```cpp
/*=============================================================================
    Copyright (c) 2001-2003 Daniel Nuffer
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
////////////////////////////////////////////////////////////////////////////////
//
//  Demonstrates the ASTs. This is discussed in the
//  "Trees" chapter in the Spirit User's Guide.
//
////////////////////////////////////////////////////////////////////////////////
#define BOOST_SPIRIT_DUMP_PARSETREE_AS_XML

#include <boost/spirit/core.hpp>
#include <boost/spirit/tree/ast.hpp>
#include <boost/spirit/tree/tree_to_xml.hpp>
#include "tree_calc_grammar.hpp"

#include <iostream>
#include <stack>
#include <functional>
#include <string>
#include <cassert>

#if defined(BOOST_SPIRIT_DUMP_PARSETREE_AS_XML)
#include <map>
#endif

// This example shows how to use an AST.
/////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

typedef char const*         iterator_t;
typedef tree_match<iterator_t> parse_tree_match_t;
typedef parse_tree_match_t::tree_iterator iter_t;

/////////////////////////////////////////////////////////////////////////////
long evaluate(parse_tree_match_t hit);
long eval_expression(iter_t const& i);

long evaluate(tree_parse_info<> info)
{
    return eval_expression(info.trees.begin());
}

long eval_expression(iter_t const& i)
{
    cout << "In eval_expression. i->value = " <<
        string(i->value.begin(), i->value.end()) <<
        " i->children.size() = " << i->children.size() << endl;

    if (i->value.id() == calculator::integerID)
    {
        assert(i->children.size() == 0);

        // extract integer (not always delimited by '\0')
        string integer(i->value.begin(), i->value.end());
```

```cpp
        return strtol(integer.c_str(), 0, 10);
    }
    else if (i->value.id() == calculator::factorID)
    {
        // factor can only be unary minus
        assert(*i->value.begin() == '-');
        return - eval_expression(i->children.begin());
    }
    else if (i->value.id() == calculator::termID)
    {
        if (*i->value.begin() == '*')
        {
            assert(i->children.size() == 2);
            return eval_expression(i->children.begin()) *
                eval_expression(i->children.begin()+1);
        }
        else if (*i->value.begin() == '/')
        {
            assert(i->children.size() == 2);
            return eval_expression(i->children.begin()) /
                eval_expression(i->children.begin()+1);
        }
        else
            assert(0);
    }
    else if (i->value.id() == calculator::expressionID)
    {
        if (*i->value.begin() == '+')
        {
            assert(i->children.size() == 2);
            return eval_expression(i->children.begin()) +
                eval_expression(i->children.begin()+1);
        }
        else if (*i->value.begin() == '-')
        {
            assert(i->children.size() == 2);
            return eval_expression(i->children.begin()) -
                eval_expression(i->children.begin()+1);
        }
        else
            assert(0);
    }
    else
    {
        assert(0); // error
    }

    return 0;
}

/////////////////////////////////////////////////////////////////////////////
int
main()
{
    // look in tree_calc_grammar for the definition of calculator
    calculator calc;

    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tThe simplest working calculator...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";
    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        tree_parse_info<> info = ast_parse(str.c_str(), calc);

        if (info.full)
        {
#if defined(BOOST_SPIRIT_DUMP_PARSETREE_AS_XML)
            // dump parse tree as XML
            std::map<parser_id, std::string> rule_names;
            rule_names[calculator::integerID] = "integer";
            rule_names[calculator::factorID] = "factor";
            rule_names[calculator::termID] = "term";
            rule_names[calculator::expressionID] = "expression";
            tree_to_xml(cout, info.trees, str.c_str(), rule_names);
#endif

            // print the result
            cout << "parsing succeeded\n";
            cout << "result = " << evaluate(info) << "\n\n";
        }
        else
        {
            cout << "parsing failed\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  Demonstrates use of boost::bind and spirit
//  This is discussed in the "Functional" chapter in the Spirit User's Guide.
//
//  [ JDG 9/29/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/bind.hpp>
#include <iostream>
#include <vector>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace boost;

///////////////////////////////////////////////////////////////////////////////
//
//  Our comma separated list parser
//
///////////////////////////////////////////////////////////////////////////////
class list_parser
{
public:

    typedef list_parser self_t;

    bool
    parse(char const* str)
    {
        return boost::spirit::parse(str,

            //  Begin grammar
            (
                real_p
                [
                    bind(&self_t::add, this, _1)
                ]

                >> *(   ','
                        >>  real_p
                            [
                                bind(&self_t::add, this, _1)
                            ]
                    )
            )
            ,
            //  End grammar

            space_p).full;
    }
```

```

    void
    add(double n)
    {
        v.push_back(n);
    }

    void
    print() const
    {
        for (vector<double>::size_type i = 0; i < v.size(); ++i)
            cout << i << ": " << v[i] << endl;
    }

    vector<double> v;
};

///////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\tA comma separated list parser for Spirit...\n";
    cout << "\tDemonstrates use of boost::bind and spirit\n";
    cout << "/////////////////////////////////////////////////////////\n\n";

    cout << "Give me a comma separated list of numbers.\n";
    cout << "The numbers will be inserted in a vector of numbers\n";
    cout << "Type [q or Q] to quit\n\n";

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        list_parser lp;
        if (lp.parse(str.c_str()))
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << str << " Parses OK: " << endl;

            lp.print();

            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```cpp
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  Boiler plate [ A template for writing your parser ]
//
//  [ JDG 9/17/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
//
//  Semantic actions
//
///////////////////////////////////////////////////////////////////////////////
struct my_action
{
    template <typename IteratorT>
    void operator()(IteratorT first, IteratorT last) const
    {
        string s(first, last);
        cout << "\tMy Action got: " << s << endl;
    }
};

///////////////////////////////////////////////////////////////////////////////
//
//  My grammar
//
///////////////////////////////////////////////////////////////////////////////
struct my_grammar : public grammar<my_grammar>
{
    template <typename ScannerT>
    struct definition
    {
        definition(my_grammar const& self)
        {
            my_rule =
                *lexeme_d[(+graph_p)[my_action()]]
                ;
        }

        rule<ScannerT> my_rule;
        rule<ScannerT> const&
        start() const { return my_rule; }
    };
};

///////////////////////////////////////////////////////////////////////////////
```

```cpp
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\t A boiler−plate parser...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type anything or [q or Q] to quit\n\n";

    my_grammar g;

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == ’q’ || str[0] == ’Q’)
            break;

        if (parse(str.c_str(), g, space_p).full)
        {
            cout << "parsing succeeded\n";
        }
        else
        {
            cout << "parsing failed\n";
        }
    }

    cout << "Bye... :−) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2001-2003 Dan Nuffer
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
////////////////////////////////////////////////////////////////////////////
//
//  Full calculator example using STL functors with debugging enabled.
//  This is discussed in the "Functional" chapter in the Spirit User's Guide
//  and the Debugging chapter.
//
//  Ported to Spirit v1.5 from v1.2/1.3 example by Dan Nuffer
//  [ JDG 9/18/2002 ]
//  [ JDG 7/29/2004 ]
//
////////////////////////////////////////////////////////////////////////////

#define BOOST_SPIRIT_DEBUG
#include <boost/spirit/core.hpp>
#include <iostream>
#include <stack>
#include <functional>
#include <string>

////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

////////////////////////////////////////////////////////////////////////////
//
//  Semantic actions
//
////////////////////////////////////////////////////////////////////////////
struct push_int
{
    push_int(stack<long>& eval_)
    : eval(eval_) {}

    void operator()(char const* str, char const* /*end*/) const
    {
        long n = strtol(str, 0, 10);
        eval.push(n);
        cout << "push\t" << long(n) << endl;
    }

    stack<long>& eval;
};

template <typename op>
struct do_op
{
    do_op(op const& the_op, stack<long>& eval_)
    : m_op(the_op), eval(eval_) {}

    void operator()(char const*, char const*) const
    {
        long rhs = eval.top();
        eval.pop();
```

```
        long lhs = eval.top();
        eval.pop();

        cout << "popped " << lhs << " and " << rhs << " from the stack. ";
        cout << "pushing " << m_op(lhs, rhs) << " onto the stack.\n";
        eval.push(m_op(lhs, rhs));
    }

    op m_op;
    stack<long>& eval;
};

template <class op>
do_op<op>
make_op(op const& the_op, stack<long>& eval)
{
    return do_op<op>(the_op, eval);
}

struct do_negate
{
    do_negate(stack<long>& eval_)
    : eval(eval_) {}

    void operator()(char const*, char const*) const
    {
        long lhs = eval.top();
        eval.pop();

        cout << "popped " << lhs << " from the stack. ";
        cout << "pushing " << -lhs << " onto the stack.\n";
        eval.push(-lhs);
    }

    stack<long>& eval;
};

////////////////////////////////////////////////////////////////////////////
//
//  Our calculator grammar
//
////////////////////////////////////////////////////////////////////////////
struct calculator : public grammar<calculator>
{
    calculator(stack<long>& eval_)
    : eval(eval_) {}

    template <typename ScannerT>
    struct definition
    {
        definition(calculator const& self)
        {
            integer =
                lexeme_d[ (+digit_p)[push_int(self.eval)] ]
                ;

            factor =
                    integer
                |   '(' >> expression >> ')'
                |   ('-' >> factor)[do_negate(self.eval)]
                |   ('+' >> factor)
                ;
            term =
                factor
                >> *(   ('*' >> factor)[make_op(multiplies<long>(), self.eval)]
                    |   ('/' >> factor)[make_op(divides<long>(), self.eval)]
                    )
                    ;

            expression =
                term
                >> *(  ('+' >> term)[make_op(plus<long>(), self.eval)]
                    |   ('-' >> term)[make_op(minus<long>(), self.eval)]
                    )
                    ;

            BOOST_SPIRIT_DEBUG_NODE(integer);
            BOOST_SPIRIT_DEBUG_NODE(factor);
            BOOST_SPIRIT_DEBUG_NODE(term);
            BOOST_SPIRIT_DEBUG_NODE(expression);
        }

        rule<ScannerT> expression, term, factor, integer;
        rule<ScannerT> const&
        start() const { return expression; }
    };

    stack<long>& eval;
};

////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tThe simplest working calculator...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";

    stack<long> eval;
    calculator  calc(eval); //  Our parser
    BOOST_SPIRIT_DEBUG_NODE(calc);

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        parse_info<> info = parse(str.c_str(), calc, space_p);

        if (info.full)
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << "result = " << calc.eval.top() << endl;
            cout << "-------------------------\n";
        }
        else
```

```
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "stopped at: \": " << info.stop << "\"\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```cpp
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  Plain calculator example demostrating the grammar and semantic actions.
//  This is discussed in the "Grammar" and "Semantic Actions" chapters in
//  the Spirit User's Guide.
//
//  [ JDG 5/10/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
//
//  Semantic actions
//
///////////////////////////////////////////////////////////////////////////////
namespace
{
    void    do_int(char const* str, char const* end)
    {
        string  s(str, end);
        cout << "PUSH(" << s << ')' << endl;
    }

    void    do_add(char const*, char const*)    { cout << "ADD\n"; }
    void    do_subt(char const*, char const*)   { cout << "SUBTRACT\n"; }
    void    do_mult(char const*, char const*)   { cout << "MULTIPLY\n"; }
    void    do_div(char const*, char const*)    { cout << "DIVIDE\n"; }
    void    do_neg(char const*, char const*)    { cout << "NEGATE\n"; }
}

///////////////////////////////////////////////////////////////////////////////
//
//  Our calculator grammar
//
///////////////////////////////////////////////////////////////////////////////
struct calculator : public grammar<calculator>
{
    template <typename ScannerT>
    struct definition
    {
        definition(calculator const& /*self*/)
        {
            expression
                =   term
                    >> *(   ('+' >> term)[&do_add]
                        |   ('-' >> term)[&do_subt]
                        )
```

```cpp
                ;

            term
                =   factor
                    >> *(   ('*' >> factor)[&do_mult]
                        |   ('/' >> factor)[&do_div]
                        )
                ;

            factor
                =   lexeme_d[(+digit_p)[&do_int]]
                |   '(' >> expression >> ')'
                |   ('-' >> factor)[&do_neg]
                |   ('+' >> factor)
                ;
        }

        rule<ScannerT> expression, term, factor;

        rule<ScannerT> const&
        start() const { return expression; }
    };
};

///////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tExpression parser...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";

    calculator calc;    //  Our parser

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        parse_info<> info = parse(str.c_str(), calc, space_p);

        if (info.full)
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "stopped at: \": " << info.stop << "\"\n";
            cout << "-------------------------\n";
        }
    }
    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2001-2003 Hartmut Kaiser
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  This example shows:
//  1.  Parsing of different comment styles
//          parsing C/C++-style comment
//          parsing C++-style comment
//          parsing PASCAL-style comment
//  2.  Parsing tagged data with the help of the confix_parser
//  3.  Parsing tagged data with the help of the confix_parser but the semantic
//      action is directly attached to the body sequence parser
//
///////////////////////////////////////////////////////////////////////////////

#include <string>
#include <iostream>
#include <cassert>

#include <boost/spirit/core.hpp>
#include <boost/spirit/utility/confix.hpp>
#include <boost/spirit/utility/chset.hpp>


///////////////////////////////////////////////////////////////////////////////
// used namespaces
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
// actor called after successfully matching a single character
class actor_string
{
public:
    actor_string(std::string &rstr) :
        matched(rstr)
    {
    }

    void operator() (const char *pbegin, const char *pend) const
    {
        matched += std::string(pbegin, pend-pbegin);
    }

private:
    std::string &matched;
};

///////////////////////////////////////////////////////////////////////////////
// actor called after successfully matching a C++-comment
void actor_cpp (const char *pfirst, const char *plast)
{
    cout << "Parsing C++–comment" <<endl;
    cout << "Matched (" << plast-pfirst << ") characters: ";

    char cbbuffer[128];
```

```
    strncpy(cbbuffer, pfirst, plast-pfirst);
    cbbuffer[plast-pfirst] = ’\0’;

    cout << "\"" << cbbuffer << "\"" << endl;
}

///////////////////////////////////////////////////////////////////////////////
// main entry point
int main ()
{

///////////////////////////////////////////////////////////////////////////////
//
//  1.  Parsing different comment styles
//      parsing C/C++-style comments (non-nested!)
//
///////////////////////////////////////////////////////////////////////////////

    char const* pCComment = "/* This is a /* nested */ C–comment */";

    rule<> cpp_comment;

    cpp_comment =
            comment_p("/*", "*/")           // rule for C-comments
        |   comment_p("//")                 // rule for C++ comments
        ;

    std::string comment_c;
    parse_info<> result;

    result = parse (pCComment, cpp_comment[actor_string(comment_c)]);
    if (result.hit)
    {
        cout << "Parsed C–comment successfully!" << endl;
        cout << "Matched (" << (int)comment_c.size() << ") characters: ";
        cout << "\"" << comment_c << "\"" << endl;
    }
    else
    {
        cout << "Failed to parse C/C++–comment!" << endl;
    }
    cout << endl;

    //      parsing C++-style comment
    char const* pCPPComment = "// This is a C++–comment\n";
    std::string comment_cpp;

    result = parse (pCPPComment, cpp_comment[&actor_cpp]);
    if (result.hit)
        cout << "Parsed C++–comment successfully!" << endl;
    else
        cout << "Failed to parse C++–comment!" << endl;

    cout << endl;


    //      parsing PASCAL-style comment (nested!)
    char const* pPComment = "{ This is a (* nested *) PASCAL–comment }";

    rule<> pascal_comment;
    pascal_comment =                    // in PASCAL we have two comment styles
            comment_nest_p('{', '}')    // both may be nested
        |   comment_nest_p("(*", "*)")
        ;

    std::string comment_pascal;

    result = parse (pPComment, pascal_comment[actor_string(comment_pascal)]);
    if (result.hit)
    {
        cout << "Parsed PASCAL-comment successfully!" << endl;
        cout << "Matched (" << (int)comment_pascal.size() << ") characters: ";
        cout << "\"" << comment_pascal << "\"" << endl;
    }
    else
    {
        cout << "Failed to parse PASCAL-comment!" << endl;
    }
    cout << endl;

///////////////////////////////////////////////////////////////////////////////
//
//  2.  Parsing tagged data with the help of the confix parser
//
///////////////////////////////////////////////////////////////////////////////

    std::string body;
    rule<> open_tag, html_tag, close_tag, body_text;

    open_tag =
            str_p("<b>")
        ;

    body_text =
            anychar_p
        ;

    close_tag =
            str_p("</b>")
        ;

    html_tag =
            confix_p (open_tag, (*body_text)[actor_string(body)], close_tag)
        ;

    char const* pTag = "<b>Body text</b>";

    result = parse (pTag, html_tag);
    if (result.hit)
    {
        cout << "Parsed HTML snippet \"<b>Body text</b>\" successfully "
            "(with re-attached actor)!" << endl;
        cout << "Found body (" << (int)body.size() << " characters): ";
        cout << "\"" << body << "\"" << endl;
    }
    else
    {
        cout << "Failed to parse HTML snippet (with re-attached actor)!"
            << endl;
    }
    cout << endl;
```

```
///////////////////////////////////////////////////////////////////////////////
//
//  3.  Parsing tagged data with the help of the confix_parser but the
//      semantic action is directly attached to the body sequence parser
//      (see comment in confix.hpp) and out of the usage of the 'direct()'
//      construction function no automatic refactoring takes place.
//
//      As you can see, for successful parsing it is required to refactor the
//      confix parser by hand. To see, how it fails, you can try the following:
//
//          html_tag_direct =
//              confix_p.direct(
//                  str_p("<b>"),
//                  (*body_text)[actor_string(bodydirect)],
//                  str_p("</b>")
//              )
//              ;
//
//      Here the *body_text parser eats up all the input up to the end of the
//      input sequence.
//
///////////////////////////////////////////////////////////////////////////////

    rule<> html_tag_direct;
    std::string bodydirect;

    html_tag_direct =
            confix_p.direct(
                str_p("<b>"),
                (*(body_text - str_p("</b>")))[actor_string(bodydirect)],
                str_p("</b>")
            )
        ;

    char const* pTagDirect = "<b>Body text</b>";

    result = parse (pTagDirect, html_tag_direct);
    if (result.hit)
    {
        cout << "Parsed HTML snippet \"<b>Body text</b>\" successfully "
            "(with direct actor)!" << endl;
        cout << "Found body (" << (int)bodydirect.size() << " characters): ";
        cout << "\"" << bodydirect << "\"" << endl;
    }
    else
    {
        cout << "Failed to parse HTML snippet (with direct actor)!" << endl;
    }
    cout << endl;

    return 0;
}
```

```
/*==============================================================================
    Copyright (c) 2001-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  A complex number micro parser (using subrules)
//
//  [ JDG 5/10/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <iostream>
#include <complex>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
//
//  Our complex number micro parser
//
///////////////////////////////////////////////////////////////////////////////
bool
parse_complex(char const* str, complex<double>& c)
{
    double rN = 0.0;
    double iN = 0.0;

    subrule<0> first;
    subrule<1> r;
    subrule<2> i;

    if (parse(str,

        //  Begin grammar
        (
            first = '(' >> r >> !(',' >> i) >> ')' | r,
            r = real_p[assign(rN)],
            i = real_p[assign(iN)]
        )
        ,
        //  End grammar

        space_p).full)
    {
        c = complex<double>(rN, iN);
        return true;
    }
    else
    {
        return false;
    }
}

////////////////////////////////////////////////////////////////////////////
```

```
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tA complex number micro parser for Spirit...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";

    cout << "Give me a complex number of the form r or (r) or (r,i) \n";
    cout << "Type [q or Q] to quit\n\n";

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        complex<double> c;
        if (parse_complex(str.c_str(), c))
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << str << " Parses OK: " << c << endl;
            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 1998-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  This sample demonstrates error handling as seen in the
//  Error Handling" chapter in the User’s Guide.
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/error_handling/exceptions.hpp>
#include <iostream>
#include <cassert>

using namespace std;
using namespace boost::spirit;

struct handler
{
    template <typename ScannerT, typename ErrorT>
    error_status<>
    operator()(ScannerT const& /*scan*/, ErrorT const& /*error*/) const
    {
        cout << "exception caught...Test concluded successfully" << endl;
        return error_status<>(error_status<>::fail);
    }
};

int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tExceptions Test...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";

    assertion<int>  expect(0);
    guard<int>      my_guard;

    rule<> start =
        my_guard(ch_p(’a’) >> ’b’ >> ’c’ >> expect( ch_p(’d’) ))
        [
            handler()
        ];

    bool r = parse("abcx", start).full;

    assert(!r);
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2003 Pavel Baranov
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  An alternate error-handling scheme where the parser will
//  complain (but not stop) if input doesn't match.
//
//  [ Pavel Baranov 8/27/2003 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/utility/functor_parser.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

static short errcount = 0;

///////////////////////////////////////////////////////////////////////////////
//
//  Error reporting parser
//
///////////////////////////////////////////////////////////////////////////////
struct error_report_parser {

    error_report_parser(const char *msg) : _msg(msg) {}

    typedef nil_t result_t;

    template <typename ScannerT>
    int operator()(ScannerT const& scan, result_t& /*result*/) const
    {
        errcount++;
        cerr << _msg << endl;
        return 0;
    }

private:
    string _msg;
};

typedef functor_parser<error_report_parser> error_report_p;

///////////////////////////////////////////////////////////////////////////////
//
//  My grammar
//
///////////////////////////////////////////////////////////////////////////////
struct my_grammar : public grammar<my_grammar>
{
    static error_report_p error_missing_semicolon;
    static error_report_p error_missing_letter;
```

```
    template <typename ScannerT>
    struct definition
    {
        definition(my_grammar const& self) :
            SEMICOLON(';')
        {
            my_rule
                = *(eps_p(alpha_p|SEMICOLON) >>
                   (alpha_p|error_missing_letter) >>
                   (SEMICOLON|error_missing_semicolon))
            ;
        }

        chlit<>
            SEMICOLON;

        rule<ScannerT> my_rule;

        rule<ScannerT> const&
        start() const { return my_rule; }
    };
};

error_report_p my_grammar::error_missing_semicolon("missing semicolon");
error_report_p my_grammar::error_missing_letter("missing letter");

///////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "////////////////////////////////////////////////////////////\n\n";
    cout << " Error handling demo\n\n";
    cout << " The parser expects a sequence of letter/semicolon pairs\n";
    cout << " and will complain (but not stop) if input doesn't match.\n\n";
    cout << "////////////////////////////////////////////////////////////\n\n";

    my_grammar g;

    string str( "a;;b;cd;e;fg;" );
    cout << "input: " << str << "\n\n";

    if( parse(str.c_str(), g, space_p).full && !errcount )
        cout << "\nparsing succeeded\n";
    else
        cout << "\nparsing failed\n";

    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002 Jeff Westfahl
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  A parser that echoes a file
//  See the "File Iterator" chapter in the User's Guide.
//
//  [ JMW 8/05/2002 ]
//
///////////////////////////////////////////////////////////////////////////////

#include <boost/spirit/core.hpp>
#include <boost/spirit/iterator/file_iterator.hpp>
#include <iostream>

///////////////////////////////////////////////////////////////////////////////
using namespace boost::spirit;

////////////////////////////////////////////////////////////////////////////
//
//  Types
//
////////////////////////////////////////////////////////////////////////////
typedef char                        char_t;
typedef file_iterator<char_t>       iterator_t;
typedef scanner<iterator_t>         scanner_t;
typedef rule<scanner_t>             rule_t;

////////////////////////////////////////////////////////////////////////////
//
//  Actions
//
////////////////////////////////////////////////////////////////////////////
void echo(iterator_t first, iterator_t const& last)
{
    while (first != last)
        std::cout << *first++;
}

////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////
int
main(int argc, char* argv[])
{
    if (2 > argc)
    {
        std::cout << "Must specify a filename!\n";
        return -1;
    }

    // Create a file iterator for this file
    iterator_t first(argv[1]);
```

```
    if (!first)
    {
        std::cout << "Unable to open file!\n";
        return -1;
    }

    // Create an EOF iterator
    iterator_t last = first.make_end();

    // A simple rule
    rule_t r = *(anychar_p);

    // Parse
    parse_info <iterator_t> info = parse(
        first,
        last,
        r[&echo]
    );

    // This really shouldn't fail...
    if (info.full)
        std::cout << "Parse succeeded!\n";
    else
        std::cout << "Parse failed!\n";

   return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2001-2003 Dan Nuffer
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  Full calculator example using STL functors
//  This is discussed in the "Functional" chapter in the Spirit User’s Guide.
//
//  Ported to Spirit v1.5 from v1.2/1.3 example by Dan Nuffer
//  [ JDG 9/18/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <iostream>
#include <stack>
#include <functional>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
//
//  Semantic actions
//
///////////////////////////////////////////////////////////////////////////////
struct push_int
{
    push_int(stack<long>& eval_)
    : eval(eval_) {}

    void operator()(char const* str, char const* /*end*/) const
    {
        long n = strtol(str, 0, 10);
        eval.push(n);
        cout << "push\t" << long(n) << endl;
    }

    stack<long>& eval;
};

template <typename op>
struct do_op
{
    do_op(op const& the_op, stack<long>& eval_)
    : m_op(the_op), eval(eval_) {}

    void operator()(char const*, char const*) const
    {
        long rhs = eval.top();
        eval.pop();
        long lhs = eval.top();
        eval.pop();

        cout << "popped " << lhs << " and " << rhs << " from the stack. ";
```

```
        cout << "pushing " << m_op(lhs, rhs) << " onto the stack.\n";
        eval.push(m_op(lhs, rhs));
    }

    op m_op;
    stack<long>& eval;
};

template <class op>
do_op<op>
make_op(op const& the_op, stack<long>& eval)
{
    return do_op<op>(the_op, eval);
}

struct do_negate
{
    do_negate(stack<long>& eval_)
    : eval(eval_) {}

    void operator()(char const*, char const*) const
    {
        long lhs = eval.top();
        eval.pop();

        cout << "popped " << lhs << " from the stack. ";
        cout << "pushing " << -lhs << " onto the stack.\n";
        eval.push(-lhs);
    }

    stack<long>& eval;
};

///////////////////////////////////////////////////////////////////////////////
//
//  Our calculator grammar
//
///////////////////////////////////////////////////////////////////////////////
struct calculator : public grammar<calculator>
{
    calculator(stack<long>& eval_)
    : eval(eval_) {}

    template <typename ScannerT>
    struct definition
    {
        definition(calculator const& self)
        {
            integer =
                lexeme_d[ (+digit_p)[push_int(self.eval)] ]
                ;

            factor =
                    integer
                |   ’(’ >> expression >> ’)’
                |   (’-’ >> factor)[do_negate(self.eval)]
                |   (’+’ >> factor)
                ;

            term =
                factor
                >> *(   (’*’ >> factor)[make_op(multiplies<long>(), self.eval)]
                    |   ('/' >> factor)[make_op(divides<long>(), self.eval)]
                    )
                    ;

            expression =
                term
                >> *(  ('+' >> term)[make_op(plus<long>(), self.eval)]
                    |   ('-' >> term)[make_op(minus<long>(), self.eval)]
                    )
                    ;
        }

        rule<ScannerT> expression, term, factor, integer;
        rule<ScannerT> const&
        start() const { return expression; }
    };

    stack<long>& eval;
};

//////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
//////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tThe simplest working calculator...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";

    stack<long> eval;
    calculator  calc(eval); //  Our parser

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        parse_info<> info = parse(str.c_str(), calc, space_p);

        if (info.full)
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "stopped at: \": " << info.stop << "\"\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    Copyright (c) 2002 Juan Carlos Arevalo-Baeza
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
#include <boost/spirit/core.hpp>
#include <boost/spirit/utility/functor_parser.hpp>
#include <boost/spirit/actor/assign_actor.hpp>
#include <iostream>
#include <vector>
#include <string>

///////////////////////////////////////////////////////////////////////////////
//
//  Demonstrates the functor_parser. This is discussed in the
//  "Functor Parser" chapter in the Spirit User’s Guide.
//
///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
//
//  Our parser functor
//
///////////////////////////////////////////////////////////////////////////////
struct number_parser
{
    typedef int result_t;
    template <typename ScannerT>
    int
    operator()(ScannerT const& scan, result_t& result) const
    {
        if (scan.at_end())
            return -1;

        char ch = *scan;
        if (ch < ’0’ || ch > ’9’)
            return -1;

        result = 0;
        int len = 0;

        do
        {
            result = result*10 + int(ch - ’0’);
            ++len;
            ++scan;
        } while (!scan.at_end() && (ch = *scan, ch >= ’0’ && ch <= ’9’));

        return len;
    }
};

functor_parser<number_parser> number_parser_p;

///////////////////////////////////////////////////////////////////////////////
//
```

```
//  Our number parser functions
//
///////////////////////////////////////////////////////////////////////////////
bool
parse_number(char const* str, int& n)
{
    return parse(str, lexeme_d[number_parser_p[assign_a(n)]], space_p).full;
}

bool
parse_numbers(char const* str, std::vector<int>& n)
{
    return
        parse(
            str,
            lexeme_d[number_parser_p[push_back_a(n)]]
                >> *(’,’ >> lexeme_d[number_parser_p[push_back_a(n)]]),
            space_p
        ).full;
}

///////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tA number parser implemented as a functor for Spirit...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";

    cout << "Give me an integer number command\n";
    cout << "Commands:\n";
    cout << "  A <num> --> parses a single number\n";
    cout << "  B <num>, <num>, ... --> parses a series of numbers ";
    cout << "separated by commas\n";
    cout << "  Q --> quit\n\n";

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == ’q’ || str[0] == ’Q’)
            break;

        else if (str[0] == ’a’ || str[0] == ’A’)
        {
            int n;
            if (parse_number(str.c_str()+1, n))
            {
                cout << "-------------------------\n";
                cout << "Parsing succeeded\n";
                cout << str << " Parses OK: " << n << endl;
                cout << "-------------------------\n";
            }
            else
            {
                cout << "-------------------------\n";
                cout << "Parsing failed\n";
                cout << "-------------------------\n";
            }
        }

        else if (str[0] == 'b' || str[0] == 'B')
        {
            std::vector<int> n;
            if (parse_numbers(str.c_str()+1, n))
            {
                cout << "---------------------------\n";
                cout << "Parsing succeeded\n";
                int size = n.size();
                cout << str << " Parses OK: " << size << " number(s): " << n[0];
                for (int i = 1; i < size; ++i) {
                    cout << ", " << n[i];
                }
                cout << endl;
                cout << "---------------------------\n";
            }
            else
            {
                cout << "---------------------------\n";
                cout << "Parsing failed\n";
                cout << "---------------------------\n";
            }
        }

        else
        {
            cout << "---------------------------\n";
            cout << "Unrecognized command!!";
            cout << "---------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```cpp
/*=============================================================================
    Copyright (c) 2001-2003 Hartmut Kaiser
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
// This sample shows the usage of the list_p utility parser
//  1. parsing a simple ',' delimited list w/o item formatting
//  2. parsing a CSV list (comma separated values - strings, integers or reals)
//  3. parsing a token list (token separated values - strings, integers or
//     reals)
// with an action parser directly attached to the item part of the list_p
// generated parser

#include <string>
#include <iostream>
#include <cassert>

#include <boost/spirit/core.hpp>
#include <boost/spirit/utility/confix.hpp>
#include <boost/spirit/utility/lists.hpp>
#include <boost/spirit/utility/escape_char.hpp>

#include <iostream>
#include <cassert>
#include <string>
#include <vector>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
// actor, attached to the list_p parser
class list_actor
{
public:
    list_actor (std::vector<std::string> &vec_) : vec(vec_) {}

    // The following operator() is called by the action parser generated by
    // attaching this actor to a list_p generated list parser.

    template <typename ActionIterT>
    void operator() (ActionIterT const &first, ActionIterT const &last) const
    {
        vec.push_back(std::string(first, last-first));
    }

private:
    std::vector<std::string> &vec;
};

///////////////////////////////////////////////////////////////////////////////
// main entry point
int main ()
{
    // 1. parsing a simple ',' delimited list w/o item formatting
    char const*                 plist_wo_item = "element1,element2,element3";
    rule<>                      list_wo_item;
```

```cpp
    std::vector<std::string>    vec_list;

    list_wo_item =
            list_p[push_back_a(vec_list)]
        ;

    parse_info<> result = parse (plist_wo_item, list_wo_item);

    cout << "-------------------------------------------------------------------"
        << endl;

    if (result.hit)
    {
        cout
            << "Parsing simple list" << endl
            << "\t" << plist_wo_item << endl
            << "Parsed successfully!" << endl << endl;

        cout
            << "Actor was called " << (int)vec_list.size()
            << " times: " << endl;

        cout
            << "Results got from the list parser:" << endl;
        for (std::vector<std::string>::iterator it = vec_list.begin();
             it != vec_list.end(); ++it)
        {
            cout << *it << endl;
        }
    }
    else
    {
        cout << "Failed to parse simple list!" << endl;
    }

    cout << endl;

    // 2. parsing a CSV list (comma separated values - strings, integers or
    // reals)
    char const *plist_csv = "\"string\",\"string with an embedded \\\"\","
        "12345,0.12345e4,,2";
    rule<> list_csv, list_csv_item;
    std::vector<std::string> vec_item;

    vec_list.clear();

    list_csv_item =
        !(
                confix_p('\"', *c_escape_ch_p, '\"')
            |   longest_d[real_p | int_p]
        );

    list_csv =
            list_p(
                list_csv_item[push_back_a(vec_item)],
                ','
            )[push_back_a(vec_list)]
        ;

    result = parse (plist_csv, list_csv);

    cout << "-------------------------------------------------------------------"
```

```
        << endl;
    if (result.hit)
    {
        cout
            << "Parsing CSV list (comma separated values) " << endl
            << "\t" << plist_csv << endl
            << "Parsed successfully!" << endl << endl;

        if (result.full)
        {
            cout << "Matched " << (int)vec_list.size() <<
                " list elements (full list): " << endl;
        }
        else
        {
            cout << "Matched " << (int)vec_list.size() <<
                " list elements: " << endl;
        }

        cout << "The list parser matched:" << endl;
        for (std::vector<std::string>::iterator itl = vec_list.begin();
                itl != vec_list.end(); ++itl)
        {
            cout << *itl << endl;
        }

        cout << endl << "Item(s) got directly from the item parser:" << endl;
        for (std::vector<std::string>::iterator it = vec_item.begin();
                it != vec_item.end(); ++it)
        {
            cout << *it << endl;
        }

    }
    else
    {
        cout << "Failed to parse CSV list!" << endl;
    }

    cout << endl;

    // 3. parsing a token list (token separated values - strings, integers or
    // reals) with an action parser directly attached to the item part of the
    // list_p generated parser
    char const *plist_csv_direct = "\"string\"<par>\"string with an embedded "
        "\\\"\"<par>12345<par>0.12345e4";
    rule<> list_csv_direct, list_csv_direct_item;

    vec_list.clear();
    vec_item.clear();

    // Note: the list parser is here generated through the list_p.direct()
    // generator function. This inhibits re-attachment of the item_actor_direct
    // during parser construction (see: comment in utility/lists.hpp)
    list_csv_direct_item =
            confix_p(’\"’, *c_escape_ch_p, ’\"’)
        |   longest_d[real_p | int_p]
        ;

    list_csv_direct =
            list_p.direct(
                (*list_csv_direct_item)[list_actor(vec_item)],
```

```
                "<par>"
            )[list_actor(vec_list)]
        ;

    result = parse (plist_csv_direct, list_csv_direct);

    cout << "-----------------------------------------------------------------"
        << endl;
    if (result.hit)
    {
        cout
            << "Parsing CSV list (comma separated values)" << endl
            << "The list parser was generated with ’list_p.direct()’" << endl
            << "\t" << plist_csv_direct << endl
            << "Parsed successfully!" << endl << endl;

        if (result.full)
        {
            cout << "Matched " << vec_list.size() <<
                " list elements (full list): " << endl;
        }
        else
        {
            cout << "Matched " << vec_list.size() <<
                " list elements: " << endl;
        }

        cout << "The list parser matched:" << endl;
        for (std::vector<std::string>::iterator itl = vec_list.begin();
                itl != vec_list.end(); ++itl)
        {
            cout << *itl << endl;
        }

        cout << endl << "Items got directly from the item parser:" << endl;
        for (std::vector<std::string>::iterator it = vec_item.begin();
                it != vec_item.end(); ++it)
        {
            cout << *it << endl;
        }

    }
    else
    {
        cout << "Failed to parse CSV list!" << endl;
    }

    cout << endl;

    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  HTML/XML like tag matching grammar
//  Demonstrates phoenix and closures and parametric parsers
//  This is discussed in the "Closures" chapter in the Spirit User’s Guide.
//
//  [ JDG 6/30/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/attribute.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

///////////////////////////////////////////////////////////////////////////////
//
//  HTML/XML like tag matching grammar
//
///////////////////////////////////////////////////////////////////////////////
struct tags_closure : boost::spirit::closure<tags_closure, string>
{
    member1 tag;
};

struct tags : public grammar<tags>
{
    template <typename ScannerT>
    struct definition {

        definition(tags const& /*self*/)
        {
            element = start_tag >> *element >> end_tag;

            start_tag =
                    ’<’
                >>  lexeme_d
                    [
                        (+alpha_p)
                        [
                            //  construct string from arg1 and arg2 lazily
                            //  and assign to element.tag

                            element.tag = construct_<string>(arg1, arg2)
                        ]
                    ]
                >> ’>’;

            end_tag = "</" >> f_str_p(element.tag) >> ’>’;
        }
```

```
        rule<ScannerT, tags_closure::context_t> element;
        rule<ScannerT> start_tag, end_tag;

        rule<ScannerT, tags_closure::context_t> const&
        start() const { return element; }
    };
};

///////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tHTML/XML like tag matching parser demo \n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an HTML/XML like nested tag input...or [q or Q] to quit\n\n";
    cout << "Example: <html><head></head><body></body></html>\n\n";

    tags p;    //  Our parser

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == ’q’ || str[0] == ’Q’)
            break;

        parse_info<> info = parse(str.c_str(), p, space_p);

        if (info.full)
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "stopped at: \": " << info.stop << "\"\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*==============================================================================
    Copyright (c) 2003 Vaclav Vesely
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
==============================================================================*/
//
//  This example demonstrates no_actions_d directive.
//
//  The no_actions_d directive ensures, that semantic actions of the inner
//  parser would NOT be invoked. See the no_actions_scanner in the Scanner
//  and Parsing chapter in the User’s Guide.
//
//------------------------------------------------------------------------------

#include <cassert>
#include <iostream>
#include <boost/cstdlib.hpp>
#include <boost/spirit/core.hpp>

using namespace std;
using namespace boost;
using namespace spirit;

//------------------------------------------------------------------------------

int main()
{
    // To use the rule in the no_action_d directive we must declare it with
    // the no_actions_scanner scanner
    rule<no_actions_scanner<>::type> r;

    int i(0);

    // r is the rule with the semantic action
    r = int_p[assign_a(i)];

    parse_info<> info = parse(
        "1",

        no_actions_d
        [
            r
        ]
    );

    assert(info.full);
    // Check, that the action hasn’t been invoked
    assert(i == 0);

    return exit_success;
}

//------------------------------------------------------------------------------
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  This sample demontrates a parser for a comma separated list of numbers
//  This is discussed in the "Quick Start" chapter in the Spirit User’s Guide.
//
//  [ JDG 5/10/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/actor/push_back_actor.hpp>
#include <iostream>
#include <vector>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
//
//  Our comma separated list parser
//
///////////////////////////////////////////////////////////////////////////////
bool
parse_numbers(char const* str, vector<double>& v)
{
    return parse(str,

        //  Begin grammar
        (
            real_p[push_back_a(v)] >> *(’,’ >> real_p[push_back_a(v)])
        )
        ,
        //  End grammar

        space_p).full;
}

////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tA comma separated list parser for Spirit...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";

    cout << "Give me a comma separated list of numbers.\n";
    cout << "The numbers will be inserted in a vector of numbers\n";
    cout << "Type [q or Q] to quit\n\n";
```

```
    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == ’q’ || str[0] == ’Q’)
            break;

        vector<double> v;
        if (parse_numbers(str.c_str(), v))
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << str << " Parses OK: " << endl;

            for (vector<double>::size_type i = 0; i < v.size(); ++i)
                cout << i << ": " << v[i] << endl;

            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2001-2003 Daniel Nuffer
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  Demonstrates parse trees. This is discussed in the
//  "Trees" chapter in the Spirit User’s Guide.
//
///////////////////////////////////////////////////////////////////////////////
#define BOOST_SPIRIT_DUMP_PARSETREE_AS_XML

#include <boost/spirit/core.hpp>
#include <boost/spirit/tree/parse_tree.hpp>

#include <iostream>
#include <stack>
#include <functional>
#include <string>

#ifdef BOOST_SPIRIT_DUMP_PARSETREE_AS_XML
#include <boost/spirit/tree/tree_to_xml.hpp>
#include <map>
#endif

////////////////////////////////////////////////////////////////////////////
// This example shows how to use a parse tree
using namespace std;
using namespace boost::spirit;

// Here’s some typedefs to simplify things
typedef char const*         iterator_t;
typedef tree_match<iterator_t> parse_tree_match_t;
typedef parse_tree_match_t::const_tree_iterator iter_t;

typedef pt_match_policy<iterator_t> match_policy_t;
typedef scanner_policies<iteration_policy, match_policy_t, action_policy> scanne
r_policy_t;
typedef scanner<iterator_t, scanner_policy_t> scanner_t;
typedef rule<scanner_t> rule_t;

//  grammar rules
rule_t expression, term, factor, integer;

////////////////////////////////////////////////////////////////////////////
// Here’s the function prototypes that we’ll use.  One function for each
// grammar rule.
long evaluate(const tree_parse_info<>& info);
long eval_expression(iter_t const& i);
long eval_term(iter_t const& i);
long eval_factor(iter_t const& i);
long eval_integer(iter_t const& i);

long evaluate(const tree_parse_info<>& info)
{
    return eval_expression(info.trees.begin());
}
```

```
// i should be pointing to a node created by the expression rule
long eval_expression(iter_t const& i)
{
    parser_id id = i->value.id();
    assert(id == expression.id()); // check the id

    // first child points to a term, so call eval_term on it
    iter_t chi = i->children.begin();
    long lhs = eval_term(chi);
    for (++chi; chi != i->children.end(); ++chi)
    {
        // next node points to the operator.  The text of the operator is
        // stored in value (a vector<char>)
        char op = *(chi->value.begin());
        ++chi;
        long rhs = eval_term(chi);
        if (op == ’+’)
            lhs += rhs;
        else if (op == ’-’)
            lhs -= rhs;
        else
            assert(0);
    }
    return lhs;
}

long eval_term(iter_t const& i)
{
    parser_id id = i->value.id();
    assert(id == term.id());

    iter_t chi = i->children.begin();
    long lhs = eval_factor(chi);
    for (++chi; chi != i->children.end(); ++chi)
    {
        char op = *(chi->value.begin());
        ++chi;
        long rhs = eval_factor(chi);
        if (op == ’*’)
            lhs *= rhs;
        else if (op == ’/’)
            lhs /= rhs;
        else
            assert(0);
    }
    return lhs;
}

long eval_factor(iter_t const& i)
{
    parser_id id = i->value.id();
    assert(id == factor.id());

    iter_t chi = i->children.begin();
    id = chi->value.id();
    if (id == integer.id())
        return eval_integer(chi->children.begin());
    else if (*(chi->value.begin()) == ’(’)
    {
        ++chi;
        return eval_expression(chi);
    }
    else if (*(chi->value.begin()) == '-')
    {
        ++chi;
        return -eval_factor(chi);
    }
    else
    {
        assert(0);
        return 0;
    }
}

long eval_integer(iter_t const& i)
{
    // extract integer (not always delimited by '\0')
    string integer(i->value.begin(), i->value.end());

    return strtol(integer.c_str(), 0, 10);
}

////////////////////////////////////////////////////////////////////////////
int
main()
{

    //  Start grammar definition
    integer     =   lexeme_d[ token_node_d[ (!ch_p('-') >> +digit_p) ] ];
    factor      =   integer
                |   '(' >> expression >> ')'
                |   ('-' >> factor);
    term        =   factor >>
                    *(  ('*' >> factor)
                      | ('/' >> factor)
                    );
    expression  =   term >>
                    *(  ('+' >> term)
                      | ('-' >> term)
                    );
    //  End grammar definition


    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tThe simplest working calculator...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        const char* first = str.c_str();

        tree_parse_info<> info = pt_parse(first, expression);

        if (info.full)
        {
#if defined(BOOST_SPIRIT_DUMP_PARSETREE_AS_XML)
            // dump parse tree as XML
            std::map<parser_id, std::string> rule_names;
            rule_names[integer.id()] = "integer";
```

```
            rule_names[factor.id()] = "factor";
            rule_names[term.id()] = "term";
            rule_names[expression.id()] = "expression";
            tree_to_xml(cout, info.trees, first, rule_names);
#endif

            // print the result
            cout << "parsing succeeded\n";
            cout << "result = " << evaluate(info) << "\n\n";
        }
        else
        {
            cout << "parsing failed\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2003 Vaclav Vesely
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
//
//  This example demonstrates usage of the parser_context template with
//  an explicit argument to declare rules with match results different from
//  nil_t. For better understanding, you should read the chapter "In-depth:
//  The Parser Context" in the documentation.
//
//  The default context of non-terminals is the parser_context.
//  The parser_context is a template with one argument AttrT, which is the type
//  of match attribute.
//
//  In this example int_rule is declared as rule with int match attribute’s
//  type, so in int_rule variable we can hold any parser, which returns int
//  value. For example int_p or bin_p. And the most important is that we can
//  use returned value in the semantic action binded to the int_rule.
//
//-----------------------------------------------------------------------------
#include <iostream>
#include <boost/cstdlib.hpp>
#include <boost/spirit/phoenix.hpp>
#include <boost/spirit/core.hpp>

using namespace std;
using namespace boost;
using namespace phoenix;
using namespace spirit;

//-----------------------------------------------------------------------------

int main()
{
    rule<parser_context<int> > int_rule = int_p;

    parse(
        "123",
        // Using a returned value in the semantic action
        int_rule[cout << arg1 << endl]
    );

    return exit_success;
}

//-----------------------------------------------------------------------------
```

```
/*==============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
==============================================================================*/
/////////////////////////////////////////////////////////////////////////////
//
//  Full calculator example demonstrating Phoenix
//  This is discussed in the "Closures" chapter in the Spirit User’s Guide.
//
//  [ JDG 6/29/2002 ]
//
/////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/attribute.hpp>
#include <iostream>
#include <string>

/////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

/////////////////////////////////////////////////////////////////////////////
//
//  Our calculator grammar using phoenix to do the semantics
//
//  Note:   The top rule propagates the expression result (value) upwards
//          to the calculator grammar self.val closure member which is
//          then visible outside the grammar (i.e. since self.val is the
//          member1 of the closure, it becomes the attribute passed by
//          the calculator to an attached semantic action. See the
//          driver code that uses the calculator below).
//
/////////////////////////////////////////////////////////////////////////////
struct calc_closure : boost::spirit::closure<calc_closure, double>
{
    member1 val;
};

struct calculator : public grammar<calculator, calc_closure::context_t>
{
    template <typename ScannerT>
    struct definition
    {
        definition(calculator const& self)
        {
            top = expression[self.val = arg1];

            expression
                =   term[expression.val = arg1]
                    >> *(   (’+’ >> term[expression.val += arg1])
                        |   (’-’ >> term[expression.val -= arg1])
                        )
                ;

            term
                =   factor[term.val = arg1]
                    >> *(   (’*’ >> factor[term.val *= arg1])
```

```
                        |   (’/’ >> factor[term.val /= arg1])
                        )
                ;

            factor
                =   ureal_p[factor.val = arg1]
                |   ’(’ >> expression[factor.val = arg1] >> ’)’
                |   (’-’ >> factor[factor.val = -arg1])
                |   (’+’ >> factor[factor.val = arg1])
                ;
        }

        typedef rule<ScannerT, calc_closure::context_t> rule_t;
        rule_t expression, term, factor;
        rule<ScannerT> top;

        rule<ScannerT> const&
        start() const { return top; }
    };
};

/////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
/////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tExpression parser using Phoenix...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";

    calculator calc;    //  Our parser

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == ’q’ || str[0] == ’Q’)
            break;

        double n = 0;
        parse_info<> info = parse(str.c_str(), calc[var(n) = arg1], space_p);

        //  calc[var(n) = arg1] invokes the calculator and extracts
        //  the result of the computation. See calculator grammar
        //  note above.

        if (info.full)
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << "result = " << n << endl;
            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "stopped at: \": " << info.stop << "\"\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Hartmut Kaiser
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//  This example shows the usage of the refactoring parser family parsers
//  See the "Refactoring Parsers" chapter in the User's Guide.

#include <iostream>
#include <string>

#include <boost/spirit/core.hpp>
#include <boost/spirit/meta/refactoring.hpp>

///////////////////////////////////////////////////////////////////////////////
// used namespaces
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
// actor, used by the refactor_action_p test
struct refactor_action_actor
{
    refactor_action_actor (std::string &str_) : str(str_) {}

    template <typename IteratorT>
    void operator() (IteratorT const &first, IteratorT const &last) const
    {
        str = std::string(first, last-first);
    }

    std::string &str;
};

///////////////////////////////////////////////////////////////////////////////
// main entry point
int main()
{
    parse_info<> result;
    char const *test_string = "Some string followed by a newline\n";

///////////////////////////////////////////////////////////////////////////////
//
//  1. Testing the refactor_unary_d parser
//
//  The following test should successfully parse the test string, because the
//
//      refactor_unary_d[
//          *anychar_p - '\n'
//      ]
//
//  is refactored into
//
//      *(anychar_p - '\n').
//
///////////////////////////////////////////////////////////////////////////////

    result = parse(test_string, refactor_unary_d[*anychar_p - '\n'] >> '\n');
```

```
    if (result.full)
    {
        cout << "Successfully refactored an unary!" << endl;
    }
    else
    {
        cout << "Failed to refactor an unary!" << endl;
    }

//  Parsing the same test string without refactoring fails, because the
//  *anychar_p eats up all the input up to the end of the input string.

    result = parse(test_string, (*anychar_p - '\n') >> '\n');

    if (result.full)
    {
        cout
            << "Successfully parsed test string (should not happen)!"
            << endl;
    }
    else
    {
        cout
            << "Correctly failed parsing the test string (without refactoring)!"
            << endl;
    }
    cout << endl;

///////////////////////////////////////////////////////////////////////////////
//
//  2. Testing the refactor_action_d parser
//
//  The following test should successfully parse the test string, because the
//
//      refactor_action_d[
//          (*(anychar_p - '$'))[refactor_action_actor(str)] >> '$'
//      ]
//
//  is refactored into
//
//      (*(anychar_p - '$') >> '$')[refactor_action_actor(str)].
//
///////////////////////////////////////////////////////////////////////////////

    std::string str;
    char const *test_string2 = "Some test string ending with a $";

    result =
        parse(test_string2,
            refactor_action_d[
                (*(anychar_p - '$'))[refactor_action_actor(str)] >> '$'
            ]
        );

    if (result.full && str == std::string(test_string2))
    {
        cout << "Successfully refactored an action!" << endl;
        cout << "Parsed: \"" << str << "\"" << endl;
    }
    else
    {
        cout << "Failed to refactor an action!" << endl;
    }

//  Parsing the same test string without refactoring fails, because the
//  the attached actor gets called only for the first part of the string
//  (without the '$')

    result =
        parse(test_string2,
            (*(anychar_p - '$'))[refactor_action_actor(str)] >> '$'
        );

    if (result.full && str == std::string(test_string2))
    {
        cout << "Successfully parsed test string!" << endl;
        cout << "Parsed: \"" << str << "\"" << endl;
    }
    else
    {
        cout
            << "Correctly failed parsing the test string (without refactoring)!"
            << endl;
        cout << "Parsed instead: \"" << str << "\"" << endl;
    }
    cout << endl;

///////////////////////////////////////////////////////////////////////////////
//
//  3. Testing the refactor_action_d parser with an embedded (nested)
//  refactor_unary_p parser
//
//  The following test should successfully parse the test string, because the
//
//      refactor_action_unary_d[
//          ((*anychar_p)[refactor_action_actor(str)] - '$')
//      ] >> '$'
//
//  is refactored into
//
//      (*(anychar_p - '$'))[refactor_action_actor(str)] >> '$'.
//
///////////////////////////////////////////////////////////////////////////////

    const refactor_action_gen<refactor_unary_gen<> > refactor_action_unary_d =
        refactor_action_gen<refactor_unary_gen<> >(refactor_unary_d);

    result =
        parse(test_string2,
            refactor_action_unary_d[
                ((*anychar_p)[refactor_action_actor(str)] - '$')
            ] >> '$'
        );

    if (result.full)
    {
        cout
            << "Successfully refactored an action attached to an unary!"
            << endl;
        cout << "Parsed: \"" << str << "\"" << endl;
    }
    else
    {
```

```
        cout << "Failed to refactor an action!" << endl;
    }

//  Parsing the same test string without refactoring fails, because the
//  anychar_p eats up all the input up to the end of the string

    result =
        parse(test_string2,
            ((*anychar_p)[refactor_action_actor(str)] - '$') >> '$'
        );

    if (result.full)
    {
        cout << "Successfully parsed test string!" << endl;
        cout << "Parsed: \"" << str << "\"" << endl;
    }
    else
    {
        cout
            << "Correctly failed parsing the test string (without refactoring)!"
            << endl;
        cout << "Parsed instead: \"" << str << "\"" << endl;
    }
    cout << endl;

    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Hartmut Kaiser
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  Demonstrate regular expression parser objects
//  See the "Regular Expression Parser" chapter in the User's Guide.
//
//  This sample requires an installed version of the boost regex library
//  (http://www.boost.org) The sample was tested with boost V1.28.0
//
///////////////////////////////////////////////////////////////////////////////
#include <string>
#include <iostream>

#include <boost/spirit/core.hpp>

///////////////////////////////////////////////////////////////////////////////
//
//  The following header must be included, if regular expression support is
//  required for Spirit.
//
//  The BOOST_SPIRIT_NO_REGEX_LIB PP constant should be defined, if you're
//  using the Boost.Regex library from one translation unit only. Otherwise
//  you have to link with the Boost.Regex library as defined in the related
//  documentation (see. http://www.boost.org).
//
///////////////////////////////////////////////////////////////////////////////
#define BOOST_SPIRIT_NO_REGEX_LIB
#include <boost/spirit/utility/regex.hpp>

///////////////////////////////////////////////////////////////////////////////
//  used namespaces
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
// main entry point
int main (int argc, char *argv[])
{
    const char *ptest = "123 E 456";
    const char *prx = "[1-9]+[[:space:]]*E[[:space:]]*";

    cout << "Parse " << ptest << " against regular expression: " << prx
        << endl;

    // 1. direct use of rxlit<>
    rxstrlit<> regexpr(prx);
    parse_info<> result;
    string str;

    result = parse (ptest, regexpr[assign(str)]);
    if (result.hit)
    {
        cout << "Parsed regular expression successfully!" << endl;
        cout << "Matched (" << (int)result.length << ") characters: ";
        cout << "\"" << str << "\"" << endl;
```

```
    }
    else
    {
        cout << "Failed to parse regular expression!" << endl;
    }
    cout << endl;

    // 2. use of regex_p predefined parser object
    str.empty();
    result = parse (ptest, regex_p(prx)[assign(str)]);
    if (result.hit)
    {
        cout << "Parsed regular expression successfully!" << endl;
        cout << "Matched (" << (int)result.length << ") characters: ";
        cout << "\"" << str << "\"" << endl;
    }
    else
    {
        cout << "Failed to parse regular expression!" << endl;
    }
    cout << endl;

    // 3. test the regression reported by Grzegorz Marcin Koczyk (gkoczyk@echost
ar.pl)
    string str1;
    string str2;
    char const *ptest1 = "Token whatever \nToken";

    result = parse(ptest1, rxstrlit<>("Token")[assign(str1)]
        >> rxstrlit<>("Token")[assign(str2)]);

    if (!result.hit)
        cout << "Parsed regular expression successfully!" << endl;
    else
        cout << "Failed to parse regular expression!" << endl;

    cout << endl;

    return 0;
}
```

```cpp
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  A Roman Numerals Parser (demonstrating the symbol table). This is
//  discussed in the "Symbols" chapter in the Spirit User's Guide.
//
//  [ JDG 8/22/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/symbols/symbols.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
//
//  Parse roman hundreds (100..900) numerals using the symbol table.
//  Notice that the data associated with each slot is passed
//  to attached semantic actions.
//
///////////////////////////////////////////////////////////////////////////////
struct hundreds : symbols<unsigned>
{
    hundreds()
    {
        add
            ("C"    , 100)
            ("CC"   , 200)
            ("CCC"  , 300)
            ("CD"   , 400)
            ("D"    , 500)
            ("DC"   , 600)
            ("DCC"  , 700)
            ("DCCC" , 800)
            ("CM"   , 900)
        ;
    }

} hundreds_p;

///////////////////////////////////////////////////////////////////////////////
//
//  Parse roman tens (10..90) numerals using the symbol table.
//
///////////////////////////////////////////////////////////////////////////////
struct tens : symbols<unsigned>
{
    tens()
    {
        add
            ("X"    , 10)
```

```cpp
            ("XX"   , 20)
            ("XXX"  , 30)
            ("XL"   , 40)
            ("L"    , 50)
            ("LX"   , 60)
            ("LXX"  , 70)
            ("LXXX" , 80)
            ("XC"   , 90)
        ;
    }

} tens_p;

///////////////////////////////////////////////////////////////////////////////
//
//  Parse roman ones (1..9) numerals using the symbol table.
//
///////////////////////////////////////////////////////////////////////////////
struct ones : symbols<unsigned>
{
    ones()
    {
        add
            ("I"    , 1)
            ("II"   , 2)
            ("III"  , 3)
            ("IV"   , 4)
            ("V"    , 5)
            ("VI"   , 6)
            ("VII"  , 7)
            ("VIII" , 8)
            ("IX"   , 9)
        ;
    }

} ones_p;

///////////////////////////////////////////////////////////////////////////////
//
//  Semantic actions
//
///////////////////////////////////////////////////////////////////////////////
struct add_1000
{
    add_1000(unsigned& r_) : r(r_) {}
    void operator()(char) const { r += 1000; }
    unsigned& r;
};

struct add_roman
{
    add_roman(unsigned& r_) : r(r_) {}
    void operator()(unsigned n) const { r += n; }
    unsigned& r;
};

///////////////////////////////////////////////////////////////////////////////
//
//  roman (numerals) grammar
//
///////////////////////////////////////////////////////////////////////////////
struct roman : public grammar<roman>
```

```
{
    template <typename ScannerT>
    struct definition
    {
        definition(roman const& self)
        {
            first
                =   +ch_p(’M’)  [add_1000(self.r)]
                ||  hundreds_p  [add_roman(self.r)]
                ||  tens_p      [add_roman(self.r)]
                ||  ones_p      [add_roman(self.r)];

            //  Note the use of the || operator. The expression
            //  a || b reads match a or b and in sequence. Try
            //  defining the roman numerals grammar in YACC or
            //  PCCTS. Spirit rules! :-)
        }

        rule<ScannerT> first;
        rule<ScannerT> const&
        start() const { return first; }
    };

    roman(unsigned& r_) : r(r_) {}
    unsigned& r;
};

///////////////////////////////////////////////////////////////////////////////
//
//  Main driver code
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tRoman Numerals Parser\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type a Roman Numeral ...or [q or Q] to quit\n\n";

    //  Start grammar definition

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == ’q’ || str[0] == ’Q’)
            break;

        unsigned n = 0;
        roman roman_p(n);
        if (parse(str.c_str(), roman_p).full)
        {
            cout << "parsing succeeded\n";
            cout << "result = " << n << "\n\n";
        }
        else
        {
            cout << "parsing failed\n\n";
        }
    }

    cout << "Bye... :–) \n\n";
```

```
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  This sample demontrates a parser for a comma separated list of numbers
//  This is the phoenix version of number_list.cpp.
//  This is discussed in the "Phoenix" chapter in the Spirit User’s Guide.
//
//  [ JDG 1/12/2004 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/phoenix/primitives.hpp>
#include <boost/spirit/phoenix/operators.hpp>
#include <boost/spirit/phoenix/functions.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

///////////////////////////////////////////////////////////////////////////////
//
//  Our comma separated list parser
//
///////////////////////////////////////////////////////////////////////////////
struct push_back_impl
{
    template <typename Container, typename Item>
    struct result
    {
        typedef void type;
    };

    template <typename Container, typename Item>
    void operator()(Container& c, Item const& item) const
    {
        c.push_back(item);
    }
};

function<push_back_impl> const push_back = push_back_impl();

bool
parse_numbers(char const* str, vector<double>& v)
{
    return parse(str,

        //  Begin grammar
        (
            real_p[push_back(var(v), arg1)]
                >> *(’,’ >> real_p[push_back(var(v), arg1)])
        )
        ,
```

```
        //  End grammar

        space_p).full;
}

////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tA comma separated list parser for Spirit...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";

    cout << "Give me a comma separated list of numbers.\n";
    cout << "The numbers will be inserted in a vector of numbers\n";
    cout << "Type [q or Q] to quit\n\n";

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == ’q’ || str[0] == ’Q’)
            break;

        vector<double> v;
        if (parse_numbers(str.c_str(), v))
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << str << " Parses OK: " << endl;

            for (vector<double>::size_type i = 0; i < v.size(); ++i)
                cout << i << ": " << v[i] << endl;

            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```cpp
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  This sample demontrates a parser for a comma separated list of identifiers
//  This is a variation of stuff_vector.cpp.
//  This is discussed in the "Phoenix" chapter in the Spirit User's Guide.
//
//  [ JDG 1/12/2004 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/phoenix/primitives.hpp>
#include <boost/spirit/phoenix/operators.hpp>
#include <boost/spirit/phoenix/functions.hpp>
#include <boost/spirit/phoenix/casts.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

///////////////////////////////////////////////////////////////////////////////
//
//  Our comma separated list parser
//
///////////////////////////////////////////////////////////////////////////////
struct push_back_impl
{
    template <typename Container, typename Item>
    struct result
    {
        typedef void type;
    };

    template <typename Container, typename Item>
    void operator()(Container& c, Item const& item) const
    {
        c.push_back(item);
    }
};

function<push_back_impl> const push_back = push_back_impl();

bool
parse_identifiers(char const* str, vector<std::string>& v)
{
    return parse(str,

        //  Begin grammar
        (
            (+alpha_p)
            [
                push_back(var(v), construct_<std::string>(arg1, arg2))
```

```cpp
            ]
            >>
            *(',' >>
                (+alpha_p)
                [
                    push_back(var(v), construct_<std::string>(arg1, arg2))
                ]
            )
        )
        ,
        //  End grammar

        space_p).full;
}

///////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tA comma separated list parser for Spirit...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";

    cout << "Give me a comma separated list of identifiers.\n";
    cout << "An identifier is comprised of one or more alphabetic characters.\n";
    cout << "The identifiers will be inserted in a vector of numbers\n";
    cout << "Type [q or Q] to quit\n\n";

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        vector<std::string> v;
        if (parse_identifiers(str.c_str(), v))
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << str << " Parses OK: " << endl;

            for (vector<std::string>::size_type i = 0; i < v.size(); ++i)
                cout << i << ": " << v[i] << endl;

            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  This calculator example demontrates the use of subrules.
//  This is discussed in the "Subrule" chapter in the Spirit User’s Guide.
//
//  [ JDG 4/11/2002 ]
//
///////////////////////////////////////////////////////////////////////////////

//#define BOOST_SPIRIT_DEBUG        // define this for debug output

#include <boost/spirit/core.hpp>
#include <iostream>
#include <string>

using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
//
//  Semantic actions
//
///////////////////////////////////////////////////////////////////////////////
namespace
{
    void    do_int(char const* str, char const* end)
    {
        string  s(str, end);
        cout << "PUSH(" << s << ’)’ << endl;
    }

    void    do_add(char const*, char const*)    { cout << "ADD\n"; }
    void    do_subt(char const*, char const*)   { cout << "SUBTRACT\n"; }
    void    do_mult(char const*, char const*)   { cout << "MULTIPLY\n"; }
    void    do_div(char const*, char const*)    { cout << "DIVIDE\n"; }
    void    do_neg(char const*, char const*)    { cout << "NEGATE\n"; }
}

///////////////////////////////////////////////////////////////////////////////
//
//  Our calculator grammar (using subrules)
//
///////////////////////////////////////////////////////////////////////////////
struct calculator : public grammar<calculator>
{
    template <typename ScannerT>
    struct definition {

        definition(calculator const& /*self*/)
        {
            first = (

                expression =
                    term
```

```
                    >> *(   (’+’ >> term)[&do_add]
                        |   (’-’ >> term)[&do_subt]
                        )
                ,

                term =
                    factor
                    >> *(   (’*’ >> factor)[&do_mult]
                        |   (’/’ >> factor)[&do_div]
                        )
                ,

                factor
                    =   lexeme_d[(+digit_p)[&do_int]]
                    |   ’(’ >> expression >> ’)’
                    |   (’-’ >> factor)[&do_neg]
                    |   (’+’ >> factor)
            );

            BOOST_SPIRIT_DEBUG_NODE(first);
            BOOST_SPIRIT_DEBUG_NODE(expression);
            BOOST_SPIRIT_DEBUG_NODE(term);
            BOOST_SPIRIT_DEBUG_NODE(factor);
        }

        subrule<0>  expression;
        subrule<1>  term;
        subrule<2>  factor;

        rule<ScannerT> first;
        rule<ScannerT> const&
        start() const { return first; }
    };
};

///////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tA calculator using subrules...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";

    calculator calc;    //  Our parser

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == ’q’ || str[0] == ’Q’)
            break;

        parse_info<> info = parse(str.c_str(), calc, space_p);

        if (info.full)
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "stopped at: \": " << info.stop << "\"\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  A parser for summing a list of numbers. Demonstrating phoenix
//  This is discussed in the "Phoenix" chapter in the Spirit User’s Guide.
//
//  [ JDG 6/28/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/phoenix/primitives.hpp>
#include <boost/spirit/phoenix/operators.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

///////////////////////////////////////////////////////////////////////////////
//
//  Our adder
//
///////////////////////////////////////////////////////////////////////////////
template <typename IteratorT>
bool adder(IteratorT first, IteratorT last, double& n)
{
    return parse(first, last,

        //  Begin grammar
        (
            real_p[var(n) = arg1] >> *(’,’ >> real_p[var(n) += arg1])
        )
        ,
        //  End grammar

        space_p).full;
}

////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tA parser for summing a list of numbers...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";

    cout << "Give me a comma separated list of numbers.\n";
    cout << "The numbers are added using Phoenix.\n";
    cout << "Type [q or Q] to quit\n\n";
```

```
    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == ’q’ || str[0] == ’Q’)
            break;

        double n;
        if (adder(str.begin(), str.end(), n))
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << str << " Parses OK: " << endl;

            cout << "sum = " << n;
            cout << "\n-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  A parser for a real number parser that parses thousands separated numbers
//  with at most two decimal places and no exponent. This is discussed in the
//  "Numerics" chapter in the Spirit User’s Guide.
//
//  [ JDG 12/16/2003 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/actor/assign_actor.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

template <typename T>
struct ts_real_parser_policies : public ureal_parser_policies<T>
{
    //  These policies can be used to parse thousand separated
    //  numbers with at most 2 decimal digits after the decimal
    //  point. e.g. 123,456,789.01

    typedef uint_parser<int, 10, 1, 2>  uint2_t;
    typedef uint_parser<T, 10, 1, -1>   uint_parser_t;
    typedef int_parser<int, 10, 1, -1>  int_parser_t;

    //////////////////////////////////  2 decimal places Max
    template <typename ScannerT>
    static typename parser_result<uint2_t, ScannerT>::type
    parse_frac_n(ScannerT& scan)
    { return uint2_t().parse(scan); }

    //////////////////////////////////  No exponent
    template <typename ScannerT>
    static typename parser_result<chlit<>, ScannerT>::type
    parse_exp(ScannerT& scan)
    { return scan.no_match(); }

    //////////////////////////////////  No exponent
    template <typename ScannerT>
    static typename parser_result<int_parser_t, ScannerT>::type
    parse_exp_n(ScannerT& scan)
    { return scan.no_match(); }

    //////////////////////////////////  Thousands separated numbers
    template <typename ScannerT>
    static typename parser_result<uint_parser_t, ScannerT>::type
    parse_n(ScannerT& scan)
    {
        typedef typename parser_result<uint_parser_t, ScannerT>::type RT;
        static uint_parser<unsigned, 10, 1, 3> uint3_p;
```

```
        static uint_parser<unsigned, 10, 3, 3> uint3_3_p;
        if (RT hit = uint3_p.parse(scan))
        {
            T n;
            typedef typename ScannerT::iterator_t iterator_t;
            iterator_t save = scan.first;
            while (match<> next = (’,’ >> uint3_3_p[assign_a(n)]).parse(scan))
            {
                hit.value((hit.value() * 1000) + n);
                scan.concat_match(hit, next);
                save = scan.first;
            }
            scan.first = save;
            return hit;

            // Note: On erroneous input such as "123,45", the result should
            // be a partial match "123". ’save’ is used to makes sure that
            // the scanner position is placed at the last *valid* parse
            // position.
        }
        return scan.no_match();
    }
};

real_parser<double, ts_real_parser_policies<double> > const
    ts_real_p = real_parser<double, ts_real_parser_policies<double> >();

////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tA real number parser that parses thousands separated\n";
    cout << "\t\tnumbers with at most two decimal places and no exponent...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";

    cout << "Give me a number.\n";
    cout << "Type [q or Q] to quit\n\n";

    string str;
    double n;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == ’q’ || str[0] == ’Q’)
            break;

        if (parse(str.c_str(), ts_real_p[assign_a(n)]).full)
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << str << " Parses OK: " << endl;
            cout << "n=" << n << endl;
            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "--------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2003 Vaclav Vesely
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
#include <cassert>
#include <iostream>
#include <boost/cstdlib.hpp>
#include <boost/spirit/core.hpp>
#include <boost/spirit/utility/distinct.hpp>

using namespace std;
using namespace boost;
using namespace spirit;

// keyword_p for C++
// (for basic usage instead of std_p)
const distinct_parser<> keyword_p("0−9a−zA−Z_");

// keyword_d for C++
// (for mor intricate usage, for example together with symbol tables)
const distinct_directive<> keyword_d("0−9a−zA−Z_");

struct my_grammar: public grammar<my_grammar>
{
    template <typename ScannerT>
    struct definition
    {
        typedef rule<ScannerT> rule_t;

        definition(my_grammar const& self)
        {
            top
                =
                    keyword_p("declare") // use keyword_p instead of std_p
                >>  !ch_p(’:’)
                >>  keyword_d[str_p("ident")] // use keyword_d
                ;
        }

        rule_t top;

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

int main()
{
    my_grammar gram;
    parse_info<> info;

    info = parse("declare ident", gram, space_p);
    assert(info.full); // valid input

    info = parse("declare: ident", gram, space_p);
    assert(info.full); // valid input
```

```

    info = parse("declareident", gram, space_p);
    assert(!info.hit); // invalid input

    return exit_success;
}
```

```
/*=============================================================================
    Copyright (c) 2003 Vaclav Vesely
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
#include <cassert>
#include <iostream>
#include <boost/cstdlib.hpp>
#include <boost/spirit/core.hpp>
#include <boost/spirit/utility/distinct.hpp>

using namespace std;
using namespace boost;
using namespace spirit;

struct my_grammar: public grammar<my_grammar>
{
    template <typename ScannerT>
    struct definition
    {
        typedef rule<ScannerT> rule_t;

        // keyword_p for ASN.1
        dynamic_distinct_parser<ScannerT> keyword_p;

        definition(my_grammar const& self)
        :   keyword_p(alnum_p | ('-' >> ~ch_p('-'))) // ASN.1 has quite complex
naming rules
        {
            top
                =
                    keyword_p("asn-declare") // use keyword_p instead of std_p
                >>  !str_p("--")
                >>  keyword_p("ident")
                ;
        }

        rule_t top;

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

int main()
{
    my_grammar gram;
    parse_info<> info;

    info = parse("asn-declare ident", gram, space_p);
    assert(info.full); // valid input

    info = parse("asn-declare--ident", gram, space_p);
    assert(info.full); // valid input

    info = parse("asn-declare-ident", gram, space_p);
    assert(!info.hit); // invalid input
```

```

    return exit_success;
}
```

```
/*=============================================================================
    Copyright (c) 2001-2003 Daniel Nuffer
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
#include <boost/spirit/core.hpp>
#include <boost/spirit/tree/ast.hpp>

#include <iostream>
#include <stack>
#include <functional>
#include <string>

// This example shows how to use an AST and tree_iter_node instead of
// tree_val_node
////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

typedef char const*         iterator_t;
typedef tree_match<iterator_t, node_iter_data_factory<> >
    parse_tree_match_t;

typedef parse_tree_match_t::tree_iterator iter_t;

typedef ast_match_policy<iterator_t, node_iter_data_factory<> > match_policy_t;
typedef scanner<iterator_t, scanner_policies<iter_policy_t, match_policy_t> > sc
anner_t;
typedef rule<scanner_t> rule_t;


//  grammar rules
rule_t expression, term, factor, integer;

////////////////////////////////////////////////////////////////////////////
long evaluate(parse_tree_match_t hit);
long eval_expression(iter_t const& i);
long eval_term(iter_t const& i);
long eval_factor(iter_t const& i);
long eval_integer(iter_t const& i);

long evaluate(parse_tree_match_t hit)
{
    return eval_expression(hit.trees.begin());
}

long eval_expression(iter_t const& i)
{
    cout << "In eval_expression. i–>value = " <<
        string(i->value.begin(), i->value.end()) <<
        " i–>children.size() = " << i->children.size() << endl;

    cout << "ID: " << i->value.id().to_long() << endl;

    if (i->value.id() == integer.id())
    {
        assert(i->children.size() == 0);
        return strtol(i->value.begin(), 0, 10);
    }
```

```
    else if (i->value.id() == factor.id())
    {
        // factor can only be unary minus
        assert(*i->value.begin() == ’-’);
        return - eval_expression(i->children.begin());
    }
    else if (i->value.id() == term.id())
    {
        if (*i->value.begin() == ’*’)
        {
            assert(i->children.size() == 2);
            return eval_expression(i->children.begin()) *
                eval_expression(i->children.begin()+1);
        }
        else if (*i->value.begin() == ’/’)
        {
            assert(i->children.size() == 2);
            return eval_expression(i->children.begin()) /
                eval_expression(i->children.begin()+1);
        }
        else
            assert(0);
    }
    else if (i->value.id() == expression.id())
    {
        if (*i->value.begin() == ’+’)
        {
            assert(i->children.size() == 2);
            return eval_expression(i->children.begin()) +
                eval_expression(i->children.begin()+1);
        }
        else if (*i->value.begin() == ’-’)
        {
            assert(i->children.size() == 2);
            return eval_expression(i->children.begin()) -
                eval_expression(i->children.begin()+1);
        }
        else
            assert(0);
    }
    else
        assert(0); // error

   return 0;
}

////////////////////////////////////////////////////////////////////////////
int
main()
{
    BOOST_SPIRIT_DEBUG_RULE(integer);
    BOOST_SPIRIT_DEBUG_RULE(factor);
    BOOST_SPIRIT_DEBUG_RULE(term);
    BOOST_SPIRIT_DEBUG_RULE(expression);
    //  Start grammar definition
    integer     =   leaf_node_d[ lexeme_d[ (!ch_p(’-’) >> +digit_p) ] ];
    factor      =   integer
                |   inner_node_d[ch_p(’(’) >> expression >> ch_p(’)’)]
                |   (root_node_d[ch_p(’-’)] >> factor);
    term        =   factor >>
                    *(  (root_node_d[ch_p(’*’)] >> factor)
                      | (root_node_d[ch_p(’/’)] >> factor)
                    );
    expression  =   term >>
                    *(  (root_node_d[ch_p('+')] >> term)
                      | (root_node_d[ch_p('-')] >> term)
                    );
    //  End grammar definition


    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tThe simplest working calculator...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        const char* str_begin = str.c_str();
        const char* str_end = str.c_str();
        while (*str_end)
            ++str_end;

        scanner_t scan(str_begin, str_end);

        parse_tree_match_t hit = expression.parse(scan);


        if (hit && str_begin == str_end)
        {
#if defined(BOOST_SPIRIT_DUMP_PARSETREE_AS_XML)
            // dump parse tree as XML
            std::map<rule_id, std::string> rule_names;
            rule_names[&integer] = "integer";
            rule_names[&factor] = "factor";
            rule_names[&term] = "term";
            rule_names[&expression] = "expression";
            tree_to_xml(cout, hit.trees, str.c_str(), rule_names);
#endif

            // print the result
            cout << "parsing succeeded\n";
            cout << "result = " << evaluate(hit) << "\n\n";
        }
        else
        {
            cout << "parsing failed\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2001-2003 Dan Nuffer
    Copyright (c) 2001-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
////////////////////////////////////////////////////////////////////////////////
//
//  Full calculator example with variables
//  [ JDG 9/18/2002 ]
//
////////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/symbols/symbols.hpp>
#include <iostream>
#include <stack>
#include <functional>
#include <string>

using namespace std;
using namespace boost::spirit;

////////////////////////////////////////////////////////////////////////////////
//
//  Semantic actions
//
////////////////////////////////////////////////////////////////////////////////
struct push_num
{
    push_num(stack<double>& eval_)
    : eval(eval_) {}

    void operator()(double n) const
    {
        eval.push(n);
        cout << "push\t" << n << endl;
    }

    stack<double>& eval;
};

template <typename op>
struct do_op
{
    do_op(op const& the_op, stack<double>& eval_)
    : m_op(the_op), eval(eval_) {}

    void operator()(char const*, char const*) const
    {
        double rhs = eval.top();
        eval.pop();
        double lhs = eval.top();
        eval.pop();

        cout << "popped " << lhs << " and " << rhs << " from the stack. ";
        cout << "pushing " << m_op(lhs, rhs) << " onto the stack.\n";
        eval.push(m_op(lhs, rhs));
    }
```

```
    op m_op;
    stack<double>& eval;
};

template <class op>
do_op<op>
make_op(op const& the_op, stack<double>& eval)
{
    return do_op<op>(the_op, eval);
}

struct do_negate
{
    do_negate(stack<double>& eval_)
    : eval(eval_) {}

    void operator()(char const*, char const*) const
    {
        double lhs = eval.top();
        eval.pop();

        cout << "popped " << lhs << " from the stack. ";
        cout << "pushing " << -lhs << " onto the stack.\n";
        eval.push(-lhs);
    }

    stack<double>& eval;
};

struct get_var
{
    get_var(stack<double>& eval_)
    : eval(eval_) {}

    void operator()(double n) const
    {
        eval.push(n);
        cout << "push\t" << n << endl;
    }

    stack<double>& eval;
};

struct set_var
{
    set_var(double*& var_)
    : var(var_) {}

    void operator()(double& n) const
    {
        var = &n;
    }

    double*& var;
};

struct redecl_var
{
    void operator()(double& /*n*/) const
    {
        cout << "Warning. You are attempting to re−declare a var.\n";
    }
```

```
};

struct do_assign
{
    do_assign(double*& var_, stack<double>& eval_)
    : var(var_), eval(eval_) {}

    void operator()(char const*, char const*) const
    {
        if (var != 0)
        {
            *var = eval.top();
            cout << "assigning\n";
        }
    }

    double*& var;
    stack<double>&  eval;
};

////////////////////////////////////////////////////////////////////////////////
//
//  Our calculator grammar
//
////////////////////////////////////////////////////////////////////////////////
struct calculator : public grammar<calculator>
{
    calculator(stack<double>& eval_)
    : eval(eval_) {}

    template <typename ScannerT>
    struct definition
    {
        definition(calculator const& self)
        {
            factor =
                    real_p[push_num(self.eval)]
                |   vars[get_var(self.eval)]
                |   '(' >> expression >> ')'
                |   ('-' >> factor)[do_negate(self.eval)]
                ;

            term =
                factor
                >> *(   ('*' >> factor)[make_op(multiplies<double>(), self.eval)
]
                    |   ('/' >> factor)[make_op(divides<double>(), self.eval)]
                    )
                    ;

            expression =
                term
                >> *(  ('+' >> term)[make_op(plus<double>(), self.eval)]
                    |   ('-' >> term)[make_op(minus<double>(), self.eval)]
                    )
                    ;

            assignment =
                vars[set_var(self.var)]
                >> '=' >> expression[do_assign(self.var, self.eval)]
                ;
```

```
            var_decl =
                lexeme_d
                [
                    ((alpha_p >> *(alnum_p | '_'))
                        - vars[redecl_var()])[vars.add]
                ]
                ;

            declaration =
                "var" >> var_decl >> *(',' >> var_decl)
                ;

            statement =
                declaration | assignment | '?' >> expression
                ;
        }

        symbols<double>     vars;
        rule<ScannerT>      statement, declaration, var_decl,
                            assignment, expression, term, factor;

        rule<ScannerT> const&
        start() const { return statement; }
    };

    mutable double* var;
    stack<double>&  eval;
};

////////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tThe calculator with variables...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type a statement...or [q or Q] to quit\n\n";
    cout << "Variables may be declared:\t\tExample: var i, j, k\n";
    cout << "Assigning to a variable:\t\tExample: i = 10 * j\n";
    cout << "To evaluate an expression:\t\tExample: ? i * 3.33E-3\n\n";

    stack<double>   eval;
    calculator      calc(eval); //  Our parser

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        parse_info<> info = parse(str.c_str(), calc, space_p);

        if (info.full)
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "stopped at: \": " << info.stop << "\"\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```cpp
/*=============================================================================
    Copyright (c) 2002-2003 Hartmut Kaiser
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  Full calculator example
//  [ demonstrating phoenix and subrules ]
//
//  [ Hartmut Kaiser 10/8/2002 ]
//
///////////////////////////////////////////////////////////////////////////////

//#define BOOST_SPIRIT_DEBUG        // define this for debug output

#include <boost/spirit/core.hpp>
#include <boost/spirit/attribute.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

///////////////////////////////////////////////////////////////////////////////
//
//  Our calculator grammar using phoenix to do the semantics and subrule's
//  as it's working horses
//
//  Note:   The top rule propagates the expression result (value) upwards
//          to the calculator grammar self.val closure member which is
//          then visible outside the grammar (i.e. since self.val is the
//          member1 of the closure, it becomes the attribute passed by
//          the calculator to an attached semantic action. See the
//          driver code that uses the calculator below).
//
///////////////////////////////////////////////////////////////////////////////
struct calc_closure : boost::spirit::closure<calc_closure, double>
{
    member1 val;
};

struct calculator : public grammar<calculator, calc_closure::context_t>
{
    template <typename ScannerT>
    struct definition
    {
        definition(calculator const& self)
        {
            top = (
                expression =
                    term[self.val = arg1]
                    >> *(   ('+' >> term[self.val += arg1])
                        |   ('-' >> term[self.val -= arg1])
                        )
                ,
```

```cpp
                term =
                    factor[term.val = arg1]
                    >> *(   ('*' >> factor[term.val *= arg1])
                        |   ('/' >> factor[term.val /= arg1])
                        )
                ,

                factor
                    =   ureal_p[factor.val = arg1]
                    |   '(' >> expression[factor.val = arg1] >> ')'
                    |   ('-' >> factor[factor.val = -arg1])
                    |   ('+' >> factor[factor.val = arg1])
            );

            BOOST_SPIRIT_DEBUG_NODE(top);
            BOOST_SPIRIT_DEBUG_NODE(expression);
            BOOST_SPIRIT_DEBUG_NODE(term);
            BOOST_SPIRIT_DEBUG_NODE(factor);
        }

        subrule<0, calc_closure::context_t>  expression;
        subrule<1, calc_closure::context_t>  term;
        subrule<2, calc_closure::context_t>  factor;

        rule<ScannerT> top;

        rule<ScannerT> const&
        start() const { return top; }
    };
};

///////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tExpression parser using Phoenix...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";

    calculator calc;    //  Our parser

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        double n = 0;
        parse_info<> info = parse(str.c_str(), calc[var(n) = arg1], space_p);

        //  calc[var(n) = arg1] invokes the calculator and extracts
        //  the result of the computation. See calculator grammar
        //  note above.

        if (info.full)
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << "result = " << n << endl;
            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "stopped at: \": " << info.stop << "\"\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```cpp
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  A primitive calculator that knows how to add and subtract.
//  [ demonstrating phoenix ]
//
//  [ JDG 6/28/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/phoenix/primitives.hpp>
#include <boost/spirit/phoenix/operators.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

///////////////////////////////////////////////////////////////////////////////
//
//  Our primitive calculator
//
///////////////////////////////////////////////////////////////////////////////
template <typename IteratorT>
bool primitive_calc(IteratorT first, IteratorT last, double& n)
{
    return parse(first, last,

        //  Begin grammar
        (
            real_p[var(n) = arg1]
            >> *(   ('+' >> real_p[var(n) += arg1])
                |   ('-' >> real_p[var(n) -= arg1])
                )
        )
        ,
        //  End grammar

        space_p).full;
}

////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tA primitive calculator...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
```

```cpp
    cout << "Give me a list of numbers to be added or subtracted.\n";
    cout << "Example: 1 + 10 + 3 - 4 + 9\n";
    cout << "The result is computed using Phoenix.\n";
    cout << "Type [q or Q] to quit\n\n";

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        double n;
        if (primitive_calc(str.begin(), str.end(), n))
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << str << " Parses OK: " << endl;

            cout << "result = " << n;
            cout << "\n-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```cpp
/*=============================================================================
    Copyright (c) 2001-2003 Hartmut Kaiser
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  This sample shows, how to use Phoenix for implementing a
//  simple (RPN style) calculator [ demonstrating phoenix ]
//
//  [ HKaiser 2001 ]
//  [ JDG 6/29/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/attribute.hpp>
#include <boost/spirit/phoenix/functions.hpp>
#include <iostream>
#include <string>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

///////////////////////////////////////////////////////////////////////////////
//
//  Our RPN calculator grammar using phoenix to do the semantics
//  The class 'RPNCalculator' implements a polish reverse notation
//  calculator which is equivalent to the following YACC description.
//
//  exp:
//        NUM           { $$ = $1;           }
//      | exp exp '+'   { $$ = $1 + $2;      }
//      | exp exp '-'   { $$ = $1 - $2;      }
//      | exp exp '*'   { $$ = $1 * $2;      }
//      | exp exp '/'   { $$ = $1 / $2;      }
//      | exp exp '^'   { $$ = pow ($1, $2); }  /* Exponentiation */
//      | exp 'n'       { $$ = -$1;          }  /* Unary minus */
//      ;
//
//  The different notation results from the requirement of LL parsers not to
//  allow left recursion in their grammar (would lead to endless recursion).
//  Therefore the left recursion in the YACC script before is transformated
//  into iteration. To some, this is less intuitive, but once you get used
//  to it, it's very easy to follow.
//
//  Note:   The top rule propagates the expression result (value) upwards
//          to the calculator grammar self.val closure member which is
//          then visible outside the grammar (i.e. since self.val is the
//          member1 of the closure, it becomes the attribute passed by
//          the calculator to an attached semantic action. See the
//          driver code that uses the calculator below).
//
///////////////////////////////////////////////////////////////////////////////
struct pow_
{
    template <typename X, typename Y>
```

```cpp
    struct result { typedef X type; };

    template <typename X, typename Y>
    X operator()(X x, Y y) const
    {
        using namespace std;
        return pow(x, y);
    }
};

//  Notice how power(x, y) is lazily implemented using Phoenix function.
function<pow_> power;

struct calc_closure : boost::spirit::closure<calc_closure, double, double>
{
    member1 x;
    member2 y;
};

struct calculator : public grammar<calculator, calc_closure::context_t>
{
    template <typename ScannerT>
    struct definition {

        definition(calculator const& self)
        {
            top = expr                      [self.x = arg1];
            expr =
                real_p                      [expr.x = arg1]
                >> *(
                        expr                [expr.y = arg1]
                        >>  (
                                ch_p('+')   [expr.x += expr.y]
                            |   ch_p('-')   [expr.x -= expr.y]
                            |   ch_p('*')   [expr.x *= expr.y]
                            |   ch_p('/')   [expr.x /= expr.y]
                            |   ch_p('^')   [expr.x = power(expr.x, expr.y)]
                            )
                    |   ch_p('n')           [expr.x = -expr.x]
                    )
                ;
        }

        typedef rule<ScannerT, calc_closure::context_t> rule_t;
        rule_t expr;
        rule<ScannerT> top;

        rule<ScannerT> const&
        start() const { return top; }
    };
};

///////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\t\tExpression parser using Phoenix...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";

    calculator calc;    //  Our parser

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        double n = 0;
        parse_info<> info = parse(str.c_str(), calc[var(n) = arg1], space_p);

        //  calc[var(n) = arg1] invokes the calculator and extracts
        //  the result of the computation. See calculator grammar
        //  note above.

        if (info.full)
        {
            cout << "-------------------------\n";
            cout << "Parsing succeeded\n";
            cout << "result = " << n << endl;
            cout << "-------------------------\n";
        }
        else
        {
            cout << "-------------------------\n";
            cout << "Parsing failed\n";
            cout << "stopped at: \": " << info.stop << "\"\n";
            cout << "-------------------------\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```cpp
/*=============================================================================
    Copyright (c) 1998-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
////////////////////////////////////////////////////////////////////////////
//
//  The calculator using a simple virtual machine and compiler.
//
//  Ported to v1.5 from the original v1.0 code by JDG
//  [ JDG 9/18/2002 ]
//
////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <iostream>
#include <vector>
#include <string>

////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
//
//  The VMachine
//
///////////////////////////////////////////////////////////////////////////////
enum ByteCodes
{
    OP_NEG,     //  negate the top stack entry
    OP_ADD,     //  add top two stack entries
    OP_SUB,     //  subtract top two stack entries
    OP_MUL,     //  multiply top two stack entries
    OP_DIV,     //  divide top two stack entries
    OP_INT,     //  push constant integer into the stack
    OP_RET      //  return from the interpreter
};

class vmachine
{
public:
                vmachine(unsigned stackSize = 1024)
                :   stack(new int[stackSize]),
                    stackPtr(stack) {}
                ~vmachine() { delete [] stack; }
    int         top() const { return stackPtr[-1]; };
    void        execute(int code[]);

private:

    int*        stack;
    int*        stackPtr;
};

void
vmachine::execute(int code[])
{
    int const*  pc = code;
    bool        running = true;
```

```cpp
    stackPtr = stack;

    while (running)
    {
        switch (*pc++)
        {
            case OP_NEG:
                stackPtr[-1] = -stackPtr[-1];
                break;

            case OP_ADD:
                stackPtr--;
                stackPtr[-1] += stackPtr[0];
                break;

            case OP_SUB:
                stackPtr--;
                stackPtr[-1] -= stackPtr[0];
                break;

            case OP_MUL:
                stackPtr--;
                stackPtr[-1] *= stackPtr[0];
                break;

            case OP_DIV:
                stackPtr--;
                stackPtr[-1] /= stackPtr[0];
                break;

            case OP_INT:
                // Check stack overflow here!
                *stackPtr++ = *pc++;
                break;

            case OP_RET:
                running = false;
                break;
        }
    }
}

///////////////////////////////////////////////////////////////////////////////
//
//  The Compiler
//
///////////////////////////////////////////////////////////////////////////////
struct push_int
{
    push_int(vector<int>& code_)
    : code(code_) {}

    void operator()(char const* str, char const* /*end*/) const
    {
        using namespace std;
        int n = strtol(str, 0, 10);
        code.push_back(OP_INT);
        code.push_back(n);
        cout << "push\t" << int(n) << endl;
    }

    vector<int>& code;
```

```
};

struct push_op
{
    push_op(int op_, vector<int>& code_)
    : op(op_), code(code_) {}

    void operator()(char const*, char const*) const
    {
        code.push_back(op);

        switch (op) {

            case OP_NEG:
                cout << "neg\n";
                break;

            case OP_ADD:
                cout << "add\n";
                break;

            case OP_SUB:
                cout << "sub\n";
                break;

            case OP_MUL:
                cout << "mul\n";
                break;

            case OP_DIV:
                cout << "div\n";
                break;
        }
    }

    int op;
    vector<int>& code;
};

template <typename GrammarT>
static bool
compile(GrammarT const& calc, char const* expr)
{
    cout << "\n////////////////////////////////////////////////////////////////\n\n";

    parse_info<char const*>
        result = parse(expr, calc, space_p);

    if (result.full)
    {
        cout << "\t\t" << expr << " Parses OK\n\n\n";
        calc.code.push_back(OP_RET);
        return true;
    }
    else
    {
        cout << "\t\t" << expr << " Fails parsing\n";
        cout << "\t\t";
        for (int i = 0; i < (result.stop - expr); i++)
            cout << " ";
        cout << "^--Here\n\n\n";
        return false;
```

```
    }
}

////////////////////////////////////////////////////////////////////////////
//
//  Our calculator grammar
//
////////////////////////////////////////////////////////////////////////////
struct calculator : public grammar<calculator>
{
    calculator(vector<int>& code_)
    : code(code_) {}

    template <typename ScannerT>
    struct definition
    {
        definition(calculator const& self)
        {
            integer =
                lexeme_d[ (+digit_p)[push_int(self.code)] ]
                ;

            factor =
                    integer
                |   '(' >> expression >> ')'
                |   ('-' >> factor)[push_op(OP_NEG, self.code)]
                |   ('+' >> factor)
                ;

            term =
                factor
                >> *(   ('*' >> factor)[push_op(OP_MUL, self.code)]
                    |   ('/' >> factor)[push_op(OP_DIV, self.code)]
                    )
                    ;

            expression =
                term
                >> *(   ('+' >> term)[push_op(OP_ADD, self.code)]
                    |   ('-' >> term)[push_op(OP_SUB, self.code)]
                    )
                    ;
        }

        rule<ScannerT> expression, term, factor, integer;

        rule<ScannerT> const&
        start() const { return expression; }
    };

    vector<int>& code;
};

////////////////////////////////////////////////////////////////////////////
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////
int
main()
{
    cout << "////////////////////////////////////////////////////////////////\n\n";
    cout << "\t\tA simple virtual machine...\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "Type an expression...or [q or Q] to quit\n\n";

    vmachine    mach;       //  Our virtual machine
    vector<int> code;       //  Our VM code
    calculator  calc(code); //  Our parser

    string str;
    while (getline(cin, str))
    {
        if (str.empty() || str[0] == 'q' || str[0] == 'Q')
            break;

        code.clear();
        if (compile(calc, str.c_str()))
        {
            mach.execute(&*code.begin());
            cout << "\n\nresult = " << mach.top() << "\n\n";
        }
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002 Juan Carlos Arevalo-Baeza
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
//
//  A parser for a comma separated list of numbers,
//  with positional error reporting
//  See the "Position Iterator" chapter in the User's Guide.
//
//  [ JCAB 9/28/2002 ]
//
///////////////////////////////////////////////////////////////////////////////
#include <boost/spirit/core.hpp>
#include <boost/spirit/iterator/position_iterator.hpp>
#include <boost/spirit/utility/functor_parser.hpp>
#include <iostream>
#include <fstream>
#include <vector>

///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;

///////////////////////////////////////////////////////////////////////////////
//
//  Our error reporting parsers
//
///////////////////////////////////////////////////////////////////////////////
std::ostream& operator<<(std::ostream& out, file_position const& lc)
{
    return out <<
            "\nFile:\t" << lc.file <<
            "\nLine:\t" << lc.line <<
            "\nCol:\t" << lc.column << endl;
}

struct error_report_parser {
    char const* eol_msg;
    char const* msg;

    error_report_parser(char const* eol_msg_, char const* msg_):
        eol_msg(eol_msg_),
        msg    (msg_)
    {}

    typedef nil_t result_t;

    template <typename ScannerT>
    int
    operator()(ScannerT const& scan, result_t& /*result*/) const
    {
        if (scan.at_end()) {
            if (eol_msg) {
                file_position fpos = scan.first.get_position();
                cerr << fpos << eol_msg << endl;
            }
        } else {
```

```
            if (msg) {
                file_position fpos = scan.first.get_position();
                cerr << fpos << msg << endl;
            }
        }

        return -1; // Fail.
    }

};
typedef functor_parser<error_report_parser> error_report_p;

error_report_p
error_badnumber_or_eol =
    error_report_parser(
        "Expecting a number, but found the end of the file\n",
        "Expecting a number, but found something else\n"
    );

error_report_p
error_badnumber =
    error_report_parser(
        0,
        "Expecting a number, but found something else\n"
    );

error_report_p
error_comma =
    error_report_parser(
        0,
        "Expecting a comma, but found something else\n"
    );

///////////////////////////////////////////////////////////////////////////////
//
//  Our comma separated list parser
//
///////////////////////////////////////////////////////////////////////////////
bool
parse_numbers(char const* filename, char const* str, vector<double>& v)
{
    typedef position_iterator<char const*> iterator_t;
    iterator_t begin(str, str + strlen(str), filename);
    iterator_t end;
    begin.set_tabchars(8);
    return parse(begin, end,

        //  Begin grammar
        (
            (real_p[push_back_a(v)] | error_badnumber)
         >> *(
                (',' | error_comma)
             >> (real_p[push_back_a(v)] | error_badnumber_or_eol)
            )
        )
        ,
        //  End grammar

        space_p).full;
}

//////////////////////////////////////////////////////////////////////////////
```

```
//
//  Main program
//
////////////////////////////////////////////////////////////////////////////
int
main(int argc, char **argv)
{
    cout << "/////////////////////////////////////////////////////////\n\n";
    cout << "\tAn error-reporting parser for Spirit...\n\n";
    cout << "Parses a comma separated list of numbers from a file.\n";
    cout << "The numbers will be inserted in a vector of numbers\n\n";
    cout << "/////////////////////////////////////////////////////////\n\n";

    char str[65536];
    char const* filename;

    if (argc > 1) {
        filename = argv[1];
        ifstream file(filename);
        file.get(str, sizeof(str), '\0');
    } else {
        filename = "<cin>";
        cin.get(str, sizeof(str), '\0');
    }

    vector<double> v;
    if (parse_numbers(filename, str, v))
    {
        cout << "-------------------------\n";
        cout << "Parsing succeeded\n";
        cout << str << " Parses OK: " << endl;

        for (vector<double>::size_type i = 0; i < v.size(); ++i)
            cout << i << ": " << v[i] << endl;

        cout << "-------------------------\n";
    }
    else
    {
        cout << "-------------------------\n";
        cout << "Parsing failed\n";
        cout << "-------------------------\n";
    }

    cout << "Bye... :-) \n\n";
    return 0;
}
```

```
/*==============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
==============================================================================*/
#include <boost/spirit/core.hpp>
#include <boost/spirit/actor/push_back_actor.hpp>
#include <boost/spirit/dynamic/if.hpp>
#include <boost/spirit/dynamic/for.hpp>
#include <boost/spirit/phoenix.hpp>
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

////////////////////////////////////////////////////////////////////////////////
//
//  Sample parser for binary data. This sample highlights the use of dynamic
//  parsing where the result of actions direct the actual parsing behavior.
//  We shall demonstrate 1) the use of phoenix to implement lambda (unnamed)
//  functions, 2) dynamic looping using for_p, 3) the push_back_a actor for
//  stuffing data into a vector, and 4) the if_p parser for choosing parser
//  branches based on semantic conditions.
//
//  << Sample idea by Florian Weimer >>
//
//  For simplicity, we shall use bytes as atoms (and not 16-bit quantities
//  in big-endian format or something similar, which would be more realistic)
//  and PASCAL strings.
//
//  A packet is the literal octet with value 255, followed by a variable
//  octet N (denoting the total length of the packet), followed by N-2 octets
//  (the payload). The payload contains a variable-length header, followed
//  by zero or more elements.
//
//  The header contains a single PASCAL string.
//
//  An element is a PASCAL string (alternative: an element is an octet M,
//  followed by [M/8] bytes, i.e. the necessary number of bytes to store M
//  bits).
//
//  (This data structure is inspired by the format of a BGP UPDATE message.)
//
//  Packet layout:
//
//       .--------------------.
//       |       0xff         |  ^
//       +--------------------+  |
//       |   packet length    |  |
//       +--------------------+  | number of bytes indicated by packet length
//       :                    :  |
//       :      payload       :  |
//       |                    |  v
//       `--------------------'
//
//  Payload layout:
//
//       .--------------------.
//       |   header length    |
```

```
//       +--------------------+
//       |   header octets    |  ^
//       :                    :  |  number of octets given by header length
//       :                    :  |
//       :                    :  v
//       +--------------------+
//       |   IPv4 prefix      |  ^
//       :                    :  |  IPv4 prefixes have variable length (see
//       +--------------------+  |  below).  The number of prefixes is
//       |   IPv4 prefix      |  |  determined by the packet length.
//       :                    :  |
//       +--------------------+  |
//       :                    :  |
//       :                    :  v
//
//
//  IPv4 prefix layout comes in five variants, depending on the first
//  octet:
//
//       .--------------------.
//       |       0x00         |     single octet, corresponds to 0.0.0.0/0
//       `--------------------'
//
//       .--------------------.
//       |    0x01 to 0x08    |     two octets, prefix lengths up to /8.
//       +--------------------+
//       |   MSB of network   |
//       `--------------------'
//
//       .--------------------.
//       |    0x09 to 0x10    |     three octets, prefix lengths up to /16.
//       +--------------------+
//       |   MSB of network   |
//       +--------------------+
//       |   next octet       |
//       `--------------------'
//
//       .--------------------.
//       |    0x11 to 0x18    |     four octets, prefix lengths up to /24.
//       +--------------------+
//       |   MSB of network   |
//       +--------------------+
//       |   next octet       |
//       +--------------------+
//       |   next octet       |
//       `--------------------'
//
//       .--------------------.
//       |    0x19 to 0x20    |     five octets, prefix lengths up to /32.
//       +--------------------+
//       |   MSB of network   |
//       +--------------------+
//       |   next octet       |
//       +--------------------+
//       |   next octet       |
//       +--------------------+
//       |   LSB of network   |
//       `--------------------'
//
////////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

struct ipv4_prefix_data
{
    char prefix_len, n0, n1, n2, n3;

    ipv4_prefix_data()
        : prefix_len(0),n0(0),n1(0),n2(0),n3(0) {}
};

struct ipv4_data
{
    char packet_len, header_len;
    std::string header;
    std::vector<ipv4_prefix_data> prefixes;

    ipv4_data()
        : packet_len(0),header_len(0){}

};

struct ipv4 : public grammar<ipv4>
{
    template <typename ScannerT>
    struct definition
    {
        definition(ipv4 const& self)
        {
            packet =
                '\xff'
                >> anychar_p[var(self.data.packet_len) = arg1]
                >> payload
            ;

            payload =
                anychar_p[var(self.data.header_len) = arg1]
                >>  for_p(var(i) = 0, var(i) < var(self.data.header_len), ++var(
i))
                    [
                        anychar_p[var(self.data.header) += arg1]
                    ]
                >> *ipv4_prefix
             ;

            ipv4_prefix =
                anychar_p
                [
                    var(temp.prefix_len) = arg1,
                    var(temp.n0) = 0,
                    var(temp.n1) = 0,
                    var(temp.n2) = 0,
                    var(temp.n3) = 0
                ]

                >>  if_p(var(temp.prefix_len) > 0x00)
                    [
                        anychar_p[var(temp.n0) = arg1]
                        >>  if_p(var(temp.prefix_len) > 0x08)
                            [
                                anychar_p[var(temp.n1) = arg1]
                                >>  if_p(var(temp.prefix_len) > 0x10)
                                    [
```

```
                                        anychar_p[var(temp.n2) = arg1]
                                        >>  if_p(var(temp.prefix_len) > 0x18)
                                            [
                                                anychar_p[var(temp.n3) = arg1]
                                            ]
                                    ]
                            ]
                    ]
                    [
                        push_back_a(self.data.prefixes, temp)
                    ]
            ;
        }

        int i;
        ipv4_prefix_data temp;
        rule<ScannerT> packet, payload, ipv4_prefix;
        rule<ScannerT> const&
        start() const { return packet; }
    };

    ipv4(ipv4_data& data)
        : data(data) {}

    ipv4_data& data;
};

///////////////////////////////////////////////////////////////////////////
//
//  Main program
//
///////////////////////////////////////////////////////////////////////////
int
as_byte(char n)
{
    if (n < 0)
        return n + 256;
    return n;
}

void
print_prefix(ipv4_prefix_data const& prefix)
{
    cout << "prefix length = " << as_byte(prefix.prefix_len) << endl;
    cout << "n0 = " << as_byte(prefix.n0) << endl;
    cout << "n1 = " << as_byte(prefix.n1) << endl;
    cout << "n2 = " << as_byte(prefix.n2) << endl;
    cout << "n3 = " << as_byte(prefix.n3) << endl;
}

void
parse_ipv4(char const* str, unsigned len)
{
    ipv4_data data;
    ipv4 g(data);
    parse_info<> info = parse(str, str+len, g);

    if (info.full)
    {
        cout << "-------------------------\n";
        cout << "Parsing succeeded\n";
        cout << "packet length = " << as_byte(data.packet_len) << endl;
        cout << "header length = " << as_byte(data.header_len) << endl;
        cout << "header = " << data.header << endl;

        for_each(data.prefixes.begin(), data.prefixes.end(), print_prefix);
        cout << "------------------------\n";
    }
    else
    {
        cout << "Parsing failed\n";
        cout << "stopped at:";
        for (char const* s = info.stop; s != str+len; ++s)
            cout << static_cast<int>(*s) << endl;
    }
}

// Test inputs:

// The string in the header is "empty", the prefix list is empty.
char const i1[8] =
{
    0xff,0x08,0x05,
    'e','m','p','t','y'
};

// The string in the header is "default route", the prefix list
// has just one element, 0.0.0.0/0.
char const i2[17] =
{
    0xff,0x11,0x0d,
    'd','e','f','a','u','l','t',' ',
    'r','o','u','t','e',
    0x00
};

// The string in the header is "private address space", the prefix list
// has the elements 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16.
char const i3[32] =
{
    0xff,0x20,0x15,
    'p','r','i','v','a','t','e',' ',
    'a','d','d','r','e','s','s',' ',
    's','p','a','c','e',
    0x08,0x0a,
    0x0c,0xac,0x10,
    0x10,0xc0,0xa8
};

int
main()
{
    parse_ipv4(i1, sizeof(i1));
    parse_ipv4(i2, sizeof(i2));
    parse_ipv4(i3, sizeof(i3));
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
#include <boost/spirit/core.hpp>
#include <boost/spirit/actor/push_back_actor.hpp>
#include <boost/spirit/dynamic/if.hpp>
#include <boost/spirit/dynamic/for.hpp>
#include <boost/spirit/phoenix.hpp>
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

///////////////////////////////////////////////////////////////////////////////
//
//  Please check it out ipv4.cpp sample first!
//  << See ipv4.cpp sample for details >>
//
//  This is a variation of the ipv4.cpp sample. The original ipv4.cpp code
//  compiles to 36k on MSVC7.1. Not bad! Yet, we want to shave a little bit
//  more. Is it possible? Yes! This time, we'll use subrules and just store
//  the rules in a plain old struct. We are parsing at the char level anyway,
//  so we know what type of rule we'll need: a plain rule<>. The result: we
//  shaved off another 20k. Now the code compiles to 16k on MSVC7.1.
//
//  Could we have done better? Yes, but only if only we had typeof! << See
//  the techniques section of the User's guide >> ... Someday... :-)
//
///////////////////////////////////////////////////////////////////////////////
using namespace std;
using namespace boost::spirit;
using namespace phoenix;

struct ipv4_prefix_data
{
    char prefix_len, n0, n1, n2, n3;

    ipv4_prefix_data()
        : prefix_len(0),n0(0),n1(0),n2(0),n3(0) {}
};

struct ipv4_data
{
    char packet_len, header_len;
    std::string header;
    std::vector<ipv4_prefix_data> prefixes;

    ipv4_data()
        : packet_len(0),header_len(0){}

};

struct ipv4
{
    ipv4(ipv4_data& data)
        : data(data)
    {
```

```
        start =
        (
            packet =
                '\xff'
                >> anychar_p[var(data.packet_len) = arg1]
                >> payload
            ,

            payload =
                anychar_p[var(data.header_len) = arg1]
                >>  for_p(var(i) = 0, var(i) < var(data.header_len), ++var(i))
                    [
                        anychar_p[var(data.header) += arg1]
                    ]
                >> *ipv4_prefix
             ,

            ipv4_prefix =
                anychar_p
                [
                    var(temp.prefix_len) = arg1,
                    var(temp.n0) = 0,
                    var(temp.n1) = 0,
                    var(temp.n2) = 0,
                    var(temp.n3) = 0
                ]

                >>  if_p(var(temp.prefix_len) > 0x00)
                    [
                        anychar_p[var(temp.n0) = arg1]
                        >>  if_p(var(temp.prefix_len) > 0x08)
                            [
                                anychar_p[var(temp.n1) = arg1]
                                >>  if_p(var(temp.prefix_len) > 0x10)
                                    [
                                        anychar_p[var(temp.n2) = arg1]
                                        >>  if_p(var(temp.prefix_len) > 0x18)
                                            [
                                                anychar_p[var(temp.n3) = arg1]
                                            ]
                                    ]
                            ]
                    ]
                    [
                        push_back_a(data.prefixes, temp)
                    ]
        );
    }

    int i;
    ipv4_prefix_data temp;

    rule<> start;
    subrule<0> packet;
    subrule<1> payload;
    subrule<2> ipv4_prefix;
    ipv4_data& data;
};

////////////////////////////////////////////////////////////////////////////
//
//  Main program
```

```
//
///////////////////////////////////////////////////////////////////////////////
int
as_byte(char n)
{
    if (n < 0)
        return n + 256;
    return n;
}

void
print_prefix(ipv4_prefix_data const& prefix)
{
    cout << "prefix length = " << as_byte(prefix.prefix_len) << endl;
    cout << "n0 = " << as_byte(prefix.n0) << endl;
    cout << "n1 = " << as_byte(prefix.n1) << endl;
    cout << "n2 = " << as_byte(prefix.n2) << endl;
    cout << "n3 = " << as_byte(prefix.n3) << endl;
}

void
parse_ipv4(char const* str, unsigned len)
{
    ipv4_data data;
    ipv4 g(data);
    parse_info<> info = parse(str, str+len, g.start);

    if (info.full)
    {
        cout << "-------------------------\n";
        cout << "Parsing succeeded\n";

        cout << "packet length = " << as_byte(data.packet_len) << endl;
        cout << "header length = " << as_byte(data.header_len) << endl;
        cout << "header = " << data.header << endl;

        for_each(data.prefixes.begin(), data.prefixes.end(), print_prefix);
        cout << "-------------------------\n";
    }
    else
    {
        cout << "Parsing failed\n";
        cout << "stopped at:";
        for (char const* s = info.stop; s != str+len; ++s)
            cout << static_cast<int>(*s) << endl;
    }
}

// Test inputs:

// The string in the header is "empty", the prefix list is empty.
char const i1[8] =
{
    0xff,0x08,0x05,
    ’e’,’m’,’p’,’t’,’y’
};

// The string in the header is "default route", the prefix list
// has just one element, 0.0.0.0/0.
char const i2[17] =
{
    0xff,0x11,0x0d,
```

```
    ’d’,’e’,’f’,’a’,’u’,’l’,’t’,’ ’,
    ’r’,’o’,’u’,’t’,’e’,
    0x00
};

// The string in the header is "private address space", the prefix list
// has the elements 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16.
char const i3[32] =
{
    0xff,0x20,0x15,
    ’p’,’r’,’i’,’v’,’a’,’t’,’e’,’ ’,
    ’a’,’d’,’d’,’r’,’e’,’s’,’s’,’ ’,
    ’s’,’p’,’a’,’c’,’e’,
    0x08,0x0a,
    0x0c,0xac,0x10,
    0x10,0xc0,0xa8
};

int
main()
{
    parse_ipv4(i1, sizeof(i1));
    parse_ipv4(i2, sizeof(i2));
    parse_ipv4(i3, sizeof(i3));
    return 0;
}
```

```
/*==============================================================================
    Copyright (c) 2003 Vaclav Vesely
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
==============================================================================*/
//
//  This example demonstrates the lazy_p parser. You should read
//  "The Lazy Parser" in the documentation.
//
//  We want to parse nested blocks of numbers like this:
//
//  dec {
//      1 2 3
//      bin {
//          1 10 11
//      }
//      4 5 6
//  }
//
//  where the numbers in the "dec" block are wrote in the decimal system and
//  the numbers in the "bin" block are wrote in the binary system. We want
//  parser to return the overall sum.
//
//  To achieve this when base ("bin" or "dec") is parsed, in semantic action
//  we store a pointer to the appropriate numeric parser in the closure
//  variable block.int_rule. Then, when we need to parse a number we use the
//  lazy_p parser to invoke the parser stored in the block.int_rule pointer.
//
//------------------------------------------------------------------------------
#include <cassert>
#include <boost/cstdlib.hpp>
#include <boost/spirit/phoenix.hpp>
#include <boost/spirit/core.hpp>
#include <boost/spirit/symbols.hpp>
#include <boost/spirit/attribute.hpp>
#include <boost/spirit/dynamic.hpp>

using namespace boost;
using namespace spirit;
using namespace phoenix;

//------------------------------------------------------------------------------
//  my grammar

struct my_grammar
    :   public grammar<my_grammar, parser_context<int> >
{
    // grammar definition
    template<typename ScannerT>
    struct definition
    {
        typedef rule<ScannerT> rule_t;
        typedef stored_rule<ScannerT, parser_context<int> > number_rule_t;

        struct block_closure;
        typedef spirit::closure<
            block_closure,
            int,
            typename number_rule_t::alias_t>
```

```
        closure_base_t;

        struct block_closure : closure_base_t
        {
            typename closure_base_t::member1 sum;
            typename closure_base_t::member2 int_rule;
        };

        // block rule type
        typedef rule<ScannerT, typename block_closure::context_t> block_rule_t;

        block_rule_t block;
        rule_t block_item;
        symbols<number_rule_t> base;

        definition(my_grammar const& self)
        {
            block =
                    base[
                        block.sum = 0,
                        // store a number rule in a closure member
                        block.int_rule = arg1
                    ]
                >>  "{"
                >>  *block_item
                >>  "}"
                ;

            block_item =
                    // use the stored rule
                    lazy_p(block.int_rule)[block.sum += arg1]
                |   block[block.sum += arg1]
                ;

            // bind base keywords and number parsers
            base.add
                ("bin", bin_p)
                ("dec", uint_p)
                ;
        }

        block_rule_t const& start() const
        {
            return block;
        }
    };
};

//------------------------------------------------------------------------------

int main()
{
    my_grammar gram;
    parse_info<> info;

    int result;
    info = parse("bin{1 dec{1 2 3} 10}", gram[var(result) = arg1], space_p);
    assert(info.full);
    assert(result == 9);

    return exit_success;
}
```

```
//----------------------------------------------------------------------------
```

```
/*=============================================================================
    Copyright (c) 2001-2003 Hartmut Kaiser
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
// This sample show the usage of parser parameters.
//
// Parser parameters are used to pass some values from the outer parsing scope
// to the next inner scope. They can be imagined as the opposite to the return
// value paradigm, which returns some value from the inner to the next outer
// scope. See the "Closures" chapter in the User's Guide.

#include <string>
#include <iostream>
#include <cassert>

#if defined(_MSC_VER) /*&& !defined(__COMO__)*/
#pragma warning(disable: 4244)
#pragma warning(disable: 4355)
#endif // defined(_MSC_VER) && !defined(__COMO__)

#include <boost/spirit/core.hpp>
#include <boost/spirit/symbols/symbols.hpp>

#include <boost/spirit/phoenix/tuples.hpp>
#include <boost/spirit/phoenix/tuple_helpers.hpp>
#include <boost/spirit/phoenix/primitives.hpp>
#include <boost/spirit/attribute/closure.hpp>

///////////////////////////////////////////////////////////////////////////////
// used namespaces
using namespace boost::spirit;
using namespace phoenix;
using namespace std;

///////////////////////////////////////////////////////////////////////////////
// Helper class for encapsulation of the type for the parsed variable names
class declaration_type
{
public:
    enum vartype {
        vartype_unknown = 0,        // unknown variable type
        vartype_int = 1,            // 'int'
        vartype_real = 2            // 'real'
    };

    declaration_type() : type(vartype_unknown)
    {
    }
    template <typename ItT>
    declaration_type(ItT const &first, ItT const &last)
    {
        init(string(first, last-first-1));
    }
    declaration_type(declaration_type const &type_) : type(type_.type)
    {
    }
    declaration_type(string const &type_) : type(vartype_unknown)
```

```
    {
        init(type_);
    }

// access to the variable type
    operator vartype const &() const { return type; }
    operator string ()
    {
        switch(type) {
        default:
        case vartype_unknown:   break;
        case vartype_int:       return string("int");
        case vartype_real:      return string("real");
        }
        return string ("unknown");
    }

    void swap(declaration_type &s) { std::swap(type, s.type); }

protected:
    void init (string const &type_)
    {
        if (type_ == "int")
            type = vartype_int;
        else if (type_ == "real")
            type = vartype_real;
        else
            type = vartype_unknown;
    }

private:
    vartype type;
};

///////////////////////////////////////////////////////////////////////////////
//
//  used closure type
//
///////////////////////////////////////////////////////////////////////////////
struct var_decl_closure : boost::spirit::closure<var_decl_closure, declaration_t
ype>
{
    member1 val;
};

///////////////////////////////////////////////////////////////////////////////
//
//  symbols_with_data
//
//      Helper class for inserting an item with data into a symbol table
//
///////////////////////////////////////////////////////////////////////////////
template <typename T, typename InitT>
class symbols_with_data
{
public:
    typedef
        symbol_inserter<T, boost::spirit::impl::tst<T, char> >
        symbol_inserter_t;

    symbols_with_data(symbol_inserter_t const &add_, InitT const &data_) :
        add(add_), data(as_actor<InitT>::convert(data_))
    {
    }

    template <typename IteratorT>
    symbol_inserter_t const &
    operator()(IteratorT const &first_, IteratorT const &last) const
    {
        IteratorT first = first_;
        return add(first, last, data());
    }

private:
    symbol_inserter_t const &add;
    typename as_actor<InitT>::type data;
};

template <typename T, typename CharT, typename InitT>
inline
symbols_with_data<T, InitT>
symbols_gen(symbol_inserter<T, boost::spirit::impl::tst<T, CharT> > const &add_,
    InitT const &data_)
{
    return symbols_with_data<T, InitT>(add_, data_);
}

///////////////////////////////////////////////////////////////////////////////
// The var_decl_list grammar parses variable declaration list

struct var_decl_list :
    public grammar<var_decl_list, var_decl_closure::context_t>
{
    template <typename ScannerT>
    struct definition
    {
        definition(var_decl_list const &self)
        {
        // pass variable type returned from 'type' to list closure member 0
            decl = type[self.val = arg1] >> +space_p >> list(self.val);

        // m0 to access arg 0 of list --> passing variable type down to ident
            list = ident(list.val) >> *(',' >> ident(list.val));

        // store identifier and type into the symbol table
            ident = (*alnum_p)[symbols_gen(symtab.add, ident.val)];

        // the type of the decl is returned in type's closure member 0
            type =
                    str_p("int")[type.val = construct_<string>(arg1, arg2)]
                |   str_p("real")[type.val = construct_<string>(arg1, arg2)]
                ;

            BOOST_SPIRIT_DEBUG_RULE(decl);
            BOOST_SPIRIT_DEBUG_RULE(list);
            BOOST_SPIRIT_DEBUG_RULE(ident);
            BOOST_SPIRIT_DEBUG_RULE(type);
        }

        rule<ScannerT> const&
        start() const { return decl; }

    private:
        typedef rule<ScannerT, var_decl_closure::context_t> rule_t;
```

```
        rule_t type;
        rule_t list;
        rule_t ident;
        symbols<declaration_type> symtab;

        rule<ScannerT> decl;        // start rule
    };
};

///////////////////////////////////////////////////////////////////////////////
// main entry point
int main()
{
var_decl_list decl;
declaration_type type;
char const *pbegin = "int var1";

    if (parse (pbegin, decl[assign(type)]).full) {
        cout << endl
             << "Parsed variable declarations successfully!" << endl
             << "Detected type: " << declaration_type::vartype(type)
             << " (" << string(type) << ")"
             << endl;
    } else {
        cout << endl
             << "Parsing the input stream failed!"
             << endl;
    }
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Martin Wille
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
// vim:ts=4:sw=4:et
//
//  Demonstrate regular expression parsers for match based text conversion
//
//  This sample requires an installed version of the boost regex library
//  (http://www.boost.org) The sample was tested with boost V1.29.0
//
//  Note: - there is no error handling in this example
//        - this program isn’t particularly useful
//
//  This example shows one way build a kind of filter program.
//  It reads input from std::cin and uses a grammar and actions
//  to print out a modified version of the input.
//
//  [ Martin Wille, 10/18/2002 ]
//
///////////////////////////////////////////////////////////////////////////////

#include <string>
#include <iostream>
#include <streambuf>
#include <sstream>
#include <deque>
#include <iterator>

#include <boost/function.hpp>
#include <boost/spirit/core.hpp>

///////////////////////////////////////////////////////////////////////////////
//
//  The following header must be included, if regular expression support is
//  required for Spirit.
//
//  The BOOST_SPIRIT_NO_REGEX_LIB PP constant should be defined, if you’re using
 the
//  Boost.Regex library from one translation unit only. Otherwise you have to
//  link with the Boost.Regex library as defined in the related documentation
//  (see. http://www.boost.org).
//
///////////////////////////////////////////////////////////////////////////////
#define BOOST_SPIRIT_NO_REGEX_LIB
#include <boost/spirit/utility/regex.hpp>

using namespace boost::spirit;
using namespace std;

namespace {
    long triple(long val)
    {
        return 3*val;
    }
    ///////////////////////////////////////////////////////////////////////////
    //
```

```
    // actions
    //
    struct emit_constant
    {
        emit_constant(string const &text)
            : msg(text)
        {}

        template<typename Iterator>
        void operator()(Iterator b, Iterator e) const
        {
            cout.rdbuf()->sputn(msg.data(), msg.size());
        }

    private:

        string msg;
    };

    void
    copy_unmodified(char letter)
    {
        cout.rdbuf()->sputc(letter);
    }

    struct emit_modified_subscript
    {
        emit_modified_subscript(boost::function<long (long)> const &f)
            : modifier(f)
        {}

        template<typename Iterator>
        void operator()(Iterator b, Iterator e) const
        {
            string tmp(b+1,e-1);
            long val = strtol(tmp.c_str(),0, 0);
            ostringstream os;
            os << modifier(val);
            tmp = os.str();
            cout.rdbuf()->sputc(’[’);
            cout.rdbuf()->sputn(tmp.c_str(), tmp.size());
            cout.rdbuf()->sputc(’]’);
        }

    private:

        boost::function<long (long)> modifier;
    };
}

///////////////////////////////////////////////////////////////////////////////
//  The grammar ’conversion_grammar’ serves as a working horse for match based
//  text conversion. It does the following:
//
//      - converts the word "class" into the word "struct"
//      - multiplies any integer number enclosed in square brackets with 3
//      - any other input is simply copied to the output

struct conversion_grammar
     : grammar<conversion_grammar>
{
    template<class ScannerT>
    struct definition
    {
        typedef ScannerT scanner_t;

        definition(conversion_grammar const &)
        {
            static const char expr[] = "\\[\\d+\\]";
            first = (
                //////////////////////////////////////////////////////////////
                // note that "fallback" is the last alternative here !
                top  = *(class2struct || subscript || fallback),
                //////////////////////////////////////////////////////////////
                // replace any occurrance of "class" by "struct"
                class2struct = str_p("class") [emit_constant("struct")],
                //////////////////////////////////////////////////////////////
                // if the input maches "[some_number]"
                // "some_number" is multiplied by 3 before printing
                subscript = regex_p(expr) [emit_modified_subscript(&triple)],
                //////////////////////////////////////////////////////////////
                // if nothing else can be done with the input
                // then it will be printed without modifications
                fallback = anychar_p [&copy_unmodified]
            );
        }

        rule<scanner_t> const & start() { return first; }

    private:

        subrule<0> top;
        subrule<1> class2struct;
        subrule<2> subscript;
        subrule<3> fallback;
        rule<scanner_t> first;
    };
};

int
main()
{
    //  this would print "struct foo {}; foo bar[9];":
    //  parse("class foo {}; foo bar[3];", conversion_grammar());

    // Note: the regular expression parser contained in the
    //       grammar requires a bidirectional iterator. Therefore,
    //       we cannot use sdt::istreambuf_iterator as one would
    //       do with other Spirit parsers.
    istreambuf_iterator<char> input_iterator(cin);
    std::deque<char> input(input_iterator, istreambuf_iterator<char>());

    parse(input.begin(), input.end(), conversion_grammar());
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2003 Jonathan de Halleux (dehalleux@pelikhan.com)
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
///////////////////////////////////////////////////////////////////////////////
// This example shows how the assign operator can be used to modify
// rules with semantic actions
//
// First we show the basic spirit without (without any dynamic feature),
// then we show how to use assign_a to make it dynamic,
//
// the grammar has to parse abcabc... sequences
///////////////////////////////////////////////////////////////////////////////
#include <iostream>

#define BOOST_SPIRIT_DEBUG
#include <boost/spirit.hpp>

#include <boost/spirit/actor/assign_actor.hpp>

int main(int argc, char* argv[])
{
    using namespace boost::spirit;
    using namespace std;

    rule<> a,b,c,next;
    const char* str="abcabc";
    parse_info<> hit;
    BOOST_SPIRIT_DEBUG_NODE(next);
    BOOST_SPIRIT_DEBUG_NODE(a);
    BOOST_SPIRIT_DEBUG_NODE(b);
    BOOST_SPIRIT_DEBUG_NODE(c);

    // basic spirit gram
    a = ch_p('a') >> !b;
    b = ch_p('b') >> !c;
    c = ch_p('c') >> !a;

    hit = parse(str, a);
    cout<<"hit :"<<( hit.hit ? "yes" : "no")<<", "
        <<(hit.full ? "full": "not full")
        <<endl;

    // using assign_a
    a = ch_p('a')[ assign_a( next, b)] >> !next;
    b = ch_p('b')[ assign_a( next, c)] >> !next;
    c = ch_p('c')[ assign_a( next, a)] >> !next;
    hit = parse(str, a);
    cout<<"hit :"<<( hit.hit ? "yes" : "no")<<", "
        <<(hit.full ? "full": "not full")
        <<endl;

    return 0;
}
```

```
/*==============================================================================
    Copyright (c) 2003 Vaclav Vesely
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
==============================================================================*/
//
//  This example demonstrates the behaviour of epsilon_p when used as parser
//  generator.
//
//  The "r" is the rule, which is passed as a subject to the epsilon_p parser
//  generator. The "r" is the rule with binded semantic actions. But epsilon_p
//  parser generator is intended for looking forward and we don’t want to
//  invoke semantic actions of subject parser. Hence the epsilon_p uses
//  the no_actions policy.
//
//  Because we want to use the "r" rule both in the epsilon_p and outside of it
//  we need the "r" to support two differant scanners, one with no_actions
//  action policy and one with the default action policy. To achieve this
//  we declare the "r" with the no_actions_scanner_list scanner type.
//
//------------------------------------------------------------------------------
#define BOOST_SPIRIT_RULE_SCANNERTYPE_LIMIT 2

#include <cassert>
#include <iostream>
#include <boost/cstdlib.hpp>
#include <boost/spirit/core.hpp>
#include <boost/spirit/phoenix.hpp>

using namespace std;
using namespace boost;
using namespace spirit;
using namespace phoenix;

//------------------------------------------------------------------------------

int main()
{
    rule<
        // Support both the default phrase_scanner_t and the modified version
        // with no_actions action_policy
        no_actions_scanner_list<phrase_scanner_t>::type
    > r;

    int i(0);

    r = int_p[var(i) += arg1];

    parse_info<> info = parse(
        "1",

        // r rule is used twice but the semantic action is invoked only once
        epsilon_p(r) >> r,

        space_p
    );

    assert(info.full);
    // Check, that the semantic action was invoked only once
```

```
    assert(i == 1);

    return exit_success;
}

//------------------------------------------------------------------------------
```

```
/*==============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
==============================================================================*/

// *** See the section "Rule With Multiple Scanners" in
// *** chapter "Techniques" of the Spirit documentation
// *** for information regarding this snippet

#define BOOST_SPIRIT_RULE_SCANNERTYPE_LIMIT 3

#include <iostream>
#include <boost/spirit/core.hpp>

using namespace boost::spirit;

struct my_grammar : grammar<my_grammar>
{
    template <typename ScannerT>
    struct definition
    {
        definition(my_grammar const& self)
        {
            r = lower_p;
            rr = +(lexeme_d[r] >> as_lower_d[r] >> r);
        }

        typedef scanner_list<
            ScannerT
          , typename lexeme_scanner<ScannerT>::type
          , typename as_lower_scanner<ScannerT>::type
        > scanners;

        rule<scanners> r;
        rule<ScannerT> rr;
        rule<ScannerT> const& start() const { return rr; }
    };
};

int
main()
{
    my_grammar g;
    bool success = parse("abcdef aBc d e f aBc d E f", g, space_p).full;
    assert(success);
    std::cout << "SUCCESS!!!\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2003 Sam Nabialek
    Copyright (c) 2003-2004 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
#include <iostream>

#define BOOST_SPIRIT_DEBUG

#include <boost/spirit/core.hpp>
#include <boost/spirit/error_handling.hpp>
#include <boost/spirit/iterator.hpp>
#include <boost/spirit/symbols.hpp>
#include <boost/spirit/utility.hpp>

using namespace boost::spirit;

template <typename Rule>
struct SetRest
{
    SetRest(Rule& the_rule)
    : m_the_rule(the_rule)
    {
    }

    void operator()(Rule* new_rule) const
    {
        m_the_rule = *new_rule;
    }

private:

    Rule& m_the_rule;
};


struct nabialek_trick : public grammar<nabialek_trick>
{
    template <typename ScannerT>
    struct definition
    {
        typedef rule<ScannerT> rule_t;

        rule_t name;
        rule_t line;
        rule_t rest;
        rule_t main;
        rule_t one;
        rule_t two;

        symbols<rule_t*> continuations;

        definition(nabialek_trick const& self)
        {
            name = lexeme_d[repeat_p(1,20)[alnum_p | ’_’]];

            one = name;
            two = name >> ’,’ >> name;
```

```
            continuations.add
                ("one", &one)
                ("two", &two)
                ;

            line = continuations[SetRest<rule_t>(rest)] >> rest;
            main = *line;

            BOOST_SPIRIT_DEBUG_RULE(name);
            BOOST_SPIRIT_DEBUG_RULE(line);
            BOOST_SPIRIT_DEBUG_RULE(rest);
            BOOST_SPIRIT_DEBUG_RULE(main);
            BOOST_SPIRIT_DEBUG_RULE(one);
            BOOST_SPIRIT_DEBUG_RULE(two);
        }

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

int
main()
{
    nabialek_trick g;
    parse("one only\none again\ntwo first,second", g, space_p);
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2005 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/
#include <iostream>

#define BOOST_SPIRIT_DEBUG
#include <boost/spirit/core.hpp>

using namespace boost::spirit;

struct non_greedy_kleene : public grammar<non_greedy_kleene>
{
    template <typename ScannerT>
    struct definition
    {
        typedef rule<ScannerT> rule_t;
        rule_t r;

        definition(non_greedy_kleene const& self)
        {
            r = (alnum_p >> r) | digit_p;
            BOOST_SPIRIT_DEBUG_RULE(r);
        }

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

struct non_greedy_plus : public grammar<non_greedy_plus>
{
    template <typename ScannerT>
    struct definition
    {
        typedef rule<ScannerT> rule_t;
        rule_t r;

        definition(non_greedy_plus const& self)
        {
            r = alnum_p >> (r | digit_p);
            BOOST_SPIRIT_DEBUG_RULE(r);
        }

        rule_t const&
        start() const
        {
            return r;
        }
    };
};
int
main()
{
    bool success;
```

```
    {
        non_greedy_kleene k;
        success = parse("3", k).full;
        assert(success);
        success = parse("abcdef3", k).full;
        assert(success);
        success = parse("abc2def3", k).full;
        assert(success);
        success = parse("abc", k).full;
        assert(!success);
    }

    {
        non_greedy_plus p;
        success = parse("3", p).full;
        assert(!success);
        success = parse("abcdef3", p).full;
        assert(success);
        success = parse("abc2def3", p).full;
        assert(success);
        success = parse("abc", p).full;
        assert(!success);
    }

    std::cout << "SUCCESS!!!\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/

// *** See the section "typeof" in chapter "Techniques" of
// *** the Spirit documentation for information regarding
// *** this snippet.

#ifdef __MWERKS__
#define typeof __typeof__
#endif

#if !defined(__MWERKS__) && !defined(__GNUC__)
#error "typeof not supported by your compiler"
#endif

#include <iostream>
#include <boost/spirit/core.hpp>

using namespace boost::spirit;

#define RULE(name, definition) typeof(definition) name = definition

int
main()
{
    RULE(
        skipper,
        (       space_p
            |   "//" >> *(anychar_p - '\n') >> '\n'
            |   "/*" >> *(anychar_p - "*/") >> "*/"
        )
    );

    bool success = parse(
        "/*this is a comment*/\n//this is a c++ comment\n\n",
        *skipper).full;
    assert(success);
    std::cout << "SUCCESS!!!\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/

// *** See the section "Look Ma’ No Rules" in
// *** chapter "Techniques" of the Spirit documentation
// *** for information regarding this snippet

#include <iostream>
#include <boost/spirit/core.hpp>

using namespace boost::spirit;

    struct skip_grammar : grammar<skip_grammar>
    {
        template <typename ScannerT>
        struct definition
        {
            definition(skip_grammar const& /*self*/)
            {
                skip
                    =   space_p
                    |   "//" >> *(anychar_p - ’\n’) >> ’\n’
                    |   "/*" >> *(anychar_p - "*/") >> "*/"
                    ;
            }

            rule<ScannerT> skip;

            rule<ScannerT> const&
            start() const { return skip; }
        };
    };

int
main()
{
    skip_grammar g;
    bool success = parse(
        "/*this is a comment*/\n//this is a c++ comment\n\n",
        *g).full;
    assert(success);
    std::cout << "SUCCESS!!!\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/

// *** See the section "Look Ma’ No Rules" in
// *** chapter "Techniques" of the Spirit documentation
// *** for information regarding this snippet

#include <iostream>
#include <boost/spirit/core.hpp>

using namespace boost::spirit;

struct skip_grammar : grammar<skip_grammar>
{
    template <typename ScannerT>
    struct definition
    {
        definition(skip_grammar const& /*self*/)
        : skip
            (       space_p
                |   "//" >> *(anychar_p - ’\n’) >> ’\n’
                |   "/*" >> *(anychar_p - "*/") >> "*/"
            )
        {
        }

        typedef
           alternative<alternative<space_parser, sequence<sequence<
           strlit<const char*>, kleene_star<difference<anychar_parser,
           chlit<char> > > >, chlit<char> > >, sequence<sequence<
           strlit<const char*>, kleene_star<difference<anychar_parser,
           strlit<const char*> > > >, strlit<const char*> > >
        skip_t;
        skip_t skip;

        skip_t const&
        start() const { return skip; }
    };
};

int
main()
{
    skip_grammar g;
    bool success = parse(
        "/*this is a comment*/\n//this is a c++ comment\n\n",
        *g).full;
    assert(success);
    std::cout << "SUCCESS!!!\n";
    return 0;
}
```

```
/*=============================================================================
    Copyright (c) 2002-2003 Joel de Guzman
    http://spirit.sourceforge.net/

    Use, modification and distribution is subject to the Boost Software
    License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at
    http://www.boost.org/LICENSE_1_0.txt)
=============================================================================*/

// *** See the section "Look Ma' No Rules" in
// *** chapter "Techniques" of the Spirit documentation
// *** for information regarding this snippet

#include <iostream>
#include <boost/spirit/core.hpp>

using namespace boost::spirit;

namespace boost { namespace spirit
{
    template <typename DerivedT>
    struct sub_grammar : parser<DerivedT>
    {
        typedef sub_grammar         self_t;
        typedef DerivedT const&     embed_t;

        template <typename ScannerT>
        struct result
        {
            typedef typename parser_result<
                typename DerivedT::start_t, ScannerT>::type
            type;
        };

        DerivedT const& derived() const
        { return *static_cast<DerivedT const*>(this); }

        template <typename ScannerT>
        typename parser_result<self_t, ScannerT>::type
        parse(ScannerT const& scan) const
        {
            return derived().start.parse(scan);
        }
    };
}}

///////////////////////////////////////////////////////////////////////////////
//
//  Client code
//
///////////////////////////////////////////////////////////////////////////////
struct skip_grammar : sub_grammar<skip_grammar>
{
    typedef
       alternative<alternative<space_parser, sequence<sequence<
       strlit<const char*>, kleene_star<difference<anychar_parser,
       chlit<char> > > >, chlit<char> > >, sequence<sequence<
       strlit<const char*>, kleene_star<difference<anychar_parser,
       strlit<const char*> > > >, strlit<const char*> > >
    start_t;

    skip_grammar()
```

```
    : start
        (
            space_p
        |   "//" >> *(anychar_p - '\n') >> '\n'
        |   "/*" >> *(anychar_p - "*/") >> "*/"
        )
    {}

    start_t start;
};

int
main()
{
    skip_grammar g;

    bool success = parse(
        "/*this is a comment*/\n//this is a c++ comment\n\n",
        *g).full;
    assert(success);
    std::cout << "SUCCESS!!!\n";
    return 0;
}
```