
            bind(&string::length,_2) );
    cout << f( "a", "bbb" )  << endl;
```



```
    cout << f( "a", "abc" )  << endl;
    cout << f( "ccc", "dd" ) << endl;

    return 0;
}
```

▲Src. 6-13　function1.cpp

実行結果

```
> function1
true
false
false
--------------
true
true
false
```

Src.6-13の例のように、万能関数オブジェクトの型は、

```
function<返値型 (引数型1, 引数型2, ...)>
```

です。ちょっと変わった形式の型ですが、これは普通の関数の宣言と似せた形になっています。どちらも、先頭に返値の型を書いて、その後ろのかっこの中に引数を書きます。

```
// T1, T2, ..., TN を取ってRを返す関数funの宣言
R fun(T1, T2, ..., TN);

// T1, T2, ..., TN を取ってRを返す関数を格納できる変数vfの宣言
function<R (T1, T2, ..., TN)> vf;
```

幾つか例をあげます。

```
// 0引数で返値なしの関数を格納できます
function<void ()>               f1;

// const char*を受け取ってdoubleを返す関数を格納できます
function<double (const char*)> f2;

// 引数のところには、普通の関数宣言の時同様、仮引数名を書いて構いません。
// 仮引数名はコンパイラには単に無視されますが、人間の目には意味が
```

```
// わかりやすいソースコードになるでしょう。
function<int (HWND wnd, MSG msg, int wparam, int lparam)> wndProc;
```

function オブジェクトには、関数ポインタと比較して柔軟な点がもう一つあります。それは、多少は型が違う関数であっても、呼び出しがうまくいくなら、その関数も格納してしまえる点。次の例を考えてみます。

```
float string_to_real( string str );

const char* p;
double d = string_to_real( p );
```

const char* から string への暗黙の変換と float から double への暗黙の変換が存在するので、このコードは型エラーにはならず、期待通りに実行できます。つまりこの string_to_real 関数は、立派な「const char* を受け取って double を返す関数」として使うことができるわけです。

こんな関数も、function<double (const char*)> オブジェクトへ格納することが可能です。

```
// これはエラー。関数ポインタは厳密に型があっていないとダメ
double (*fun_ptr)(const char*)     = &string_to_real;

// ＯＫ！
function<double (const char*)> fun = &string_to_real;
```


function<Signature> クラステンプレート

必要なヘッダファイル	#include <boost/function.hpp>
名前空間	boost

```
template< typename Signature, typename Allocator = std::allocator<void> >
  class function
  {
  public:
      // Signature == R (T1, T2, …, TN) の場合
      typedef R         result_type;
      typedef Allocator allocator_type;
      typedef T1        argument_type;         // If N == 1
      typedef T1        first_argument_type;   // If N == 2
      typedef T2        second_argument_type;  // If N == 2
      typedef T1        arg1_type;
      typedef T2        arg2_type;
       …
      typedef TN        argN_type;

      function();
      template<typename F>
        function( F fun );
      ~funtcion();

      function& operator=( const function& rhs );
      void swap( function& rhs );

      result_type operator()( arg1_type, arg2_type, …, argN_type ) const;

      void clear();
      bool empty();
      operator safe_bool();
      bool operator!();
```

```
  };

// 他に比較演算子（==,!=）・swap関数が定義されています
```

テンプレート引数Signatureには、functionオブジェクトに格納したい関数の型を指定します。例えば、doubleとchar*を受け取ってintを返す関数を表すfunction型を使いたいときは、function<int (double,char*)>とします。

このクラスのオブジェクトには、関数ポインタや関数オブジェクトなど、Signatureに合った()演算子を持つものなら何でも指定することができます。functionオブジェクト内にはこの関数的オブジェクトのコピーが格納されます。デフォルトコンストラクタでは、何も格納されない「empty」状態となります。

```
result_type operator()( arg1_type, arg2_type, …, argN_type ) const;
```

格納された関数オブジェクトを呼び出します。empty状態ではbad_function_call例外が発生します。

```
bool empty();
operator safe_bool();
bool operator!();
```

現在empty状態かどうかを判定します。

```
void clear();
```

関数オブジェクトを解放し、empty状態にします。

functionN<RetType, T1, T2, ...>クラステンプレート

必要なヘッダファイル	#include <boost/function.hpp>
名前空間	boost

```
template< typename RetType,
          typename Allocator = std::allocator<void> >
  class function0;
template< typename RetType, typename T1,
          typename Allocator = std::allocator<void> >
  class function1;
template< typename RetType, typename T1, typename T2,
          typename Allocator = std::allocator<void> >
  class function2;
...
```

function<Signature>クラスが使用できない古いコンパイラのために残されている別記法です。例えばfunction2<int,double,char*>はfunction<int (double,char*)>と同じ動作をします。定義されているメンバ関数等も全く同一です。

bad_function_call例外

必要なヘッダファイル	#include <boost/function.hpp>
名前空間	boost

```
class bad_function_call : public std::runtime_error;
```

empty状態のfunctionオブジェクトを実行しようとすると発生する例外です。

6.3 無名関数

6.3.1 lambda(入門編)

Boost.Lambdaは、式の途中で無名の関数を定義する機能を提供します。このライブラリに関しては百聞は一見にしかず。まずサンプルをご覧下さい。

```
for_each( v.begin(), v.end(), (cout << _1 << ',') );
```

▲Src. 6-14　lambdaの使用例

この例は、コンテナvの中身をコンマで区切って標準出力へ書き出しています。(cout << _1 << ',')という式自体が、何か1つ引数を受け取ってそれを標準出力へ送り込む、という関数オブジェクトを表しています。_1は、Boost.Bind同様、あとで関数として呼び出したときに第一引数が入る場所、という意味です。Boost.Lambdaを使わずに書くと次のような感じになるでしょう。(仮に、vはint型のコンテナとします。)

```
void output( int x )
{
    cout << x << ',';
}
for_each( v.begin(), v.end(), &output );
```

この場合は、outputという関数をいったん定義してから、それをfor_eachアルゴリズムに渡しています。これをわざわざ独立の関数を用意せずに、式の中で即座に出力関数を書いてしまえるというのがBoost.Lambdaの特徴です。

「(cout << _1 << ',')が関数オブジェクトになる」なんて、ぱっと見、C++としてかなり異質なコードに思えるかも知れません。実際このライブラリの実装は演算子オーバーロードとテンプレートを駆使したトリッキーなものになっていて、古いC++コンパイラでは動作しなかったりもします。しかしそれにも関わらず、Boost.Lambdaには積極的に利用する価値があります。次の例をご覧下さい。vector<int>の中から100未満の要素を探し出す関数を、4通りの方法で書いています。


```
// 1：全部自分でforループを回す
vector<int>::const_iterator my_find1( const vector<int>& v )
{
    vector<int>::const_iterator i;
    for(i=v.begin(); i!=v.end(); ++i)
        if( *i < 100 )
            break;
    return i;
}

// 2：STLのアルゴリズムと普通の関数を使う
bool less_than_100( int x )
{
    return (x < 100);
}
vector<int>::const_iterator my_find2( const vector<int>& v )
{
    return find_if( v.begin(), v.end(), &less_than_100 );
}

// 3：Boost.Bindを使う
vector<int>::const_iterator my_find3( const vector<int>& v )
{
    return find_if( v.begin(), v.end(), bind(less<int>(), _1, 100) );
}

// 4：Boost.Lambdaを使う
vector<int>::const_iterator my_find4( const vector<int>& v )
{
    return find_if( v.begin(), v.end(), (_1 < 100) );
}
```

▲Src. 6-15　100未満の要素を検索するコードの比較

一番簡潔に書けているのがBoost.Lambdaを使った場合です。また、「_1」の意味を知っていることが前提となりますが、「100未満のものを探している」ことを最も読みとりやすいのも、Boost.Lambdaを使った場合でしょう。このようにSTLのアルゴリズムとBoost.Lambdaを組み合わせることで、非常に簡潔で可読性の高いコードが書けるようになります。

幾つか例を見てみましょう。次のサンプルは、いずれもSTLのアルゴリズムへ渡す関数を、Boost.Lambdaを使った表現(「lambda expression」あるいは「ラムダ式」と呼びます)で記述した例です。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <boost/lambda/lambda.hpp>
using namespace std;
using namespace boost;

int main()
{
    using namespace boost::lambda;

    int init_v[] = {-25,-15,0,25,15};
    vector<int> v( init_v, init_v+5 );

    // vの各要素に10を足します
    //「第一引数を参照として受けて、10を足す」関数
    for_each( v.begin(), v.end(), (_1 += 10) );

    // 全要素表示
    //「第一引数をcoutへ出力してから' 'を出力する」関数
    for_each( v.begin(), v.end(), (cout << _1 << ' ') );
    cout << endl;

    // 全要素の二乗の合計を計算しています
    //「第一引数を二乗してからsq_sum変数へ足す」関数
    int sq_sum = 0;
    for_each( v.begin(), v.end(), (sq_sum += _1*_1) );
    cout << "square_sum: " << sq_sum << endl;

    // 前にある要素の方が大きくなっているような箇所を探します
    //「第一引数と第二引数を > で比較してその結果を返す」関数
    vector<int>::iterator it =
        adjacent_find( v.begin(), v.end(), (_1 > _2) );
    cout << "skew_pair: " << *it << ' ' << *(it+1) << endl;

    // 0未満だったり100より大きかったりする要素は0にクリアします
    //「第一引数と0や100を比較して結果を返す」関数
    replace_if( v.begin(), v.end(), (_1 < 0 || 100 < _1), 0 );

    for_each( v.begin(), v.end(), (cout << _1 << ' ') );
    cout << endl;

    return 0;
}
```

▲Src. 6-16　演算子を使ったラムダ式(lambda1.cpp)

実行結果

```
> lambda1
-15 -5 10 35 25
square_sum: 2200
skew_pair: 35 25
0 0 10 35 25
```

lambda::_1やlambda::_2という変数の混ざった式を書くと、ラムダ式になります。あとは、上の例にあげたようにC++の演算子のみを使って書けるような簡単な関数であれば、_1や_2以外の部分は普通にC++の式を書くのと同じように書けばOKです。ラムダ式を関数オブジェクトとして引数を与えて実行すると、_1と書かれた場所に第一引数、_2と書かれた場所に第二引数、と引数が展開されてからラムダ式全体が実行されます。_9まで、つまり九引数関数まで記述できるようになっています。

ここまでの例では作った無名関数を別の関数へ直接渡していましたが、万能関数ポインタことBoost.Functionを使えば、できた無名関数を変数に格納しておくこともできます。

```
function<bool (int, int)>      greater = (_1 > _2);
function<void (int)> output_with_space = (cout << _1 << ' ');
```

…と、基本的な使い方は非常にシンプルです。ですが、このラムダ式の見た目は演算子オーバーロードを使ったトリックで実現されているため、注意すべき落とし穴がいくつか存在します。

例えば、vの各要素を[]でくくって表示するためにラムダ式で関数を書いてみましょう。一番自然なコードは、次のようになると思います。ところがこれでは、期待したとおりには動作しません。

```
// エラーではないけれど…
for_each( v.begin(), v.end(), (cout << '[' << _1 << ']') );
```

▲Src. 6-17　落とし穴その1

実行結果

```
> lambda1
[-15]-5]10]35]25]
```

C++の演算子結合順序の決まりで、for_eachの最後の引数の式は、(((cout << '[') << _1) << ']')と解釈されます。この場合、_1によってラムダ式化が行われる前に(cout<<'[')という式が先に評価されてしまうため、できる関数オブジェクトは(cout<<_1<<']')と同じものになってしまいます。これを避けるには、constant関数を使って、'['などの定数を明示的にラムダ式化する必要があります。

```
for_each( v.begin(), v.end(), (cout << constant('[') << _1 << ']') );
```

▲Src. 6-18　落とし穴その1の解決策

今度は最初に評価される式が(cout<<constant('['))で、coutとラムダ式に関する演算になっているので、全体もちゃんと、'['を表示する関数のラムダ式として評価されます。

もう一つの落とし穴は、Boost.Lambdaライブラリでは実装の都合上、std::endlなどのマニピュレータを使えない、という点です。代わりの手としては、改行文字の出力には'¥n'を使うことになります。

```
for_each( v.begin(), v.end(), (cout << _1 << endl) ); // エラー
for_each( v.begin(), v.end(), (cout << _1 << '¥n') ); // OK
```

▲Src. 6-19　落とし穴その2

最後の落とし穴は、_1などの特別な変数にあります。この名前はBoost.Bindライブラリの使っている名前と同じですが、実体は別物です。したがって、この二つを混ぜて使うとコンパイルエラーとなってしまいます。

```
#include <boost/bind.hpp>
#include <boost/lambda/lambda.hpp>
using namespace boost;
using namespace boost::lambda;

boost::bind( &f, _2, _1 );                  // '_1': あいまいなシンボルです！
for_each( v.begin(), v.end(), cout << _1 ); // '_1': あいまいなシンボルです！
```

▲Src. 6-20　落とし穴その3

回避するには、三つの方法があります。一つ目は、Boost.Bindを使わず全てBoost.Lambdaに統一すること。次の節で紹介するように、LambdaにはBindの機能が全て含まれています。二つ目は、毎回名前空間を指定するようにして、_1などのあいまいさをなくす方法です。

```
bind( &f, boost::_2, boost::_1 );
for_each( v.begin(), v.end(), cout << boost::lambda::_1 );
```

▲Src. 6-21　落とし穴3の回避策

三つ目は、ラムダ式の引数が入る場所を指定する変数を、別の名前で宣言してしまう方法。placeholderN_typeという型で宣言された変数は、_Nの代わりに使うことが可能です。

```
boost::lambda::placeholder1_type X;
for_each( v.begin(), v.end(), cout << X );
```

▲Src. 6-22　落とし穴3の回避策

6.3.2 lambda(発展編)

前節では、演算子(+や=や<など)を適用する無名関数をLambdaで作る例を紹介しました。この節ではさらに、一般の関数呼び出しや、if文のような条件分岐、果てはtry～catch文のような例外処理までをラムダ式で記述する方法を紹介します。

なお、以下では可読性のため、_1, _2ではなくX, Yという名前の変数を使用しています。これは、ソース中で次のように変数X, Yを宣言することで実現できます。

```
boost::lambda::placeholder1_type X;
boost::lambda::placeholder2_type Y;
```

では、ここから一気にさまざまなラムダ式の例をあげます。お楽しみ下さい。

```
// 全ての値を足した結果とかけた結果を計算
int sum=0, prod=1;
for_each( v.begin(), v.end(),
    ( sum  += X,
      prod *= X )
);
```

▲Src. 6-23　複数の文を持つラムダ式

複数の処理を行う無名関数を書くには、二つの処理をカンマ演算子で区切ります。つまり、C++でいうセミコロンが、ラムダ式ではカンマになります。カンマを使ったラムダ式をfor_eachなどに渡す場合は、必ずラムダ式全体を括弧でくくります。そうしないと、カンマがfor_eachに渡す引数を区切る印と解釈されてしまいます。

```
#include <boost/lambda/if.hpp>
// 偶数か奇数かに応じてメッセージを表示
for_each( v.begin(), v.end(),
    if_( X % 2 == 0 ) [
        cout << constant("偶数です¥n")
    ].else_[
        cout << constant("奇数です¥n")
    ]
);
```

▲Src. 6-24　条件分岐

C++のif～else文にあたる処理は、ラムダ式ではif_～else_で表現します。中括弧{}ではなく角括弧[]でブロックを表現することにご注意下さい。else_の省略も可能です。

```
#include <boost/lambda/loops.hpp>

// 各要素を2で割り切れなくなるまで割りまくる
for_each( v.begin(), v.end(),
    while_( X != 0 && X % 2 == 0 ) [
        X /= 2
    ]
);
```

▲Src. 6-25　while_ループ

C++のwhile文にあたる処理は、ラムダ式ではwhile_です。残念ながらbreakやcontinueにあたるものは存在しません。

```
#include <boost/lambda/loops.hpp>

int                 i;
var_type<int>::type li = var(i); // var()でラムダ式用の変数化
for_each( v.begin(), v.end(),
    for_( li=0; li!=X; ++li ) [
        cout << X
    ]
);
```

▲Src. 6-26　for_ループ

C++のfor文にあたる、for_です。ラムダ式中では新しく変数を宣言できないので、ループカウンタを使う場合は、上のようにラムダ式外部で変数を定義してから使う必要があります。

```
#include <boost/lambda/bind.hpp>

// 各要素Xについて、sin(1/X)を表示
for_each( v.begin(), v.end(),
    cout << bind(&sin, 1.0/X) << '¥n'
);
```

▲Src. 6-27　関数呼び出し

ラムダ式の中で他の関数を呼び出すには、関数名(引数リスト)という直感的な書き方は使えません。代わりに、bind(関数名,引数リスト)と書きます。このbindは、Boost.Bindライブラリのものとは別物ですが、機能的には、完全な拡張になっています。全てのboost::bindをboost::lambda::bindに置き換えることも可能です。

```
#include <boost/lambda/bind.hpp>
#include <boost/lambda/exceptions.hpp>

typedef vector<int> table;
table t = ...;

// vの各要素をテーブルのインデックスと見なして、
// その値でテーブルを引いた結果へ変換する処理。
// 普通にforループで書くとこんな感じです。
//   for(vector<int>::iterator i=v.begin(); i!=v.end(); ++i) {
//       int& X = *i;
//       try {
//           X = t.at(X);
//       } catch( std::exception& _e ) {
//           cerr << _e.what() << '¥n';
//           X = 0;
//       }
//   }

const int& (table::*table_at)(size_t) const
    = static_cast<const int& (table::*)(size_t) const>( &table::at );

for_each( v.begin(), v.end(),
    try_catch(
        X = bind( table_at, t, X ),
        catch_exception<std::exception>((
            cerr << bind( &std::exception::what, _e ) << '¥n',
            X = 0
        ))
    )
);
```

▲Src. 6-28　例外処理

try～catch文にあたる、try_catch～catch_exceptionです。捕まえた例外には、特別な変数_eでアクセスできます。

と、このようにラムダ式だけで幾らでも複雑な関数を作ることができます。ここで紹介した以外の文法について、詳しくはリファレンスをどうぞ。ただし、あまり大きな関数になると、ラムダ式で書くよりもちゃんとした一つの関数として切り出した方が読みやすいことが多いです。どちらにするか、よく見極めて下さい。

6.3.3 リファレンス

Boost.Lambdaライブラリでは、「ラムダ式」と呼ばれる式を書くことで、C++の無名関数オブジェクトを表現します。このリファレンスは、C++との相互作用を解説した最初の2節「ラムダ式の実行」「変数・定数」を除いて、全てC++の言語内言語であるラムダ式の文法説明となっています。

ラムダ式の実行

必要なヘッダファイル	#include <boost/lambda/lambda.hpp>
名前空間	boost::lambda

```
template<>
  RetType operator()();
template<typename T1>
  RetType operator()( T1& x1 );
...
template<typename T1, ..., typename TN>
  RetType operator()( T1& x1, ..., TN& xN );
```

ラムダ式は、参照型の引数をとる関数オブジェクトです。()演算子に実引数を渡すことで、ラムダ式を実行します。

```
unspecified make_const( C++式 );
```

ラムダ式の表す関数オブジェクトは参照型の引数を取るので、定数や一時オブジェクトを渡して呼び出すことができません。この問題の一つの解決策として、(_1 + _1)(make_const(1))などのように、make_constでくるんで定数を引数として渡せるようになっています。

```
unspecified const_parameters( ラムダ式 );
unspecified break_const( ラムダ式 );
```


ラムダ式の表す関数オブジェクトは参照型の引数を取りますが、const_parameters(ラムダ式)といった風にconst_parameametersで包むことで、const参照型の引数を取る関数オブジェクトへと変換することができます。ラムダ式へ渡す全ての引数をconst参照としてしまえる場合は、この方法がmake_constの項で述べた問題に対する完璧な解決策になります。

break_constも同様にconst参照型の引数を取る関数オブジェクトへの変換を行いますが、内部ですぐにconst_castを使って再度通常の参照へ戻してラムダ式を実行する、という実装になっています。このため、非const参照とconst参照の混ざった引数達をラムダ式へ渡したいときには有効です。しかし、const_castを用いるため、非常に危険な関数です。この関数の使用は推奨されていません。

```
unspecified unlambda( ラムダ式 );
```

unlambda関数は、ラムダ式を、ラムダ式として扱われない関数オブジェクトへ変換します。これは、ラムダ式のなかでさらに別の無名関数をラムダ式で記述する場合に、内側のラムダ式と外側のラムダ式が合成されないようにunlambda(内側ラムダ式)とくくる形で使用されます。

変数・定数

必要なヘッダファイル	#include <boost/lambda/lambda.hpp>
名前空間	boost::lambda

```
placeholder1_type _1;
placeholder2_type _2;
...
placeholder9_type _9;
```

ラムダ式中のプレースホルダ_Nは、operator()で実行する時に、そのN番目の引数へと展開されます。placeholderN_type型の変数を宣言すれば、_1や_2の代わりに、好きな名前のプレースホルダを使うことも可能です。

この_1,_2,...は、同じ名前がBoost.BindやBoost.MPLでも使われていますが、それぞれ別物です。混用しないようご注意下さい。

```
// expは通常のC++の式とする
constant( exp )
```

C++の式を、ラムダ式中での定数へと変換します。このconstantを使わなくても、ラムダ式中に現れたC++の式は定数として扱われますので、ほとんどの場合このconstantは省略可能です。ただし、演算子の優先順位の関係でC++式とC++式が隣り合ってしまう場合には、その演算をラムダ式の世界で行わせるために、一方のオペランドをconstant()でくくる必要があります。

```
// xは通常のC++の変数とする
var(x)
constant_ref(x)
```

varは、C++の変数を、ラムダ式中で使える変数へと変換します。constant_refは、ラムダ式中での変数へのconst参照に変換します。C++の変数の値を書き換えるようなラムダ式を書く場合は、その変数をvarで包んでラムダ式に埋め込むことになります。

```
constant_type<T>::type
var_type<T>::type
constant_ref_type<T>::type
```

それぞれ、T型の引数でconstant，var，constant_ref関数を呼び出したときの返値の型です。この型の変数にvar関数の結果などを格納して使い回すことで、ラムダ式中に何度もvar(x)を書かなくてもよくなります。

返値型の明示指定

必要なヘッダファイル	#include <boost/lambda/lambda.hpp>
名前空間	boost::lambda

```
ret<type>( ラムダ式 )
```

通常、ラムダ式の型はBoost.Lambdaライブラリによって自動的に推論されます。しかし、多重定義された関数を使っている場合など型の曖昧なラムダ式については、

プログラマが明示的にret<T>(ラムダ式)と書くことで、型の曖昧さのないラムダ式とできます。

演算子

必要なヘッダファイル	#include <boost/lambda/lambda.hpp>
名前空間	boost::lambda

Tbl. 6-1　ラムダ式で使える演算子

算術	+ - * / % << >> & | ^ ! ~
比較	== != < > <= =>
代入	++ -- = += -= *= /= %= <<= >>= &= |= ^=
参照	* [] ->*
論理	&& ||
順次	,

C++で多重定義できる演算子はほぼ全てラムダ式で使用可能です。上の表が、ラムダ式で使える全演算子です。どの演算子も、ラムダ式の実行の際にはC++の時と同じ動作をします。&&や||についてもC++同様short-circuit動作となっています。

()演算子はC++の世界からラムダ式の表す関数オブジェクトへの引数渡しに使われるので、ラムダ式中の関数適用として使うことができません。ラムダ式中で関数適用を行うには、bindを使います。(「関数適用・関数合成」の項参照)

分岐(if)

必要なヘッダファイル	#include <boost/lambda/if.hpp>
名前空間	boost::lambda

```
if_( 条件ラムダ式 )[ ラムダ式 ]
if_( 条件ラムダ式 )[ ラムダ式 ].else_[ ラムダ式 ]
```

C++のif文同様、条件判断に応じてラムダ式を実行します。

分岐(switch)

必要なヘッダファイル	#include <boost/lambda/switch.hpp>
名前空間	boost::lambda

```
switch_statement( 条件ラムダ式,
    case_statement<integer>( ラムダ式 ),
    case_statement<integer>( ラムダ式 ),
    ...
    default_statement( ラムダ式 )
)
```

C++のswitch文と同様、整数による条件分岐を行います。

default_statementは省略可能です。

ループ

必要なヘッダファイル	#include <boost/lambda/loops.hpp>
名前空間	boost::lambda

```
while_( 条件ラムダ式 )[ ラムダ式 ]
do_[ 条件ラムダ式 ].while_( ラムダ式 )
for_( 初期化ラムダ式, 継続条件ラムダ式, 増分ラムダ式 )[ ラムダ式 ]
```

C++のwhile,do～while,for文と同様です。ただし、for_では3つの式のいずれも省略不可能です。

キャスト

必要なヘッダファイル	#include <boost/lambda/casts.hpp>
名前空間	boost::lambda


```
ll_static_cast<type>( ラムダ式 )
ll_reinterpret_cast<type>( ラムダ式 )
ll_const_cast<type>( ラムダ式 )
ll_dynamic_cast<type>( ラムダ式 )
ll_typeid( ラムダ式 )
ll_sizeof( ラムダ式 )
```

C++のキャストや、sizeof、typeid演算子と同様です。

例外処理

必要なヘッダファイル	#include <boost/lambda/exceptions.hpp>
名前空間	boost::lambda

```
try_catch(
    ラムダ式,
  catch_exception<type>( ラムダ式 ),
  catch_exception<type>( ラムダ式 ),
  ...
  catch_all( ラムダ式 )
)
```

C++のtry～catch文と同様です。

C++のcatch(...)の代わりに、catch_allがあります。

```
_e
```

例外オブジェクトを指す変数です。catch_expressionの中でのみ使用できます。

```
throw_exception( ラムダ式 )
rethrow()
```

C++のthrow文と同様です。

rethrowはcatch_expressionまたはcatch_all中でのみ使用でき、現在処理中の例外を再送出します。

コンストラクタ・new・delete

必要なヘッダファイル	#include <boost/lambda/construct.hpp>
名前空間	boost::lambda

```
constructor<T>()
destructor()
```

コンストラクタやデストラクタを呼び出す関数オブジェクトを生成します。(つまり例えば、destructor(obj)ではなく、destructor()(obj)でobjのデストラクタが呼び出されます。)ラムダ式中でこれらの関数オブジェクトを呼び出すには、bindを使います。(「関数適用・関数合成」の項参照)

```
new_ptr<T>()
new_array<T>()
delete_ptr()
delete_array()
```

new式やdelete式を実行する関数オブジェクトを生成します。

関数適用・関数合成

必要なヘッダファイル	#include <boost/lambda/bind.hpp>
名前空間	boost::lambda

Boost.Bindライブラリのラムダ式バージョンです。プレースホルダやprotect関数としてboost::lambda名前空間にあるものを使う必要がある以外は、全てBoost.Bindと同じ機能を備えています。詳細は本書の解説ページをご覧下さい。

ラムダ式中で関数適用を実行するのには、このbindを使用します。

```
int f(int x) { return x*x; }

(lambda::bind(f,_1) + _1) (10);  // f(x)+xを計算する関数
```


標準アルゴリズム

必要なヘッダファイル	#include <boost/lambda/algorithm.hpp>
	#include <boost/lambda/numeric.hpp>
名前空間	boost::lambda

```
ll::for_each()
ll::transform()
...
ll::accumulate()
```

STLのアルゴリズム関数を呼び出す関数オブジェクトを生成します。ラムダ式中でSTLアルゴリズムを使うときはこのラッパを通して使います。標準の<algorithm>ヘッダと<numeric>ヘッダで定義されたアルゴリズムが全て利用できます。

```
call_begin()
call_end()
```

引数のメンバ関数begin()やend()を呼び出す関数オブジェクトを生成します。標準アルゴリズムと組み合わせて使うときに便利です。

```
// 引数をxとすると、for_each( x.begin(), x.end(), cout<<_1 ) を
// 実行する無名関数
bind(
  ll::for_each(),
  bind( call_begin(), _1 ),
  bind( call_end(),   _1 ),
  protect( cout << _1 )
);
```

6.4 構文解析

6.4.1 spirit(文法定義)

関数型プログラミングの章の最後をしめるのは、構文解析ライブラリBoost.Spiritです。構文解析というのは、プログラミング言語や設定ファイルなどの構造を持ったテキストデータを、ただの文字列として与えられた状態から、言語の文法にしたがって構造を取り出す作業です。lexやyaccといったツールが有名です。

テキストデータを処理するライブラリなら、Chapter 2「文字列処理」の章の方が適切なのではないか？　と感じた方もいらっしゃると思います。実はその通りなのですが、しかし一方でこのSpiritライブラリは、構文解析処理を関数型の考えを広く取り込んで実現した、C++における関数型プログラミングの最大の応用例でもあります。そこでこの本では、この意味でSpiritライブラリをChapter 6に配置しました。

では、前置きが済んだところで、簡単な例でSpiritを使ってみましょう。テキストからデータを取り出す処理は後回しにして、まずは、入力テキストが、想定したフォーマットに従っているかどうかのチェックだけを行うプログラムを幾つか書きます。

```
#include <iostream>
#include <string>
#include <boost/spirit.hpp>
using namespace std;
using namespace boost::spirit;

// 文法"MyGrammar"の定義
struct MyGrammar : grammar<MyGrammar>
{
    template<typename ScannerT>
      struct definition
      {
          typedef rule<ScannerT> rule_t;
          rule_t r;

          definition( const MyGrammar& )
          {
              r = ch_p('a') >> *ch_p('b') >> ch_p('c');
          }

          const rule_t& start() const { return r; }
      };
};

int main()
{
    // 文法MyGrammarに基づく構文解析器作成
    MyGrammar parser;

    // 1行読み込み
    string line;
    while( cout<<"# ", getline(cin, line) )
    {
        // 1行構文解析
        parse_info<string::const_iterator> info =
            parse( line.begin(), line.end(), parser );
        // 入力全体をparseできたら、"OK"と表示
        cout << (info.full ? "OK" : "fail") << endl;
    }   // 以下EOFが入力されるまで繰り返し
    return 0;
}
```

▲Src. 6-29　spirit1.cpp

yaccなどでは入力文字列の構文を指定するのには.yファイルというC++とは別のソースファイルを書くのですが、Spiritは、全てC++のソースコードとして構文を指定します。長々としたサンプルですが、この例で構文を指定しているのは、この一文。

```
r = ch_p('a') >> *ch_p('b') >> ch_p('c');
```

これは、文字'a'の次に、文字'b'が0文字以上何文字か繰り返して、その次に文字'c'が続くという構文のテキストを表しています。>>演算子が文字が左右に続くことを示し、*演算子は、0回以上の繰り返しを意味しています。ch_p('a')やch_p('b')、あるいはそれを全部つなげたrなどのことを「パーサ(Parser)」と呼びます。Spiritでは、パーサオブジェクトはこのように構文の指定をする役割と、

後で実際にテキストを解析する役割の二つを兼ねています。

以下は実行して幾つかテキストを打ち込んでみた例です。

実行結果

```
> spirit1
# ac
OK
# abbbbbc
OK
# acc
fail
```

確かに、aの次にbが0個以上並んで最後にc、という場合のみOKが出ていることが確認できます。構文指定の部分だけ色々変えて遊んでみましょう。

```
            r = int_p >> +( '*' >> int_p );
```

実行結果

```
> spirit1-1
# 123*-456*789012*3
OK
# 45
fail
```

int_pはSpiritが標準で提供する部品で、int型で表せる範囲の整数を十進数表示した文字列を認識します。+演算子は、1回以上の繰り返しの意味です。なお、今回は先ほどの例と違って'*'という文字を表すのにch_p('*')としていません。これは、'*'と>>演算子でくっつけている相手がint_pというSpiritの部品であるため、ただの文字'*'もライブラリ側でch_p('*')というパーサに自動で変換してくれることを利用しています。

```
            r = repeat_p(2,4)[upper_p] % ',';
```

実行結果

```
> spirit1-2
# AB,CDEF,XYZ,KKKK
OK
# AIUEO,KA
fail
```


repeat_p(2,4)[upper_p]で、大文字を2個以上4個以下続けた文字列の意味になります。a % bで、bで区切ってaを並べたものという文法を意味します。

…と、こんな風に、ライブラリから色々と文字列パターンを指定するための道具が演算子をふんだんに使って提供されています。しかしこれだけでは、演算子で書ける正規表現ライブラリでしかありませんね。もう少し複雑な例をどうぞ。

```
struct ArithCalc : grammar<ArithCalc>
{
    template<typename ScannerT>
      struct definition
      {
          typedef rule<ScannerT> rule_t;
          rule_t expr, fctr, term;

          definition( const ArithCalc& )
          {
              expr = term >> *('+'>>term | '-'>>term);
              term = fctr >> *('*'>>fctr | '/'>>fctr);
              fctr = real_p | '('>>expr>>')';
          }

          const rule_t& start() const { return expr; }
      };
};
```

▲ Src. 6-30　spirit2.cpp(一部)

main関数は、spirit1.cppの例と文法名だけ変えた同じ物を使います。この6.4節では、以下、基本的にmain関数は省略したソースコードを掲載します。

実行結果

```
> spirit2
# 1.23+4.56
OK
# 3.14*5*5-4.2/(-2.1*0.001+2)+0
OK
# 5^2*pi
fail
```

今度は、加減乗除と、括弧の入った数式の文法を定義しました。構文の指定を、expr(数式)、term(項)、fctr(因子)の3つの規則に分けて記述しています。新し

く登場した | 演算子は左右のどちらか一方とマッチする文字列という意味なので、ArithCalcで定義した内容は次のようになります。

- exprは、termの後ろに'+'termか'-'termを0回以上繰り返し並べたもの（数式。"項+項"や"項-項-項+項"など）
- termは、fctrの後ろに'*'fctrか'/'fctrを0回以上繰り返し並べたもの（項。"因子"や"因子/因子*因子"など）
- fctrは、実数を表す文字列か、または'(' expr ')'（因子。"3.14"や"(数式)"）

exprの規則の中にexprが再度登場する再帰的な定義になっています。これはもうすでに、単なる正規表現では解析できないパターンです。

余談ですが、文字列が文法にあっているかのチェックをするだけならば、この数式の文法は3つの規則に分けずに1行で書くことも可能です。先ほどの例で3つの規則による定義を使った理由は、実は、あとあとで解析した数式を処理しやすくするためです。（次節で解説します。）

```
expr = (real_p | '('>>expr>>')') % (ch_p('+')|'-'|'*'|'/');
```

▲ Src. 6-31　1行で数式の文法を定義

このたった18行のクラスで、文字列が数式であるかどうかを認識する処理が完成しました…と言い切る前に、もう一つだけ改善したい点があります。

実行結果

```
> spirit2
# 1+2
OK
# 1 + 2
fail
```

この文法だと、式の途中に空白を入れることができないのです。確かに空白文字については特に何も書かなかったので当たり前なのですが、かといって、exprやtermの定義にごちゃごちゃと空白の扱いを混ぜるのも読みにくくなるので避けたいところ。

こんな場合は、「スキップパーサ」を指定します。main関数内でparse関数を呼び出している部分に、一つ追加の引数を与えます。


```
    parse_info<string::const_iterator> info =
        parse( line.begin(), line.end(), parser, space_p );
```

parse関数の第四引数は「スキップパーサ」と呼ばれるもので、文法にはあまり関係ないので適宜読み飛ばして欲しい文字を認識するパーサを指定できます。空白文字を無視したければ、Spirit側で定義されたspace_pというパーサを渡します。これで

実行結果

```
> spirit2-1
# 1 + 2*3 - (4   /5)
OK
```

無事に空白を無視してくれるようになりました。

まとめ：Boost.Spiritによる文法記述

Spiritを使って解析したいテキストの文法を記述するには、grammarクラステンプレートから派生したクラスを定義します。grammarへの引数には必ず文法クラス自身を与えるようにします。

```
struct ArithCalc : grammar<ArithCalc>
```

あるいは

```
class ArithCalc : public grammar<ArithCalc>
```

でも構いません。この文法クラスの中には、definitionという名前で内部クラステンプレートを定義して下さい。通常は、このdefinititionのコンストラクタにて文法を全て記述します。

```
    template<typename ScannerT>
      struct definition
```

definitionクラステンプレートの引数には、Spiritライブラリ側で「スキャナ」というオブジェクトを表す型が指定されます。直接この型をいじる機会はまずないので、今のところは単なるおまじないと考えておきましょう。

文法規則は、rule<ScannerT>という型のメンバ変数を定義し、それらrule達と、Spiritが提供する出来合いのパーサを組み合わせる形で記述します。C++の演算子オーバーロードを巧みに使って、BNF記法という文法記法によく似た式で直接文法を書き下せるように設計されています。

```
        typedef rule<ScannerT> rule_t;
        rule_t expr, fctr, term;

        definition( const ArithCalc& self )
        {
            expr = term >> *('+'>>term | '-'>>term);
            term = fctr >> *('*'>>fctr | '/'>>fctr);
            fctr = real_p | '('>>expr>>')';
        }
```

コンストラクタの引数には、解析に使われる文法クラスのオブジェクトが渡されます。使い道は次節で紹介します。

definitionクラスに必要なもう一つのメンバは、startメンバ関数です。

```
        const rule_t& start() const { return expr; }
```

この関数では、文法全体の開始ruleを返します。

以上の条件を満たしてさえいれば、文法クラスには他にどんなメンバ関数やメンバ変数を追加しても大丈夫です。あとは、文法クラスのオブジェクトをparse関数に渡すとSpiritによって解析が行われます。

Column Expression Template技法

Spiritでは、演算子の多重定義を利用してパーサ合成の式を記述します。

```
alpha_p >> *('+' >> alpha_p)
```

このような合成の記法を実現する一つの手段は、オブジェクト指向です。全てのパーサはparserという基底クラスから派生し、演算子は、一般のparserとparserの合成として定義します。

```
class parser;
class alpha_parser : public parser { … }:
class char_parser  : public parser { … };
parser operator>>( parser lhs, parser rhs ) { …合成パーサ作成… }
parser operator*( parser lhs ) { …合成パーサ作成… }
```

しかしこの方式では、パーサを一回合成する毎に、解析時に一段深い仮想関数呼び出しが必要になってしまい、速度を稼ぐことができません。できるならば、合成前のパーサの詳細をparserという一般的な型に落として忘れてしまうことはせず、元のパーサの情報を利用した合成パーサを作りたいところです。

この問題に対する一つのアプローチが、「Expression Template」というテクニックです。これは、Spiritのパーサ合成式のようなC++の式の構造を、全て返値型のテンプレートの構造として記憶するというものです。上の例の合成パーサの型は、実際には次のようになっています。

```
sequence<alpha_parser,
           kleene_star< sequence<chlit<char>,alpha_parser> > >
```

>>で連接したことや、*(クリーネ・スター)で繰り返しをかけたことが全て型の情報として記録されています。この合成パーサは合成前のパーサの型を知っているので、仮想関数呼び出しは不要です。また場合によっては、広範囲の合成の構造を見て、うまい変形を加えて最適化をすることもできます。

Spiritでは途中でパーサをruleに入れるとExpression Templateの型情報の伝達がとぎれてしまうという欠点があるのですが、これを改善するsubruleという機構も用意されています(リファレンス参照)。

Expression Template技法は、BoostではLambda(6.3)やuBLAS(9.3)などでも応用されています。

6.4.2 spirit(アクション)

前節では、入力文字列がSpiritで書いた文法定義と合っているかをチェックするプログラムを書きました。今度は、チェックだけでなく、解析した結果を使って処理を行う方法を紹介します。まずは簡単な例から。

```
void print_int( int x ) { cout << x << "ですよー" << endl; }

struct IntList : grammar<IntList>
{
    template<typename ScannerT>
      struct definition
      {
          typedef rule<ScannerT> rule_t;
          rule_t r;
          definition( const IntList& ) { r = int_p[&print_int] % ',' ; }
          const rule_t& start() const  { return r; }
      };
};
```

▲ Src. 6-32　spirit3.cpp

文法定義のなかに、何やら[&print_int]という見慣れないものがあらわれました。r = int_p % ','だけならば、整数値をコンマで区切ったリストの意味なのですが…。実行結果は次のようになります。

実行結果

```
> spirit3
# 2,3,5
2ですよー
3ですよー
5ですよー
OK
```

さて、おわかりいただけたでしょうか？　Spiritは、文法定義の際にパーサの後ろに[]演算子で関数オブジェクトを渡しておくと、テキストの該当部分が解析されるたびにその関数オブジェクトを実行します。この[]でくっつけた関数のことを、「解析時アクション」とか「意味アクション」と呼びます。Spiritでは、この解析時アクションによって解析されたテキストの処理動作を行います。

それでは早速解析時アクションを使って、整数のリストを構文解析して結果をvectorに格納するプログラムを書いてみます。アクションに指定する関数を別に書くのは面倒ですから、ここは、6.3節で紹介したBoost.Lambdaを使うところでしょう。

```
#include <iostream>
#include <string>
#include <vector>
#include <boost/spirit.hpp>
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/bind.hpp>
using namespace std;
using namespace boost::spirit;

struct IntList2 : grammar<IntList2>
{
    IntList2( vector<int>& vi ) : storage(vi) {}
    vector<int>& storage;

    template<typename ScannerT>
      struct definition
      {
          typedef rule<ScannerT> rule_t;
          rule_t r;
          definition( const IntList2& self )
          {
              using namespace boost::lambda;
              r = int_p[
                bind(vector<int>::push_back,var(self.storage),_1)
              ] % ',' ;
          }
          const rule_t& start() const { return r; }
      };
};

int main()
{
    for( string line; cout<<"# ", getline(cin, line); )
```

```
    {
        vector<int> v;
        if( parse( line.begin(), line.end(), IntList2(v) ).full )
            cout << v.size() << "個" << endl;
    }
    return 0;
}
```

▲ Src. 6-33　spirit4.cpp

実行結果

```
> spirit4
# 2,3,5
3個
# 7,11,13,17,23,29
6個
```

int_pに対するアクションとして、引数を一つ取って、self.storageの末尾へ追加する関数オブジェクトを指定しました。要素を追加するvectorは、文法クラスのコンストラクタと、definitionクラスのコンストラクタを通して与えるようにしています。

なお、例では喜び勇んでBoost.Lambdaを使いましたが、変数への代入やvectorへの追加などのよく使われるアクションについては、簡単に関数オブジェクトが作れるようにSpirit側も用意を整えています。今回のようなコンテナへの追加ならば、push_back_a(コンテナ)です。

```
r = int_p[ push_back_a(self.storage) ] % ',';
```

どのパーサにも解析時アクションを追加できます(ruleや、演算子を使って合成した結果パーサにもアクションを追加できます。解析時アクションを追加ずみのパーサにさらにアクションを追加して、複数個のアクションを実行することもできます。)。しかし、アクション関数に渡される引数は、パーサの種類によって異なります。

Tbl. 6-2　解析時アクションに渡る引数

upper_pなどの一文字パーサ	char
int_p	int
real_p	double
その他	イテレータ2つ

イテレータ2つとは、入力文字列のうち、アクションの付属しているパーサが認識した範囲を指すイテレータです。入力をconst char*で与えればconst char*型が、stringで与えればstring::iterator型が渡されるなど型が不確かなので、関数テンプレートで受けるのが無難でしょう。(スキップパーサを使った場合などは更に別のイテレータが渡されます。)以下に例を示します。

```
struct Abc : grammar<Abc>
{
    struct MyAction
    {
        template<typename Ite>
          void operator()( Ite i1, Ite i2 ) const
            { cout << "文字数:" << i2 - i1 << endl
                   << "  内容:" << string(i1,i2) << endl; }
    };

    template<typename ScannerT>
      struct definition
      {
          typedef rule<ScannerT> rule_t;
          rule_t r;
          definition( const Abc& self )
          {
              r = 'a' >> (*ch_p('b'))[MyAction()] >> 'c';
          }
          const rule_t& start() const { return r; }
      };
};
```

▲ Src. 6-34　spirit5.cpp

実行結果

```
> spirit5
# abbbc
文字数：3
  内容：bbb
```

解析時アクションにも慣れてきたところで、6.4.1で作った数式文法を使って、入力された数式を計算して結果を返す電卓プログラムを作成します。real_pで実数が読みとられた点や、'+'>>term、'*'>>fctrなどで式の値の加減乗除が行われる点にアクションを追加して、スタックを使って計算処理を実現しました。

```
#include <iostream>
#include <string>
#include <stack>
#include <boost/spirit.hpp>
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/bind.hpp>
using namespace std;
using namespace boost::spirit;

// スタックを使って計算処理を行うクラス
template<typename Ite>
  class CalculatorT
  {
  public:
      double answer() const { return stk.top(); }
      void set( double d ) { stk.push(d); }
      void add( Ite, Ite ) { double y=tp(), x=tp(); stk.push(x+y); }
      void sub( Ite, Ite ) { double y=tp(), x=tp(); stk.push(x-y); }
      void mul( Ite, Ite ) { double y=tp(), x=tp(); stk.push(x*y); }
      void div( Ite, Ite ) { double y=tp(), x=tp(); stk.push(x/y); }
  private:
      stack<double> stk;
      double tp() { double d = stk.top(); stk.pop(); return d; }
  };
typedef CalculatorT<string::const_iterator> Calc;

// 文法"CalcGrammar"の定義
struct CalcGrammar : grammar<CalcGrammar>
{
    CalcGrammar( Calc& c ) : cal(c) {}
    Calc& cal;

    template<typename ScannerT>
      struct definition
      {
          typedef rule<ScannerT> rule_t;
          rule_t expr, fctr, term;
          definition( const CalcGrammar& self )
          {
              using namespace boost::lambda;
              var_type<Calc>::type cal( var(self.cal) );

              // '+'>>term や '/'>>fctr にアクションを付加
              expr = term >> *( ('+'>>term) [ bind(&Calc::add,cal,_1,_2) ]
                              | ('-'>>term) [ bind(&Calc::sub,cal,_1,_2) ]
                              );
              term = fctr >> *( ('*'>>fctr) [ bind(&Calc::mul,cal,_1,_2) ]
                              | ('/'>>fctr) [ bind(&Calc::div,cal,_1,_2) ]
                              );
              fctr = real_p                 [ bind(&Calc::set,cal,_1)    ]
                   | '(' >> expr >> ')';
          }
          const rule_t& start() const { return expr; }
      };
};

int main()
{
    for( string s; cout<<"# ", getline(cin, s); )
    {
        Calc cal;
        if( parse( s.begin(), s.end(), CalcGrammar(cal), space_p ).full )
            cout << cal.answer() << endl;
    }
    return 0;
}
```

▲ Src. 6-35　spirit6.cpp

実行結果

```
> spirit6
# 1 + 2
3
# 3.14*5*5 - 4.2 / (-2.1*0.001+2) + 0
76.3978
```

アクションとして呼び出されるCalculatorTオブジェクトは、実数値が読み込まれるとその値をスタックに積み、式の加算が読み込まれると、スタックから値を二つ取り出してその和をスタックに積み戻すといった処理を行っています。"+ 2"のような実際の文字列の情報は必要ないので、addメンバ関数などでは引数は使用していません。ruleを意味のある単位で分割して定義しておいたため、+-*/の優先順位も正しく解釈できています。

構文解析の結果を利用するには、解析時アクションで直接計算を行う方法の他に、一度構文の木構造を作って返すという方法もあります。木構造を作る方法については章末のリファレンスで詳しく解説してありますので、そちらをご覧下さい。

発展：クロージャ

さきほどのスタックを使った計算の例、いまいち自然さに欠けると感じた方もいらっしゃるかもしれません。その原因は、文法の方ではexprやfctrといった単位で式の部分部分をまとめて表していたのにも関わらず、計算機構の方ではスタックという形で全部の式を平らに展開していたことにあると考えられます。

そこで、文法の構造と計算の構造を綺麗に対応させられるようにSpiritに用意されているのが、「クロージャ」です。クロージャは、文法上のrule一個一個ずつに値を持たせる機構です。これを使うと、数式の例で言えば、exprやterm、fctrにdouble型で部分式の値を保持させることができます。

```
#include <iostream>
#include <string>
#include <boost/spirit.hpp>
using namespace std;
using namespace boost::spirit;

struct CalcGrammar2 : grammar<CalcGrammar2>
{
    CalcGrammar2( double& r ) : answer(r) {}
    double& answer;

    // double型のvalというメンバ変数を持つクロージャ
    struct DoubleVal : closure<DoubleVal, double> { member1 val; };

    // 文法定義
```



```
    template<typename ScannerT>
      struct definition
      {
          rule<ScannerT>                        top;
          rule<ScannerT,DoubleVal::context_t> expr, fctr, term;

          definition( const CalcGrammar2& self )
          {
              // クロージャ変数の操作を、解析時アクションで記述
              using phoenix::arg1;
              top  = expr[ assign_a(self.answer) ];
              expr = term[ expr.val=arg1 ]
                       >> *( '+'>>term[ expr.val+=arg1 ]
                           | '-'>>term[ expr.val-=arg1 ] );
              term = fctr[ term.val=arg1 ]
                       >> *( '*'>>fctr[ term.val*=arg1 ]
                           | '/'>>fctr[ term.val/=arg1 ] );
              fctr = real_p[ fctr.val=arg1 ]
                   | '(' >> expr[ fctr.val=arg1 ] >> ')';
          }
          const rule<ScannerT>& start() const { return top; }
      };
};

int main()
{
    double r;
    for( string s; cout<<"# ", getline(cin, s); )
        if( parse( s.begin(), s.end(),
                   CalcGrammar2(r), space_p ).full )
            cout << r << endl;
    return 0;
}
```

▲ Src. 6-36　spirit7.cpp

実行結果

```
# (1+2+3)*4+5+6+7*8+9
100
# (9.87-6.54)/3+2+0.1
3.21
```

順に説明します。

```
    struct DoubleVal : closure<DoubleVal, double> { member1 val; };
```

まず、double型の値を式に持たせるクロージャを、上のような形で宣言します。第一引数に自分自身、第二引数にdouble型を入れてclosureテンプレートから派生し、member1型のメンバ変数を定義しました。メンバ変数名はvalではなく他の名前にしても構いません。（double型ではなくmember1型の変数を作ることに注意してください。）

```
          rule<ScannerT,DoubleVal::context_t> expr, fctr, term;
```

ruleに値を持たせるには、第二テンプレート引数にさっきのDoubleValクロージャの、context_tという型を指定します。これで、解析時アクションの中でexpr.valやfctr.valという形で式の値へアクセスできるようになりました。

```
top  = expr[ assign_a(self.answer) ];
```

また、クロージャを付けたruleの解析時アクションには、そのruleのmember1の値が一引数で渡されるようになります。この例の場合なら、exprの解析時アクションにはexprの式の値がdouble型で渡ります。Spiritに用意されたassign_aアクションを使って、expr式全体の値をself.answerへ代入しています。

さて、値を計算したいexpr式の文法定義は、次のような形をしていました。

```
expr = term >> *('+'>>term | '-'>>term)
```

これに対するアクションは、文法と対応した「exprの値を、一個目のtermの値とそれに続くtermの値を足し引きして計算」という処理を書くことになります。

```
expr = term[ expr.val=arg1 ] >> *( '+'>>term[ expr.val+=arg1 ]
                                 | '-'>>term[ expr.val-=arg1 ] );
```

arg1は、Boost.Lambdaでいう_1と同じ物で、第一引数があとで入る場所です。つまりexpr.val=arg1は、term式の値をexpr.valに代入する無名関数になります。以下、'+'の場合は続くterm式の値を足し、'-'の場合は引き算しています。

この数式文法は再帰的な定義になっているので、expr.valなどの式の値は、ruleが再帰的に呼び出されるたびに別の領域を割り当てて使う必要がありますが、その操作はSpiritが自動的に処理してくれます。ユーザは、再帰を全く気にせずにexpr.valとだけ書いておけば問題なく動作します。クロージャの定義のところでvalをdoubleではなくmember1という謎の型で宣言した理由は、このように、valをラムダ式中に直接置いたり、再帰の時の独自スタックフレーム管理をする必要があったからなのでした。

歴史的な経緯から、Spiritのクロージャでは、無名関数の記述にBoost.Lambdaではなく Phoenixという別のライブラリを使用します。こちらも詳細は章末のリファレンスをご覧下さい。

6.4.3 spirit(概念説明)

ここで少し足を止めて、Spiritを構成している概念について説明します。構文解析フレームワークであるSpiritは、基本的に次の4つの要素の組み合わせで動作しています。

- パーサ(Parser)
- スキャナ(Scanner)
- ディレクティブ(Directive)
- 解析時アクション(Semantic Action)

これらは互いに深く関わりあっていますが、まず、順番に見ていきましょう。

パーサ

パーサは、ある種の文字の列を解析し認識するオブジェクトです。

例えばreal_pパーサは小数の文字列表記となっている文字列を認識しますし、repeat_p(2,3)[alpha_p]は、アルファベットが2文字または3文字並んだ文字列を認識します。ユーザは、各種パーサを組み合わせることで、自分の扱いたい文字列のフォーマットを指定します。そして組み合わされたパーサ達は、Spiritのparse関数に渡されると、入力文字列の解析を始めます。

今までに登場したパーサには、次のようなものがありました。

- ch_p('a')やint_p、real_pなど、Spiritの提供する基本パーサ
- >>や*などの演算子によってパーサ同士を合成してできたもの
- repeat_pやlazy_pなどの関数が返す、複雑な合成パーサ
- rule<ScannerT>オブジェクト
- grammarオブジェクト

これらはすべて同じ、パーサです。grammarではない合成パーサもparse関数に渡して処理ができますし、


```
const char* str = "1,2,3";
if( parse( str, int_p%',' ).full )
    cout << "OK" << endl;
```

逆に、grammarオブジェクトをさらに他のパーサと合成することも可能です。

```
ArichCalc expr; // 6.4.1のspirit2.cppで定義したgrammar

parse( "1+2",   '[' >> expr >> ']' );   // fail
parse( "[1+2]", '[' >> expr >> ']' );   // OK
```

スキャナ

スキャナは、入力文字列とパーサの間に立って、入力文字列を加工してパーサに渡す役目をもっています。これまで目立った形でスキャナは現れていませんでしたが、裏ではしっかり活躍していました。「スキップパーサ」を使ったparse関数呼び出しの例を思い返してみましょう。

```
parse( "a, b, c",  alpha_p%',' );          // fail
parse( "a, b, c",  alpha_p%',', space_p ); // OK
```

スキップパーサを指定しなかった場合は、パーサと入力文字列の間にはlexeme_scannerというスキャナが入ります。このスキャナは文字列のフィルタ処理は何も行わず、そのまま入力をパーサに渡してきます。この状態を、文字レベルの解析といいます。

スキップパーサを指定した場合は、パーサと入力文字列の間にはphrase_scannerというスキャナが入ります。このスキャナは、指定されたスキップパーサに文字レベルの解析でマッチする部分を取り除いた結果をパーサに渡します。space_pを指定していれば、入力文字列中の空白文字は全て無視されるわけです。この状態を、フレーズレベルの解析といいます。

スキャナはこの二種類だけではなく、例えば、入力中の英字を全て小文字に変換して渡すas_lower_scannerというスキャナ(大文字小文字を無視した構文解析に便利です)など、幾つかが定義されています。他のスキャナへの切り替えは、「ディレクティブ」を通して行います。

ディレクティブ

ディレクティブは、パーサの組み合わせによる文法定義の中に混ぜて、「この辺りを解析するときはスキャナをこれに切り替えてね」という指令を出すものです。例えば、as_lower_dというディレクティブは、その中身を解析するときにはスキャナをas_lower_parserに切り替えます。

```
str_p("abc") << ":" << as_lower_d[str_p("abc")]
//                                ↑
// この中を解析するときは、スキャナをas_lowers_scanner に切り替え！
```

これを使うと、':'の左側では小文字の"abc"のみを許し、':'の右側では"ABC"や"aBc"など大文字混じりでも許可する文法が記述できます。

```
// OK
parse( "abc:abc", str_p("abc") << ":" << as_lower_d[str_p("abc")] );
// fail
parse( "Abc:abc", str_p("abc") << ":" << as_lower_d[str_p("abc")] );
// OK
parse( "abc:ABc", str_p("abc") << ":" << as_lower_d[str_p("abc")] );
```

このように、ディレクティブは、[]演算子によってスキャナ切り替えの有効範囲を示します。

文字レベル解析やフレーズレベル解析の途中での切り替えもディレクティブで行えます。こちらもなかなか有用な使い道があります。次の例をご覧下さい。C++の変数名をコンマで区切ったリストを認識する文法を書こうとしています。

```
#include <iostream>
#include <string>
#include <boost/spirit.hpp>
using namespace std;
using namespace boost::spirit;

struct VariableNameList : grammar<VariableNameList>
{
    template<typename ScannerT>
      struct definition
      {
          rule<ScannerT> name, name_list;

          definition( const VariableNameList& )
          {
              // 変数名文字列の定義。先頭はアルファベットか'_'で、
              // 後ろにアルファベットか'_'か数字の列が続く
              name      = (alpha_p|'_') >> *(alpha_p|'_'|digit_p);
              // 変数名のコンマ区切りリスト
              name_list = name % ',';
          }
          const rule<ScannerT>& start() const { return name_list; }
      };
};

int main()
{
    for( string line; cout<<"# ", getline(cin, line); )
        if( parse(line.c_str(), VariableNameList()).full )
            cout << "OK" << endl;
        else cout << "fail" << endl;
    return 0;
}
```

▲Src. 6-37　spirit8.cpp

実行結果

```
> spirit8
# hoge,fuga,Value2,x
OK
# hoge, fuga, Value2, x
fail
```

変数名のコンマ区切りリストは確かに認識するのですが、区切りの周りに空白を入れるとエラーになってしまって使いにくいです。こういうときは、space_pを指定してフレーズレベル解析をすればよいのでした。

```
        if( parse(line.c_str(), VariableNameList(), space_p).full )
```

実行結果

```
> spirit8-1
# hoge, fuga, Value2, x
OK
# h o g e, fuga,Val    ue2, x
OK
```

おっと、しかし、今度はコンマの周りどころか、変数名の途中の空白までもを無視して許可する文法になってしまいます。これは困りました。つまり、全体としてはフレーズレベル解析でも、変数名を解析するruleであるnameの所だけ、文字レベル解析に切り替える必要があります。

```
            name = lexeme_d[ (alpha_p|'_') >> *(alpha_p|'_'|digit_p) ];
// この範囲はスキャナをlexeme_scannerに切り替え！     ↑
```

実行結果

```
> spirit8-1
# hoge, fuga, Value2, x
OK
# h o g e, fuga,Val    ue2, x
fail
```

うまくいきました。

解析時アクション

解析時アクションは、パーサに関連づけておいて、そのパーサが文字列を解析できた時に呼び出される関数です。パーサの解析結果に単なる文字列以上の意味を持たせる役割を担っていることから、意味アクション(Semantic Action)とも言います。

パーサの後ろに[]演算子で関数を指定することで、解析時アクションを関連づけできます。なお、ディレクティブやrepeat_pなどの[]演算子はこれとは別物です。

```
string tag;
parse( "<html>", '<' >> (+~ch_p('>'))[assign_a(tag)] >> '>' );
assert( tag=="html" );
```

解析時アクションについては6.4.2で詳しく扱ったので、特に付け加えるべきことはありません。

スキャナ再び

話が一回りしたところで、再びスキャナの話に戻ります。先ほどスキャナの役目は「入力文字列とパーサの間に立って、入力文字列を加工してパーサに渡すこと」と書きました。実はこれはスキャナの一面しか説明していなかったのです。

スキャナのもう一つの役割は、「パーサの返してくる解析結果を集め、そして必要ならば解析時アクションを呼び出すこと」です。パーサへの入力だけでなく、出力の管理もカバーするのがスキャナと言うわけです。

スキャナが解析時アクションの呼び出しを行っているとすれば、当然、ディレクティブによって「解析時アクションを一切呼ばないスキャナ」へ切り替えたりすることができます。

```
int main()
{
    // IntListは、6.4.2のspirit3.cppで定義した文法
    IntList intl_p;
    if( parse( "1,2,3", intl_p ).full )
         cout << "OK" << endl;
    if( parse( "1,2,3", no_actions_d[intl_p] ).full )
         cout << "OK" << endl;
    return 0;
}
```

▲Src. 6-38　spirit9.cpp

6.4.2で定義したIntListは、リストの各要素ごとに「1ですよー」などと出力してくれる、ちょびっとにぎやかすぎるパーサでした。このパーサの構文認識能力だけを再利用しつつ、余計なアクションを抑止するには、no_actions_dディレクティブでスキャナを切り替えます。

実行結果

```
> spirit9
1ですよー
2ですよー
3ですよー
OK
OK
```

二回目は静かになりましたね。

6.4.4 spirit(発展編)

他の字句解析・構文解析のフレームワークと比較してのSpiritの最大の特徴は、外部のツールや独自の構文定義ファイルなどを使わず、C++コンパイラとC++のソースコードのみで文法の定義が書けるという点にあります。

C++のコードで全てを記述すると言うことは、構文の一部を部品として付けたり外したりするのが非常に簡単になることを意味しています。例えば次の例では、実行時のboolフラグの値に応じて、文法規則を増減させています。

```
struct MyProgrammingLanguage : grammar<MyProgrammingLanguage>
{
    MyProgrammingLanguage( bool eaf=false )
      : enable_advanced_feature( eaf ) {}

    template<typename ScannerT>
      strcuct definition
      {
          definition( const MyProgrammingLanguage& self )
          {
              … 文法定義 …
              if( self.enable_advanced_feature ) {
                  … 発展的な文法の定義 …
              }
          }
      };
};
```

文法定義の時にif文で文法を変えるだけではなく、構文解析の途中に動的に変化するパーサを活用することも可能です。例として、Perl言語の正規表現リテラルの構文(を簡略化したもの)を解析する文法を考えます。

正規表現リテラルは /¥¥d+(m|cm|mm)/ など、通常は'/'で始まって'/'で終わるのですが、最初に'm'をつけると、m#http://#やm!regexp!のように、好きな区切り文字を'/'の代わりに使うことができるのです。この構文をSpiritで表現しようと思うと、こんな風になるでしょう。


```
// confix_p(A,*B,C) は A>>*(B-C)>>Cの意味。
// 今回はサンプルなので正規表現の中身の解析までは行なわず、
// *anychar_pとしてしまいます。

r =        confix_p( '/',       *anychar_p, '/' )
  | 'm' >> confix_p( anychar_p, *anychar_p, 最初のanychar_pの解析結果 );
```

mの直後に来た開き括弧にあたる文字が何であったかによって、最後の閉じ括弧にあたる文字は動的に変化しなければなりません。この問題はクロージャと動的パーサを使うことで解決します。

```
struct RegExpLiteral : grammar<RegExpLiteral>
{
    struct Bracket : closure<Bracket,char> { member1 br; };

    template<typename ScannerT>
      struct definition
      {
          rule<ScannerT> top;
          rule<ScannerT,Bracket::context_t> expr;
          definition( const RegExpLiteral& ) {
              using phoenix::arg1;
              top  = expr;
              expr = confix_p( '/', *anychar_p, '/' )
                   | 'm' >>
                     confix_p( anychar_p[expr.br=arg1],
                               *anychar_p,
                               f_ch_p(expr.br) );
          }
          const rule<ScannerT>& start() const  { return top; }
      };
};
```

▲Src. 6-39 spriti10.cpp

問題の部分はここです。

```
confix_p( anychar_p[expr.br=arg1],
          *anychar_p,
          f_ch_p(expr.br) )
```

まず、'm'の直後の一文字をanychar_pで解析した際に、解析時アクションによって、その文字をexprのクロージャに格納します。そして、閉じの部分では、

f_ch_p(expr.br)というパーサを使います。このパーサはexpr.brという関数の返値の一文字を認識します。つまり、関数の返値が変化すれば、認識する文字も動的に変わるパーサなのです。この例では、expr.br()はクロージャに格納された開き文字が返ってきますから、確かに欲しい文法が定義できています。

実行結果

```
> spirit10
# /hoge/
OK
# /http:///
fail
# m#http://#
OK
# m#...!
fail
```

このクロージャと動的パーサを組み合わせる方法は、他に、XMLパーサの開始タグと終了タグの対応関係を解析する際などに役に立つでしょう。あるいは、C++のようなプログラミング言語のパーサを書く際に、未定義変数の使用をパーサレベルで発見するといった応用も考えられます。(Spiritでは、この目的でsymbolsというクラステンプレートが用意されています。)

6.4.5 リファレンス

概要

必要なヘッダファイル	#include <boost/spirit.hpp>
名前空間	boost::spirit

Boost.Spiritライブラリで導入される用語や概念については、「6.4.3 概念説明」をご覧下さい。また、Spirit全体に関する注意として、現在のバージョンではマルチバイト文字列は一切考慮されていないという点があります。(例えばanychar_pは任意の1文字ではなく、任意の1バイトにマッチします。)特にASCII外の文字を区別しなければならない文法を書くときは、ワイド文字(wchar_t)を使って利用するか、マルチバイトのことを考慮した文法を定義するかのどちらかになります。

ヘッダファイル

Spiritのほぼ全機能を利用できるヘッダファイルとして、<boost/spirit.hpp>が用意されています。しかし、このファイルを#includeするとSpiritの全ソースコードが引用されるため、コンパイル時間の長大化を招くことがあります。そこで、Spiritの必要な機能ごとに分けて#includeできるよう、分割されたヘッダファイルも同時に用意されています。

Tbl. 6-3 分割#include用ヘッダ

機能	ヘッダファイル名
基本的な機能	#include <boost/spirit/core.hpp>
定義済みパーサ	#include <boost/spirit/utility.hpp>
遅延パーサ	#include <boost/spirit/attribute.hpp>
定義済み解析時アクション	#include <boost/spirit/actor.hpp>
シンボルテーブル	#include <boost/spirit/symbols.hpp>
イテレータ	#include <boost/spirit/iterator.hpp>

パーサの合成

Boost.Spirit ライブラリの定義する演算子の多重定義によって、何個かのパーサから一段複雑な別のパーサを作成することが出来ます。

Tbl. 6-4　集合的合成

演算	マッチする文字列	同義のパーサ
P | Q	PまたはQにマッチする文字列	
P & Q	PとQの両方にマッチする文字列	
P - Q	Pにマッチして Q にマッチしない文字列	
P ^ Q	PとQのどちらか一方にだけマッチする文字列	
~P	Pにマッチしない文字列(基本的に、Pが一文字パーサの時のみ使えます)	

Tbl. 6-5　連結・繰り返し合成

演算	マッチする文字列	同義のパーサ
P >> Q	最初P、続けてQにマッチする文字列	
P && Q	P>>Qの別記法	P>>Q
P || Q	P>>QまたはPまたはQ	(P>>!Q)|Q
*P	0回以上のPの繰り返し	
+P	1回以上のPの繰り返し	P>>*P
!P	Pが0回または1回	
repeat_p(n)[P]	ちょうどn回のPの繰り返し	
repeat_p(n1,n2)[P]	n1回以上n2回以下のPの繰り返し	
repeat_p(n1,more)[P]	n1回以上のPの繰り返し	

Tbl. 6-6　リスト合成

演算	マッチする文字列	同義のパーサ
P % Q	Qで区切ってPを並べたリスト	P>>*(Q>>P)
list_p(P,Q,E)	Qで区切ってPを並べ、最後にEが来るリスト	(P%Q)>>!E
list_p(P,Q)	Qで区切ってPを並べたリスト	P%Q
list_p(Q)	Qで区切って任意の文字列を並べたリスト	(*anychar_p-Q)%Q
list_p	コンマで区切って任意の文字列を並べたリスト	(*anychar_p-',')%','

コンマ区切りや空白文字区切りなどのリストを解析するパーサの作成に使われる合成演算です。

Tbl. 6-7　中置合成

演算	マッチする文字列	同義のパーサ
confix_p(S,E,C)	Sで始まり、Eが続いてCで終わる文字列	S>>(E-C)>>C
confix_p(S,E[f],C)	〃	S>>(E-C)[f]>>C
confix_p(S,*E,C)	〃	S>>*(E-C)>>C
confix_p(S,(*E)[f],F)	〃	S>>(*(E-C))[f]>>C

括弧やコメント開始タグの対応関係などを解析するパーサの作成に使われる合成演算です。

定義済みパーサ

一文字パーサ

Tbl. 6-8　一文字パーサ

パーサ	マッチする文字
anychar_p	任意の一文字
ch_p(ch)	文字ch
range_p(ch1,ch2)	ch1以上ch2以下の範囲の一文字
alnum_p	アルファベットもしくは数字
alpha_p	アルファベット
blank_p	スペースもしくはタブ文字
cntrl_p	制御文字
digit_p	数字

graph_p	印字可能かつ表示可能文字
lower_p	英小文字
print_p	印字可能文字
punct_p	句読点
sign_p	'+'または'-'
space_p	スペース、タブ、改行文字
upper_p	英大文字
xdigit_p	16進数字
chset<>(defstr)	定義文字列defstrの表す範囲の一文字
c_escape_ch_p	Cのエスケープ文字列、すなわち¥b, ¥t, ¥n, ¥f, ¥r, ¥¥, ¥", ¥', ¥xHH, ¥OOOのいずれかか、またはただの一文字
lex_escape_ch_p	Cのエスケープ文字列に加え、¥にその他の文字が続く場合もエスケープ文字列として認識

chsetの定義文字列には、例えばchset<>("a-zA-Z012")のようにハイフン(-)を使った記法で範囲を指定します。

一文字パーサに関連づけられた解析時アクション関数には、文字型(char、wchar_tなど)一つが引数として渡されます。

文字列・トークン列パーサ

Tbl. 6-9　文字列パーサ

パーサ	マッチする文字列
str_p(str)	文字列str
str_p(it1,it2)	イテレータit1,it2の指す範囲の文字列
regex_p(rxstr)	指定の正規表現にマッチする文字列
eol_p	改行文字列(CR, LF, CRLFなど)
chseq_p(str)	文字列strの表す、文字の並び
chseq_p(it1,it2)	イテレータit1,it2の指す範囲の文字列の表す、文字の並び
comment_p(str)	strから行末までのトークン列(C++の//コメントと同じ規則)
comment_p(str1,str2)	str1からstr2までのトークン列(C++の/*～*/コメントと同じ規則)

regex_pは、正規表現エンジンとしてBoost.Regex(第Ⅱ部Chapter 2参照)を使います。

str_pとchseq_pの違いは、前者はひとかたまりの文字列を表すのに対して、後者は、空白文字などのトークン境界を間に挟んだ文字の並びにもマッチできる点です。例えば、"A B C"は、str_p("ABC")にはマッチしませんがchseq_p("ABC")にはマッチします。

数値パーサ

Tbl. 6-10　数値パーサ

パーサ	マッチする文字列
uint_parser<T,Radix,Min,Max>	型T、Radix進数、Min桁以上Max桁以下の符号無し整数
bin_p	unsigned int型、2進数表記の符号無し整数
oct_p	unsigned int型、8進数表記の符号無し整数
uint_p	unsigned int型、10進数表記の符号無し整数
hex_p	unsigned int型、16進数表記の符号無し整数
int_parser<T,Radix,Min,Max>	型T、Radix進数、Min桁以上Max桁以下の符号付き整数
int_p	int型、符号付き整数
real_parser<T=double, Policies>	型Tの浮動小数点数
ureal_p	double型、符号無し浮動小数点数
real_p	double型、符号付き浮動小数点数
strict_ureal_p	double型、符号無し浮動小数点数
strict_real_p	double型、符号付き浮動小数点数

real_pとstrict_real_pの違いは、前者は小数点のない数(整数)ともマッチしますが、後者は、小数点の入っている数にのみマッチする点です。

数値パーサに関連づけられた解析時アクション関数には、対応する数値型(int, doubleなど)の値一つが引数として渡されます。

ゼロ文字パーサ

Tbl. 6-11　ゼロ文字パーサ

パーサ	マッチする位置
end_p	入力文字列終端
eps_p	任意の位置の、ゼロ文字の文字列に常にマッチ
eps_p(P)	現在位置からパーサPでマッチすれば、Pで読み進めた分を戻してその位置にゼロ文字マッチ
epsilon_p	eps_pと同じ意味
epsilon_p(P)	eps_pと同じ意味
nothing_p	どんな文字列にもマッチせず、常にマッチ失敗

ゼロ文字パーサは、入力の終端にのみマッチする表現を書きたいときや、特定の箇所で解析時アクションを起こしたいときのダミーとして主に使用されます。ゼロ文字パーサに関連づけられた解析時アクション関数は、0引数で呼び出されます。

パーサを一つとるeps_pは、次の例のように、続くパーサによる判定へ進むかどうかの判定用ダミーとして使われます。

```
eps_p( chset_p("1-5") ) >> uint_p  // 1～5で始まる整数
```

▲Src. 6-40　eps_pの例

制御パーサ

必要なヘッダファイル　`#include <boost/spirit/dynamic.hpp>`

Tbl. 6-12　制御パーサ

パーサ	機能
if_p(CP)[TP]	パーサCPがマッチすれば、次にTPにマッチ。しなければ空列にマッチ。
if_p(CP)[TP].else_p[EP]	CPがマッチすれば次にTPにマッチ。しなければEPにマッチ。
while_p(CP)[BP]	パーサCPがマッチする間、続けてBPにマッチ。
do_p[BP].while_p(CP)	パーサCPがマッチする間、続けてBPにマッチ。
for_p(if,CP,sf)[BP]	パーサCPがマッチする間、続けてBPにマッチ。ループ中にゼロ引数関数if,sfが呼び出される。
select_p(P0,…,Pn)	P0からPnのいずれかにマッチ。つまり、P0|P1|…|Pn。
switch_p(nf) [case_p<0>(P0), … case_p<n>(Pn), default_p]	ゼロ引数関数nfの返値によって、マッチするパーサを切り替える。

select_pに関連づけられた解析時アクション関数には、P0からPnのどれにマッチしたかのインデックスが整数で渡されるのが、P0|…|Pnとの違いです。通常、直後にswitch_pを続けて使用します。select_pは、現在のところBOOST_SPIRIT_SELECT_LIMIT=3個までしか分岐パーサを取ることができません。

```
add_expr = int_p >> while_p('+')[ int_p ]
```

▲Src. 6-41　制御パーサの例

遅延パーサ

Tbl. 6-13　遅延パーサ

パーサ	マッチする文字列
lazy_p(f)	パーサを返すゼロ引数関数pfを呼び出し、返値のパーサとマッチする文字列
f_ch_p(f)	ゼロ引数関数fの返値の文字一文字にマッチ
f_range_p(f1,f2)	f1の返値以上f2の返値以下の範囲の一文字にマッチ
f_str_p(f)	ゼロ引数関数fの返値文字列にマッチ
f_chseq_p(f)	ゼロ引数関数fの返値文字列の表す文字の並びにマッチ

マッチする文字列を関数呼び出しの返値で決定することで、実行時までマッチ対象の決定を遅らせます。他のパーサの解析時アクションによって変わる値を返すような関数を遅延パーサに持たせておくと、別の位置の解析結果に依存して結果の変わるパーサが実現されます。

ディレクティブ

Tbl. 6-14　ディレクティブ一覧

ディレクティブ	効果
as_lower_d[P]	大文字小文字を無視するため、全て小文字に統一して処理する
no_actions_d[P]	P内の解析時アクションは呼び出さずにマッチ処理のみを行う
lexeme_d[P]	パーサを、フレーズレベルでなく文字レベルで動作するように切り替える
limit_d(min,max)[P]	min文字以上max文字以下のマッチのみを採用する
min_limit_d(max)[P]	min文字以上のマッチのみを採用する
max_limit_d(min)[P]	max文字以下のマッチのみを採用する
longest_d[P1|P2|…|Pn]	P1からPnでマッチしたもののうち、最長マッチを採用する
shortest_d[P1|P2|…|Pn]	P1からPnでマッチしたもののうち、最短マッチを採用する
scoped_lock_d(mutex)[P]	Pのマッチ処理中、渡されたmutexをロックする

以下に、いくつかディレクティブの使用例を示します。

```
// "aaaaa"とのマッチを取ると…
          ( str_p("aa") | "a" | "aaa" ) >> *anychar_p // "aa" "aaa"
shortest_d[ str_p("aa") | "a" | "aaa" ] >> *anychar_p // "a" "aaaa"
 longest_d[ str_p("aa") | "a" | "aaa" ] >> *anychar_p // "aaa" "aa"
```

▲Src. 6-42　longest_d,shortest_dの効果

定義済み解析時アクション

Tbl. 6-15　定義済み解析時アクション

アクション	関数の機能
assign_a(x)	x=value
assign_a(x,y)	x=y
increment_a(x)	++x
decrement_a(x)	--x
push_back_a(x)	x.push_back(value)
push_front_a(x)	x.push_back(value)
clear_a(x)	x.clear()
insert_key_a(x,y)	x.insert(pair(value,y))
assign_key_a(x,y)	x[value] = y
erase_a(x)	x.erase(value)
erase_a(x,y)	x.erase(y)
swap_a(x,y)	swap(x,y)

ただし、valueは、パーサが解析時アクション関数に渡す引数とします。一引数で呼び出された場合はその値そのもの、二引数で呼び出された場合は、string::assignメンバ関数で構築されたstringオブジェクトを指します。

clear_aやswap_aなど幾つかのアクションは、valueを単に無視します。

parse関数

```
template<typename CharT, typename D>
  parse_info<const CharT*>
    parse( const CharT* str, const parser<D>& p );

template<typename Iterator, typename D>
  parse_info<Iterator>
    parse( const Iterator& begin, const Iterator& end,
           const parser<D>& p );

template<typename Char, typename D, typename S>
  parse_info<const CharT*>
    parse( const CharT* str,
           const parser<D>& p,
           const parser<S>& skip );

template<typename Iterator, typename D, typename S>
  parse_info<Iterator>
    parse( const Iterator& begin, const Iterator& end,
           const parser<D>& p,
           const parser<S>& skip );
```

文字列ポインタ、または二つのイテレータによって入力文字列を指定し、続く引数で渡されたパーサによって構文解析を実行します。

引数skipがない関数は、文字レベルの構文解析を行います。つまり、特別な処理は行わず、一文字一文字入力の文字列とパーサの指定とのマッチ処理が実行されます。

引数skipがある関数は、語句レベルの構文解析を行います。skipに指定されたパーサとマッチする部分が読み飛ばされ、残りの文字列でパーサpとのマッチ処理が実行されます。

parse_info構造体

```
template<typename Iterator>
  struct parse_info
  {
      bool     hit;
      bool     full;
      Iterator stop;
      size_t   length;
  };
```

parse関数による解析の結果情報を返します。

```
      bool     hit;
```

入力文字列の前半だけでもマッチしたらtrue。

```
      bool     full;
```

入力文字列の全体がマッチしたらtrue。

```
      Iterator stop;
```

マッチした部分の終端を示すイテレータ。

```
      size_t   length;
```

マッチした部分の長さ。

構文木の作成

入力文字列とパーサから、構文木を作成して返すための機能が用意されています。解析時アクションを通して構文解析と並行に処理を行うのではなく、テキストデータの解析結果をとっておいて後で何度も操作を加えたい時などに使用します。

pt_parse関数

必要なヘッダファイル	#include <boost/spirit/tree/parse_tree.hpp>

```
template<class Factory,typename Iterator,
         typename Parser, typename Skip>
  tree_parse_info<Iterator,Factory>
    pt_parse( const Iterator& begin, const Iterator& end,
              const Parser& parser, const Skip& skip,
              const Factory& factory );
template<typename Iterator, typename Parser, typename Skip>
  tree_parse_info<Iterator>
    pt_parse( const Iterator& begin, const Iterator& end,
              const Parser& parser, const Skip& skip );
template<typename Iterator, typename Parser>
  tree_parse_info<Iterator>
    pt_parse( const Iterator& begin, const Iterator& end,
              const Parser& parser );
template<typename Char, typename Parser, typename Skip>
  tree_parse_info<const Char*>
    pt_parse( const Char* str, const Parser& parser, const Skip& skip );
template<typename Char, typename Parser>
  tree_parse_info<const Char*>
    pt_parse( const Char* str, const Parser& parser );
```

構文解析を行い、文法中の各ruleをノードに、ruleの構成要素を子ノードにする構文木を生成して返します。例えば、本書の6.4.1などで登場した数式パーサを使って「2*(3+4)」という式から構文木を作ると、次のような木になります。

```
              +-- fctr ---"2"
              |
expr -- term -+-- "*"
              |         +-- "("     +-- term --- "3"
              +-- fctr -+-- expr ---+-- "+"
                        +-- ")"     +-- term --- "4"
```

ast_parse関数

必要なヘッダファイル `#include <boost/spirit/tree/ast.hpp>`

```
template<class Factory, typename Iterator,
         typename Parser, typename Skip>
  tree_parse_info<Iterator,Factory>
    ast_parse( const Iterator& begin, const Iterator& end,
               const Parser& parser, const Skip& skip,
               const Factory& factory );
template<typename Iterator, typename Parser, typename Skip>
  tree_parse_info<Iterator>
    ast_parse( const Iterator& begin, const Iterator& end,
               const Parser& parser, const Skip& skip );
template<typename Iterator, typename Parser>
  tree_parse_info<Iterator>
    ast_parse( const Iterator& begin, const Iterator& end,
               const Parser& parser );
template<typename Char, typename Parser, typename Skip>
  tree_parse_info<const Char*>
    ast_parse( const Char* str, const Parser& parser, const Skip& skip );
template<typename Char, typename Parser>
  tree_parse_info<const Char*>
    ast_parse( const Char* str, const Parser& parser );
```

ほぼpt_parse関数と同じですが、ノードの子が一つしかなかった場合は、ノードを破棄し、その位置に子ノードを繰り上げた木を返す点が異なります。例えば、本書の6.4.1などで登場した数式パーサを使って「2*(3+4)」という式から構文木を作ると、次のような木になります。


```
      +-- "2"
      |
term -+-- "*"
      |         +-- "("     +-- "3"
      +-- fctr -+-- term ---+-- "+"
                +-- ")"     +-- "4"
```

ast_parse関数は通常、木構造指定ディレクティブと合わせて、ruleそのものを指すノードが全て取り除かれた形の木を得るために使用されます。

木構造指定ディレクティブ

leaf_node_d	このディレクティブ範囲の文字列を全て合わせて木の一つのノードとする。
token_node_d	〃
discard_node_d	このディレクティブの範囲のノードは全て木から取り除く。
discard_first_node_d	このディレクティブの範囲の、先頭のノードを木から取り除く。
discard_last_node_d	このディレクティブの範囲の、末尾のノードを木から取り除く。
inner_node_d	このディレクティブの範囲の、先頭と末尾のノードを木から取り除く。
infix_node_d	このディレクティブの範囲の、偶数番目のノードを木から取り除く。中にリストパーサを置いて、区切り文字のノードを除去するという使い方が多い。
no_node_d	このディレクティブの範囲からは木のノードを作らない。
root_node_d	このディレクティブの指すノードを、現在のrule全体が作る木のルートノードとし、左右のノードは全てこのノードの子とする。
gen_pt_node_d	このディレクティブの範囲ではpt_parse方式でノードを生成する。
gen_ast_node_d	このディレクティブの範囲ではast_parse方式でノードを生成する。
access_node_d	木のノードに格納する値を計算するためのアクションを付加する。access_node_d[P][f]と書くと、パーサPの作るノードのルートノードと、Pの解析した範囲の先頭、末尾を指すイテレータの計3つの引数でfが呼び出される。fの返値はノードのvalueメンバ変数へ格納される。

例えば、本書の6.4.1などで登場した数式パーサを次のように変更して

```
expr = term % root_node_d[ ch_p('+')|ch_p('-') ];
term = fctr % root_node_d[ ch_p('*')|ch_p('/') ];
fctr = real_p | inner_node_d[ '('>>expr>>')' ];
```

「2*(3+4)」という式からast_parse関数で構文木を作ると、次のような木になります。

```
      +-- "2"
"*" --+         +-- "3"
      +-- "+" --+
                +-- "4"
```

tree_parse_info<Iterator>構造体

```
template<typename Iterator = const char*>
  struct tree_parse_info
  {
      Iterator    stop;
      bool        match;
      bool        full;
      std::size_t length;
      std::vector< tree_node<node_val_data<Iterator>> > trees;
  };
```

pt_parse関数やast_parse関数の結果を格納している構造体です。parse関数の返値であるparse_infoと基本的に同じですが、木構造情報を格納したtreesメンバ変数がある点だけが異なります。パースに成功している場合treesは1つの要素だけを含むコンテナで、trees[0]が木のルートを表しています。

構文木を構成するデータ型

```
template<typename T>
  struct tree_node
  {
      T value;
      std::vector<tree_node> children;
  };
```


一般に木構造を表すためのデータ型です。pt_parseやast_parseの返値の場合は、Tとして、次で解説するnode_val_data構造体が入っています。

```
template<typename Iterator = const char*, typename ValueT = nil_t>
  struct node_val_data
  {
      typedef typename
        std::iterator_traits<Iterator>::value_type  value_type;
      typedef typename
        std::vector<value_type>::const_iterator const_iterator;

      const_iterator begin()   const;
      const_iterator end()     const;
      parser_id      id()      const;
      bool           is_root() const;
      const ValueT&  value()   const;
  };
```

beginからendの範囲の文字列が、このノードの指すトークン文字列です。idメンバ関数は、このトークンを生成したパーサ(もしあれば)の識別番号を返します。パーサやrule、grammarにもidメンバ関数が存在し、自分の識別番号を返すようになっているので、その値と比較することで、どのパーサから作られたノードかを判定することが可能です。

valueメンバ関数はデフォルトでは意味を持ちません。pt_parse関数やast_parse関数で、Factoryテンプレート引数にnode_val_data_factory<T>型を指定すると、各ノードに型Tの値を設定できるようになり、その値をvalueメンバ関数で取り出すことができます。valueの設定は、access_node_dディレクティブを使って行います。

木構造の例をあげます。「木構造指定ディレクティブ」の項であげた例

```
      +-- "2"
"*" --+         +-- "3"
      +-- "+" --+
                +-- "4"
```

では、

```
*info.trees[0].children[1].children[0].value.begin()
```

の値が'3'になります。

構文木のXML化

必要なヘッダファイル	#include <boost/spirit/tree/tree_to_xml.hpp>

```
template<typename TreeNodeVec, typename AssocContainer>
         typename GetIdFun, typename GetValueFun>
  void tree_to_xml( std::ostream& os, const TreeNodeVec& tree,
                    const std::string& comment,
                    const AssocContainer& id_to_name,
                    const GetIdFun& get_token_id,
                    const GetValueFun& get_token_value );
template<typename TreeNodeVec, typename AssocContainer>
  void tree_to_xml( std::ostream& os, const TreeNodeVec& tree,
                    const std::string& comment,
                    const AssocContainer& id_to_name );
template<typename TreeNodeVec>
  void tree_to_xml( std::ostream& os, const TreeNodeVec& tree,
                    const std::string& comment = "" );
```

pt_parse関数やast_parse関数の返す木構造をXML形式でストリームへ出力します。引数osにストリームを指定し、引数treeにはtree_parse_info.treesなど、ノードのvectorを指定するようです。バージョン1.31現在では出力エンコーディングはISO-8859-1に固定されており、あくまでデバッグ用のダンプ出力程度の使用にとどめた方が良いと思われます。

引数commentには、XMLのDTD宣言直後に埋め込まれるコメントの中に書く文字列を指定します。

引数id_to_nameには、パーサの識別番号をrule属性値となる文字列へ変換するためのstd::mapを指定します。

引数get_token_id、get_token_valueには、トークンを引数に取りidや文字列を返す関数を指定します。

grammar

grammar(文法)は、Spiritによって大規模な言語の解析を行うための枠組みです。

文法クラスの基本形

```
// 文法クラス GrammarClass の定義
class GrammarClass : public grammar<GrammarClass>
{
public:
    template<typename ScannerT>
      class definition
      {
      public:
          definition( const GrammarClass& self )
          {
               ra = rb >> rc; // ここで構文定義や
               …              // 解析時アクションの関連づけ
          }
          const rule<ScannerT>& start() const { return ra; }
          rule<ScannerT> ra, rb, rc, …; // ルール変数
      };
};
```

▲Src. 6-43　文法クラスの基本形

文法は、grammarクラステンプレートから派生し、内部にdefinitionという型テンプレートを持つクラスで表現します。上の文法クラスの基本形のソースコードにあげたメンバ関数が実装されたクラスでさえあればその他のメンバは自由に実装して構いませんが、ほとんどの場合、definitionクラスにrule<ScannerT>型のメンバ変数を持ち、definitionクラスのコンストラクタの中で、パーサの合成演算によって構文の定義を行う実装とすることが一般的です。また、startメンバ関数では、言語の文法のトップレベルのruleを返します。

文法クラスのオブジェクトそれ自体も、Spiritではパーサとして使うことができます。つまり、parse関数に渡して文字列の構文解析に使ったり、パーサの合成演算で他のパーサと組み合わせることが可能です。

rule<ScannerT> クラステンプレート

rule は、任意のパーサへの参照を格納しておけるオブジェクトです。通常は、grammar の中で、言語の構文生成規則一つに rule 一つを割り当てる形で使用します。

subrule<int N>

rule はクラスの継承と仮想関数を使った仕組みで任意のパーサを格納できるようにしているため(4.4.2 any のコラム参照)、rule を使った構文解析には仮想関数呼び出しが多数発生し、速度が若干低下するという問題点があります。しかし一方で、細かく rule に分割をしないと記述の難しい文法が現実には大多数を占めます。

例えば次は、三個の rule を持つ文法の例です。

```
rule<ScannerT> ra, rb, rc;
definition( const MyGrammar& self )
{
    ra = rb >> *("+" >> rb);
    rb = rc >> *("*" >> rc);
    rc = '(' >> ra >> ')'  |  '1'
}
const rule<ScannerT>& start() const { return ra; }
```

▲Src. 6-44　rule を使った文法定義の例

これは、subrule を使うと、rule の数(と、構文解析時に発生する仮想関数呼び出し)を1つに押さえることができます。以下が、subrule を使って書き直した例です。

```
rule<ScannerT> top;
subrule<0> ra;
subrule<1> rb;
subrule<2> rc;
definition( const MyGrammar& self )
{
    top = (
      ra = rb >> *("+" >> rb),
      rb = rc >> *("*" >> rc),
      rc = '(' >> ra >> ')'  |  '1'
    );
}
const rule<ScannerT>& start() const { return top; }
```

▲Src. 6-45　subrule を使って書き直し

top= の式の右辺全体が一つの式になったため、ra, rb, rc の再帰構造は全て型情報としてコンパイル時に推論されます。

subrule の使い方は、次の3点に留意すれば、rule による定義と特に違いはありません。

- subrule は、rule 定義(この例では rule top の定義)の中で使う。
- セミコロン(;)区切りではなく、コンマ(,)区切りで複数の subrule 定義を行う。
- 一番上で定義した subrule(この例では ra)が、rule(top)の定義となる。

なお、subrule の使用は実行時の効率を向上させますが、膨大な数のクラステンプレートの使用によって実現されているため、コンパイル時にコンパイラに負担をかけます。現状では、全ての文法を subrule で書いてしまうこともまた、現実的ではないようです。

シンボルテーブル

```
template<typename T=int, typename CharT=char, typename SetT=impl_defd>
  class symbols
  {
      functor<…> add;
  };
```

シンボルテーブルは、文字列(シンボル)から T 型の値へのマッピングです。シンボルテーブルはパーサの一種で、parse 関数に渡して文字列の構文解析に使ったり、パーサの合成演算で他のパーサと組み合わせることが可能です。言語の変数テーブルの処理などへの応用が考えられます。

```
symbols<int> sym;
sym.add ("aaa",1) ("bbb",2) ("ccc",1) ("ddd",3); // 登録
"+" >> sym[f] >> "+"    // +aaa+ や +bbb+ などとマッチするパーサ
parse( "+ccc+", "+" >> sym[f] >> "+" ); // f(1) が呼び出される
```

▲Src. 6-46　シンボルテーブルの使い方

マッチの際にはテーブルに登録されたシンボル名全ての | による結合として扱われ、解析時アクション関数には、マッチしたシンボルに対応する値として登録されていた値が渡されます。

登録・検索関数

```
sym = symname1, symname2, symname3, …;
sym.add( const CharT* symname, const T& value );
sym.add( const CharT* begin, const CharT* end, const T& value );
T* add(  symbols<T,CharT>& sym, const CharT* symname, const T& value );
```

以上のいずれかの方法で、シンボルテーブル sym へのシンボルの登録ができます。value を指定しない形式の場合、T 型のデフォルト値で登録が行われます。

```
T* find( symbols<T,CharT>& sym, const CharT* symname );
```

シンボルテーブルからシンボルと対応する値を検索します。

Phoenix

必要なヘッダファイル	#include <boost/spirit/phoenix.hpp>
名前空間	phoenix

Phoenix は Boost.Spirit のサブライブラリで、無名関数オブジェクトを書くための機能を提供しています。3.3 節で紹介した Boost.Lambda とほぼ同じ目的を持っていますが、Boost と Spirit は元は別々に開発されていたという歴史的経緯から、二種類のライブラリが並立しています。

現在の Boost.Spirit は Phoenix と共に使うことが前提となっているため、ここでは Phoenix についての簡単なリファレンスを、Boost.Lambda と対比する形で掲載します。それぞれの詳細については 3.3 節をご覧下さい。なお、将来的には両ライブラリは統合される予定だそうです。

Phoenix と Boost.Lambda の対比

Phoenix	機能	Boost.Lambda
arg1, arg2, …, argN	プレースホルダ	_1, _2, …, _N
val(c)	C++ 定数	constant(c)
var(x)	C++ 変数	var(x)

+, -, *, /, …	演算子全般	+, -, *, /, …
if_(…)[…].else_[…]	if 分岐	if_(…)[…].else_[…]
なし	switch 条件分岐	switch_statement(…)
while_(…)[…]	while ループ	while_(…)[…]
do_[…].while_(…)	do 〜 while ループ	do_[…].while_(…)
for_(…,…,…)[…]	for ループ	for_(…,…,…)[…]
static_cast_<T>(…) など	キャスト	ll_static_cast<T>(…) など
なし	try 〜 catch	try_catch(…)
construct_<T>	コンストラクタ	constructor<T>()
new_<T>	new	new_ptr<T>()
なし	delete	delete_ptr()

Lazy function と bind 関数適用

```
bind( std::less<int>, _1, constant(4) )
                        // 第一引数の方が 4 より小さい時 true を返す関数
bind( std::less<int>, constant(1), constant(2) )   // 常に true を返す関数
bind( std::less<int>, constant(1), constant(2) )() // 値 true
```

▲Src. 6-47　Boost.Lambda の場合

Boost.Lambda の場合、関数適用は bind 関数によって表現しました。

Phoenix では、まず通常の関数から「Lazy Function」オブジェクトを作成し、その Lazy Function を無名関数式中で普通の関数のような () による構文で使用します。

```
functor< std::less<int> > less_;

less_(arg1, val(4))     // 第一引数の方が 4 より小さい時 true を返す関数
std::less<int>()(1, 2)  // 値 true
less_(1, 2)             // 常に true を返す関数
less_(1, 2)()           // 値 true
```

▲Src. 6-48　Phoenix の場合

この Phoenix の方法の利点は、無名関数式中での関数適用が、bind を使う方式と比較して自然な表現になる点にあります。

```
// Boost.Lambda
if_( bind(&cmp,arg1,bind(&f,arg2)) )[ … ] // 無名関数オブジェクト
```

```
// Phoenix
function_ptr<bool,int,int> cmp_ = &cmp;
function_ptr<int,int>        f_ = &f;
if_( cmp_(arg1, f_(arg2)) )[ … ]          // 無名関数オブジェクト
```

▲Src. 6-49　比較

Lazy Functionは、以下のような型の変数に保持します。

- function_ptr<Ret,Arg1,…,ArgN>
- functor<FuncObjClass>
- member_function_ptr<Ret,Class,Arg1,…,ArgN>
- member_var_ptr<Type,Class>

また、Phoenixのbind関数を使うと、通常の関数をLazy Functionへとその場で変換も可能です。

```
using phoenix::bind;
bind(std::less<int>())( arg2, arg1 )// 無名関数オブジェクト
```

▲Src. 6-50　phoenix::bind

Phoenixを利用するところ

解析時アクション関数の定義に無名関数を使う場合、Boost.Lambdaを使うよりもPhoenixを使った方が相性がよいようです。また、次の項目「クロージャ」は、Phoenixに強く依存したSpiritの機能です。

クロージャ

クロージャは、grammarやrule、subruleに、解析時アクションから操作できる好きな型のデータ格納領域を追加するための仕組みです。このクロージャによるデータ格納領域は、ruleなどが再帰的に使われている場合に自動的に領域をスタック的に拡大するので、再帰を気にせず使うことができます。

```
// 型Type1の値v1、…、型TypeNの値vNを
// 格納するクロージャMyClosureの定義
struct MyClosure
  : closure<MyClosure, Type1, Type2, …, TypeN>
{
    member1 v1;  // 変数名はv1, …, vNに限らず自由に決めて良いが、
    member2 v2;  // 型は常に(Type1, …, TypeNに何を指定していても)
     …           // ここではmember1, …, memberNとする。
    memberN vN;
};
```

▲Src. 6-51　クロージャ定義の基本形

クロージャは、closureクラステンプレートから派生することで定義します。closureの第一テンプレート引数にはクロージャ型自身を、第二以降の引数には、クロージャに格納したいデータの型を順に記述します。

```
// クロージャ付きgrammarの定義の例
class MyGrammar : public grammar<MyGrammar,MyClosure::context_t>
  { … };

// クロージャ付きruleの宣言の例
rule<ScannerT, MyClosure::context_t> ra;

// クロージャ付きsubruleの宣言の例
subrule<2, MyClosure::context_t> sr2;
```

▲Src. 6-52　文法関係のパーサにクロージャをつける例

grammar, rule, subruleの第二テンプレート引数として「クロージャ型名::context_t」を指定します。すると、それらのルールや文法にデータ格納領域が追加されます。

```
struct IntCls : closure<IntCls,int> { member1 val; };

rule<ScannerT,IntCls::context_t> ra, rb, rc;
definition( const MyGrammar& self )
{
    using phoenix::arg1;
    ra = rb[ra.val = arg1] >> *("+" >> rb[ra.val += arg1]);
    rb = rc[rb.val = arg1] >> *("*" >> rc[rc.val *= arg1]);
    rc = '(' >> ra[rc.val=arg1] >> ')'  |  ch_p('1')[rc.val = 1];
}
```

▲Src. 6-53　クロージャを使う例

解析時アクションからクロージャへは、Phoenixを使った無名関数式の内部で、「ルール名.クロージャのメンバ名」の形でアクセスします。

また、クロージャの付いたルールは、自身に付属した解析時アクション関数を、

member1の値を引数として呼び出しますので、アクション関数側では、プレースホルダarg1でその値を受け取ります。

```
ra = rb[ra.val = arg1] >> *("+" >> rb[ra.val += arg1]);
```

▲Src. 6-54　クロージャを使う例

この例では、rb[ra.val = arg1]で、「arg1に受け取ったrb.valの値をra.valへ代入するアクション関数」をrbに付属させています。さらに、rb[ra.val += arg1]では、「arg1に受け取ったrb.valの値をra.valへ加算するアクション関数」です。

便利なイテレータ

構文解析に使われるライブラリであるBoost.Spiritは、特にファイルに書かれたテキストに対して解析処理を実行する用途によく使われます。そこで、ファイルの内容を走査するイテレータとして有用なクラスがBoost.Spirit内に幾つか定義されています。

multi_pass<Iterator>クラステンプレート

```
template<typename InputIterator>
  class multi_pass;

template<typename InputIterator>
  multi_pass<InputIterator> make_multi_pass( InputIterator i );
```

▲Src. 6-55　ここまで

一度しかデータを読みとれない可能性のあるイテレータ(istream::iteratorなど)を、バッファにデータを溜めておくことで複数回の読みとりを可能にするアダプタです。multi_passクラステンプレートのインスタンスもまた、イテレータとして機能します。

file_iterator<T>クラステンプレート

```
template<typename T = char>
  class file_iterator
  {
  public:
      file_iterator( const char* filename );

      file_iterator make_end() const;
      operator safe_bool() const;
  };
```

コンストラクタでファイルfilenameを開き、そのファイル内容の上を走るイテレータとなります。ファイルをifstreamで開き、ifstream::begin()を呼んでmulti_passクラステンプレートで複数パス化する、という手間を省いた型として使うことができます。

position_iterator<typename Iterator>クラステンプレート

```
template<typename Iterator>
  class position_iterator
  {
  public:
      position_iterator();
      position_iterator( Iterator begin, Iterator end );

      void set_tabchars( int tabN );
      const file_position& get_position() const;
      void set_position( const file_position& pos );
  };

strcut file_position
{
    std::string file;
    int         line;    // 最初の行は1
    int         column;  // 最初の文字は1
};
```

改行文字の位置をカウントすることで、現在がファイル中の何行目、何桁目であるかを記憶しているイテレータです。例えばSpiritを使った構文解析のエラー時にエラーメッセージを表示するなどの場面で便利です。

Column 下降型と上昇型とSpirit

構文解析を行うアルゴリズムの方針には、主に二種類あります。

一つは、下降型(Top-Down)構文解析。こちらは、Spiritで採用されている方針です。例えば、1+2*3を、次の文法に従って解析してみましょう。

```
const rule_t& start() { return expr; }
expr = term >> *('+' >> term);
term = fctr >> *('*' >> fctr);
fctr = int_p | '(' >> expr >> ')';
```

まず、開始ルールであるexprを使って解析しようと試みます。exprの先頭はtermで、termの先頭はfctrルールが来ていますから、結局、最初はfctrルールに従った解析が行われます。fctrはint_pか、または'('で始まるとわかっているので、入力の最初の1文字が数字の1ならば、ここはint_pの方で解析を進めればいいと判断できます。このように、開始ルールから下へ降りていくのが下降型です。

もう一つは、上昇型(Bottom-Up)構文解析。こちらはyaccなどが採用しています。

今度は、まず入力文字列の先頭の1文字'1'をにらんでから、'1'で始まりうる基本ルールを文法から探します。すぐに、int_pが見つかりますね。次に、int_pから始まるルールを探すとこれはfctrだけ、fctrから始まるルールはtermだけです。termはfctrの後ろに'*'が続く可能性があるルールですが、入力では'+'が続いているので、さらに、termから始まるルールexprへと昇り、解析を続けます。このように、基本ルールから上へ昇っていくのが上昇型です。

Spiritでは、Expression Templateでの文法記述や、f_ch_p等の遅延パーサを簡単に実装できることから下降型の方法を導入していますが、こちらには一つ短所があります。Spiritでは、「左再帰」を含む文法を解析できません。左再帰とは、ルールの定義の一番左にそのルール自身が登場する状態を指します。

```
add_expr = int_p | add_expr >> '+' >> int_p
```

add_exprを解析するために、すぐ下のadd_exprルールへ降りていって、そのadd_exprのためにまた下のadd_exprへ…という無限ループに陥ってしまいます。左再帰が見つかった場合、プログラマが手で文法を変形して左再帰を解消する必要があります。(この場合なら例えば、次のように。)

```
add_expr = int_p >> *('+' >> int_p)
```