```
#pragma once
////////////////////////////////////////////////////////////////////////////
//Copyright (c) 2010-2013, Ugo Varetto
//All rights reserved.
//
//Redistribution and use in source and binary forms, with or without
//modification, are permitted provided that the following conditions are met:
//    * Redistributions of source code must retain the above copyright
//      notice, this list of conditions and the following disclaimer.
//    * Redistributions in binary form must reproduce the above copyright
//      notice, this list of conditions and the following disclaimer in the
//      documentation and/or other materials provided with the distribution.
//    * Neither the name of the copyright holder nor the
//      names of its contributors may be used to endorse or promote products
//      derived from this software without specific prior written permission.
//
//THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
//ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
//WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
//DISCLAIMED. IN NO EVENT SHALL UGO VARETTO BE LIABLE FOR ANY
//DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
//(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
//LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
//ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
//(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
//SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
////////////////////////////////////////////////////////////////////////////

/// @file Parser.h Definition of Parser interface and implementation of basic
/// parser classes.

#include <list>
#include <map>
#include <memory>
#include "types.h"
#include "Any.h"

namespace parsley {

/// @interface IParser Parser.h Public interface for parsers.
struct IParser {
    /// Value type in map.
    typedef Values::value_type::second_type ValueType;
    /// Key type in map.
    typedef Values::key_type KeyType;
    /// Returns extracted values if any.
    /// @return constant reference to extracted values.
    virtual const Values& GetValues() const = 0;
    /// Returns value identified by specific key. This method should throw an
    /// @c std::logic_error exception if key not found.
    /// @return constant reference to values from key.
    /// @exception std::logic error
    virtual const ValueType& operator[]( const KeyType& k ) const = 0;
    /// Performs parsing on input stream.
    virtual bool Parse( InStream& ) = 0;
    /// Returns copy of current instance.
    /// @return pointer to newly created instance.
    virtual IParser* Clone() const = 0;
    virtual ~IParser() {}
};

//----------------------------------------------------------------------------
/// @brief Implementation of IParser interface.
///
/// Allows value semantics with polymorphic types through the @c pImpl idiom:
/// - an implementation of IParser is stored internally
/// - all calls to IParser methods are forwarded to the containted IParser
///   implementation
/// the contained IParser instance is owned my Parser objects and deleted when
/// Parser deleted.
/// Note that contained parsers are never shared: new copies are created during
/// construction and assignment.
/// @ingroup MainClasses
```

```
class Parser : public IParser {
public:
    typedef Values::value_type::second_type ValueType;
    typedef Values::key_type KeyType;
    typedef InCharStream InStream;
    /// Returns @c false if no contained implementation available, true otherwise.
    /// @return @c false if internal pointer to IParser instance is @c NULL
    ///         @c true otherwise.
    bool Empty() const { return pImpl_ == 0; }
    /// Default constructor.
    Parser() {}
    /// Copy constructor.
    Parser( const Parser& l ) : pImpl_( l.pImpl_ ? l.pImpl_->Clone() : 0 ) {}
    /// Swap: swap internal pointers to IParser implementations.
    /// @return reference to @c *this after swap.
    Parser& Swap( Parser& l ) {
        std::swap( pImpl_, l.pImpl_ );
        return *this;
    }
    /// Assignment operator from other parser.
    Parser& operator=( const Parser& l ) {
        Parser( l ).Swap( *this ); return *this;
    }
    /// Constructor from IParser implementation.
    Parser( const IParser& l ) : pImpl_( l.Clone() ) {}
    /// Implementation of IParser::GetValues.
    const Values& GetValues() const {
        CheckPointer();
        return pImpl_->GetValues();
    }
    /// Implementation of IParser::operator[]
    const ValueType& operator[]( const KeyType& k ) const {
        CheckPointer();
        return pImpl_->operator[]( k );
    }
    /// Implementation of IParser::Parse.
    bool Parse( InStream& is ) {
        CheckPointer();
        if( !pImpl_ ) return false;
        return pImpl_->Parse( is );
    }
    /// Implementation of IParser::Clone.
    Parser* Clone() const { return new Parser( *this ); }
    /// Returns @c true if pImpl_ non-null
    bool Valid() const { return pImpl_.get() != 0; }

private:
    /// Check internal pointer to owned IParser instance.
    /// @return @c true if internal pointer not @c NULL, @c false otherwise.
    /// @exception std::runtime_error if pointer @c NULL.
    bool CheckPointer() const {
        assert( pImpl_ );
        if( !pImpl_ ) throw std::runtime_error( "NULL Parser pointer" );
        if( !pImpl_ ) return false; //release mode, no exceptions
        return true;
    }
    /// Owned instance of IParser to which all IParser method calls are
    /// forwarded.
    std::unique_ptr< IParser > pImpl_;
};

//---------------------------------------------------------------------------
/// @brief Utility class used to rewind an input stream if a monitored values
/// is @c false.
///
/// The input stream get pointer position is recorded in the constructor and r
/// reset in the destructor if the condition value is @c false.
/// @tparam IstreamT input stream type e.g. @c std::istream.
///
/// Usage:
/// @code
/// std::ifstream ifs("file.txt");
/// bool ok = false;
```

```
/// ...
/// {
/// // record position
/// RewindManager< std::istream > RM( ok, ifs );
/// //read operations; set ok to false if input not validated.
/// ...
/// }
/// //rewind if condition not met
/// @endcode
template< class IstreamT > class RewindManager {
    /// Position in stream.
    typedef typename IstreamT::streampos StreamPos;
public:
    /// Constructor. The current input stream position is recorded.
    /// @param flag monitored value.
    /// @param is input stream reference to act on.
    RewindManager( bool& flag, IstreamT& is )
        : pos_( is.tellg() ), flag_( flag ), is_( is ) {}
    /// Destructor.
    /// When object goes out of scope the monitored value is checked:
    /// if the condition value is @c false the input stream get pointer
    /// is rewound to the initial position recorded in the constructor.
    ~RewindManager() {
        if( !flag_ ) {
//#ifdef __GNUC__ // note: gcc requires eof flag not set for seekg to work
                  //       doesn't arm to force a clear on vc++ either
            is_.clear();
//#endif
            is_.seekg( pos_ );
        }
    } //rewind if condition not met
private:
    /// Stream get pointer position.
    const StreamPos pos_;
    /// Condition flag reference.
    const bool& flag_;
    /// Input stream reference.
    IstreamT& is_;
};

/// Convenience typedef.
typedef RewindManager< InStream > REWIND;

//-------------------------------------------------------------------------------
/// @brief Applies a sequence of parsers to the input stream in the order
/// supplied by the client code, optionally skipping leading blanks.
class AndParser : public IParser {
public:
    typedef Values::value_type::second_type ValueType;
    typedef Values::key_type KeyType;
    /// Default constructor; enables skipping of leading blanks.
    AndParser() : skipBlanks_( true ) {}
    /// Copy constructor.
    AndParser( const AndParser& p )
        : parsers_( p.parsers_ ), valueMap_( p.valueMap_ ),
          skipBlanks_( p.skipBlanks_ ) {}
    /// Constructor.
    /// @param b enable skipping of trailing blanks flag.
    AndParser( bool b ) : skipBlanks_( b ) {}
    /// Add new parser as the first element in the sequence of parsers to apply.
    /// @param p parser to insert.
    /// @return reference to @c *this.
    AndParser& AddFront( const Parser& p ) {
        parsers_.push_front( p );  return *this;
    }
    /// Append parser to parser sequence.
    /// @param p parser to append.
    /// @return reference to @c *this.
    AndParser& Add( const Parser& p ) {
        parsers_.push_back( p );  return *this;
    }
    /// Implementation of IParser::Parse. Returns @c true if and only if all
    /// the parsers in the sequnce return true; @c false otherwise.
```

```
    bool Parse( InStream& is ) {
        valueMap_.clear();
        //if( !is.good() ) return false;
        bool ok = false; {
        REWIND r( ok, is );
        for( Parsers::iterator i = parsers_.begin();
             i != parsers_.end();
             ++i ) {
            if( skipBlanks_) SkipBlanks( is );
            if( !i->Parse( is ) ) return false;
        }
        ok = true;
        }
        return ok;
    }
    /// Used to chain parsers.
    AndParser& operator >=( const Parser& p ) { return Add( p ); }
    /// Implementation of IParser::GetValues.
    /// Returns all the values parsed by provided parsers. Lazy behavior: values
    /// are retrieved from parsers the first time this method is invoked.
    const Values& GetValues() const {
        if( valueMap_.empty() ) AppendValues();
        return valueMap_;
    }
    /// Implementation of IParser::operator[]. Invokes GetValues() first.
    /// @exception std::logic_error if key not found.
    const ValueType& operator[]( const KeyType& k ) const {
        const Values& v = GetValues();
        Values::const_iterator i = v.find( k );
        if( i == v.end() ) throw std::logic_error( "Cannot find value" );
        return i->second;
    }

    /// Implementation of IParser::Clone.
    AndParser* Clone() const { return new AndParser( *this ); }

private:
    /// Appends values extracted from child parsers to the value map.
    void AppendValues() const {
        for( Parsers::const_iterator i = parsers_.begin();
             i != parsers_.end();
             ++i ) {
            if( i->GetValues().empty() ) continue;
            valueMap_.insert( i->GetValues().begin(), i->GetValues().end() );
        }
    }
    /// Advance to first non-blank character.
    void SkipBlanks( InStream& is ) {
        if( !is.good() ) return;
        Char c = is.get();
        while( is.good() && parsley::IsSpace( c ) != 0 ) c = is.get();
        if( is.good() ) is.unget();
    }
    /// Parser list type.
    typedef std::list< Parser > Parsers;
    /// Parser list.
    Parsers parsers_;
    /// Extracted values.@c mutable is required to allow for lazy behavior:
    /// the map is populated only at first call of MultiMap::GetValues method.
    mutable Values valueMap_;
    /// Skip blanks flag.
    bool skipBlanks_;
};


//-----------------------------------------------------------------------------
/// @brief Applies the same parser a fixed number of times or until it fails.
class MultiParser : public IParser {
public:
    typedef Values::value_type::second_type ValueType;
    typedef Values::key_type KeyType;
    /// Constructor.
    /// @param p parser to execute.
```

```
    /// @param name value identifier. This is the key used to identify the
    ///         list of parsed values.
    /// @param countMin minimum amount of times the parser is invoked
    /// @param countMax maximum amount of times the parser is invoked.
    ///        If a number < 0 is specifie the parser is invoked until it fails.
    MultiParser( const IParser& p, const ValueID& name = ValueID(),
                 int countMin = 1, int countMax = -1 )
        : parser_( p ), countMin_( countMin ), countMax_( countMax ) {}
    /// Implementation of IParser::Parse. Invokes the contained parser at least
    /// MultiParser#countMin_ times, and no more than MultiParser#countMax_
    /// times or until parsing fails if  MultiParser#countMax_ < 0.
    /// @param is input stream.
    /// @return true if parser invoked at least MultiParser#countMin_ times and
    ///    no more than MultiParser#countMax_ times or invoked at least
    ///    MultiParser#countMax_ times and MultiParser#countMax_ < 0.
    bool Parse( InStream& is ) {
        valueMap_.clear();
        bool ok = false; {
        REWIND r( ok, is );
        int counter = 0;
        while( is.good() &&
               ( counter < countMax_ || countMax_ < 0 ) &&
               parser_.Parse( is ) ) {
            ++counter;
            const Values& v = parser_.GetValues();
            for( Values::const_iterator i = v.begin(); i != v.end(); ++i ) {
                values_.push_back( i->second );
            }
        }
        if( counter < countMin_ || ( counter > countMax_ && countMax_ >= 0 ) ) {
            ok = false;
        }
        else ok = true; // counter in [countMin_, countMax_]
        }
        return ok;
    }
    /// Convenience operator to set the minimum and maximum amount of times the
    /// parser should be invoked.
    MultiParser& operator()( int minCount, int maxCount = -1 )
    { countMin_ = minCount; countMax_ = maxCount; return *this; }
    /// Implementation of IParser::GetValues. Returns a single (key, value) pair
    /// where the key is the name specified in the constructor and the value is
    /// a list of parsed values.
    /// @return (key,value) pair where key == MultiParser::name_.
    const Values& GetValues() const {
        if( valueMap_.empty() ) {
            valueMap_.insert( std::make_pair( name_, values_ ) );
        }
        return valueMap_;
    }
    /// Implementation of IParser::operator[].
    /// @exception std::logic_error if key not in map.
    const ValueType& operator[]( const KeyType& k ) const {
        const Values& v = GetValues();
        Values::const_iterator i = v.find( k );
        if( i == v.end() ) throw std::logic_error( "Cannot find value" );
        return i->second;
    }

    /// Implementation of IParser::Clone.
    MultiParser* Clone() const { return new MultiParser( *this ); }

private:
    /// Identifier for parsed value list in returned value map.
    ValueID name_;
    /// Parser to apply.
    Parser parser_;
    /// Minimum amount of times to execute parser.
    int countMin_;
    /// Maximum amount of times to execute parser or < 0 to signal
    /// an indefinite amount.
    int countMax_;
    /// Value map: one element with key == MultiParser#name_ and
```

```
    /// value == @c std::list of parsed values. @c mutable is required to allow
    /// for lazy behavior: the map is populated only at first call of
    /// MultiMap::GetValues method.
    mutable Values valueMap_;
    /// List of parsed values. Values are appended to list after each invocation
    /// of MultiParser#parser_.Parse().
    std::list< Any > values_;
};

/// Convenience operator to mimic regex syntax and generate a multiparser
/// that applies multiple times a given parser.
MultiParser operator*( const Parser& p ) { return MultiParser( p ); }


//------------------------------------------------------------------------------
/// @brief Parse method returns the negation of what the contained parser Parse
/// method returns.
///
/// The validated character sequence is stored internally into a string
/// and can be retrieved through the NotParser::GetValues method.
/// @tparam Parser type.
template < class ParserT > class NotParser : public IParser {
public:
    typedef Values::value_type::second_type ValueType;
    typedef Values::key_type KeyType;
    /// Implementation of IParser::GetValues.
    /// @return one @c (key,value) pair where @c key is the name specified in
    /// the constructor or the empty string and @c value is a string containing
    /// the parsed characters.
    const Values& GetValues() const { return valueMap_; }
    NotParser* Clone() const { return new NotParser< ParserT >( *this ); }
    bool Parse( InStream& is ) {
        // if ParserT::Parse returns true
        // the get pointer will point one char
        // past the end of the validated char sequence,
        // it is therefore required to save the pointer
        // before the invocation and restore it in case
        // the parser validates the string to make it point
        // to one char before the beginning of the string to
        // validate
        valueMap_.clear();
        String s;
        Char c = 0;
        StreamPos pos = is.tellg();
        while( is.good() && !parser_.Parse( is ) ) {
            c = is.get();
            if( !is.good() ) break;
            s.push_back( c );
            pos = is.tellg();
        }
        if( is.good() ) is.seekg( pos );
        if( s.length() > 0 ) valueMap_.insert( std::make_pair( name_, s ) );
        return s.length() > 0;
    }

    /// Implementation of IParser::operator[].
    /// @exception std::logic error if key not found.
    const ValueType& operator[]( const KeyType& k ) const {
        const Values& v = GetValues();
        Values::const_iterator i = v.find( k );
        if( i == v.end() ) throw std::logic_error( "Cannot find value" );
        return i->second;
    }
    /// (Default) constructor.
    /// @param name key indentifier for parsed text.
    NotParser( const ValueID& name = ValueID() ) : name_( name ) {}
    /// Constructor. Allows for construction from any value that can be used to
    /// construct the contained parser type.
    template < class T > NotParser( const T& v,
                                    const ValueID& name = ValueID() )
        : parser_( v ), name_( name ) {}
private:
    /// Contained parser.
```

```
    ParserT parser_;
    /// Value map: contains a @c (NotParser#name_,String) pair.
    Values valueMap_;
    /// Value identifier used as key in value map.
    ValueID name_;
};


//------------------------------------------------------------------------------
/// @brief Parser for optional expressions.
///
/// The Parse method applies the contained parser and always returns true.
class OptionalParser : public IParser {
public:
    typedef Values::value_type::second_type ValueType;
    typedef Values::key_type KeyType;
    OptionalParser( const Parser& p, const ValueID& name = ValueID() )
        : parser_( p ), name_( name ) {}
    /// Implementation of IParser::GetValues.
    /// @return one @c (key,value) pair where @c key is the name specified in
    /// the constructor or the empty string and @c value is a string containing
    /// the parsed characters.
    const Values& GetValues() const { return valueMap_; }
    OptionalParser* Clone() const { return new OptionalParser( *this ); }
    /// Implementation of IParser::Parse: apply parser and always returns true.
    bool Parse( InStream& is ) {
        valueMap_.clear();
        if( parser_.Parse( is ) ) valueMap_ = parser_.GetValues();
        return true;
    }
    /// Implementation of IParser::operator[].
    /// @exception std::logic error if key not found.
    const ValueType& operator[]( const KeyType& k ) const {
        const Values& v = GetValues();
        Values::const_iterator i = v.find( k );
        if( i == v.end() ) throw std::logic_error( "Cannot find value" );
        return i->second;
    }
    /// Constructor. Allows for construction from any value that can be used
    /// to construct the contained parser type.
    template < class T > OptionalParser( const T& v,
                                         const ValueID& name = ValueID() )
        : parser_( v ), name_( name ) {}

private:
    /// Contained parser.
    Parser parser_;
    /// Value map: contains a @c (NotParser#name_,String) pair.
    Values valueMap_;
    /// Value identifier used as key in value map.
    ValueID name_;
};


//------------------------------------------------------------------------------
/// @brief Ordered OR parser: invokes the contained parsers until one returns
/// true.
///
/// The values parsed by the first parser that returns true are accessible
/// through the OrParser::GetValue method.
class OrParser : public IParser {
public:
    typedef Values::value_type::second_type ValueType;
    typedef Values::key_type KeyType;
    /// Append new parser to parser list.
    /// @param p parser to append to parser list.
    /// @return reference to @c *this.
    OrParser& Add( const Parser& p ) { parsers_.push_back( p ); return *this; }
    /// Invokes OrParser::Add(const Parser&).
    OrParser& operator/=( const Parser& p ) { return Add(p); }
    /// Convenience method to apped a parser non contained into a Parser
    /// instance.
    OrParser& operator/=( const IParser& p ) { return Add( p ); }
```

```
    /// Default constructor.
    OrParser() : matchedParser_( parsers_.end() ) {}
    /// Copy constructor.
    OrParser( const OrParser& op )
        : parsers_( op.parsers_ ), matchedParser_( parsers_.end() ) {}
    /// Implementation of IParser::Parse.
    /// Iterates of alternative parsers and stops at the first parser that
    /// parses the input.
    /// @param is input stream.
    /// @return true if at least one one parser is successful, false otherwise.
    bool Parse( InStream& is ) {
        matchedParser_ = parsers_.end();
        bool ok = false; {
        REWIND r( ok, is );
        Parsers::iterator i = parsers_.begin();
        for( ; i != parsers_.end(); ++i ) {
            const StreamPos pos = is.tellg();
            if( !i->Parse( is ) ) {
                is.seekg( pos );
                continue;
            }
            else break;
        }
        matchedParser_ = i;
        ok = i != parsers_.end();
        }
        return ok;
    }
    /// Implementation of IParser::GetValues.
    /// @return values parsed by matching parser of emtpy value map.
    const Values& GetValues() const {
        static const Values dummy;
        return matchedParser_ != parsers_.end() ?
               matchedParser_->GetValues() : dummy;
    }
    /// Implementation of IParser::operator[].
    /// @throw std::logic_error if key not found.
    const ValueType& operator[]( const KeyType& k ) const {
        const Values& v = GetValues();
        Values::const_iterator i = v.find( k );
        if( i == v.end() ) throw std::logic_error( "Cannot find value" );
        return i->second;
    }

    /// Implemetation of IParser::Clone.
    OrParser* Clone() const { return new OrParser( *this ); }

private:
    /// Parser list type.
    typedef std::list< Parser > Parsers;
    /// Alternative parsers.
    Parsers parsers_;
    /// Reference to matched parser or to @c parsers_.end() if no match found.
    Parsers::const_iterator matchedParser_;
};


//------------------------------------------------------------------------------
/// @brief Or Parser: applies a set of parsers to the given stream and validates
/// the input only if at least one of the applied parser validates the input
/// characters.
///
/// The parser selected as the validating parser is the one that  validates
/// the maximum number of characters.
/// At the end of the parsing operation the get pointer is positioned:
/// - at the same position where it was before calling the Parser method if
///   no parser that validates the input is found
/// - one character past the last character validated by the parser that
///   validates the maximum number of characters.
class GreedyOrParser : public IParser {
public:
    typedef Values::value_type::second_type ValueType;
    typedef Values::key_type KeyType;
```

```
    /// Default constructor.
    GreedyOrParser() : matchedParser_( parsers_.end() ) {}
    /// Copy constructor.
    GreedyOrParser( const GreedyOrParser& op )
        : parsers_( op.parsers_ ), matchedParser_( parsers_.end() ) {}
    /// Add parser to the parser list.
    /// @param p parser to append to parser list.
    /// @return reference to @c *this.
    GreedyOrParser& Add( const Parser& p ) {
        parsers_.push_back( p ); return *this;
    }
    /// Implementation of IParser::Parse method: applies each parser to the
    /// input stream and selects the parser that parses the most input or
    /// returns @c false if no validating parser found.
    bool Parse( InStream& is ) {
        matchedParser_ = parsers_.end();
        matchedParsers_.clear();
        bool ok = false; {
        REWIND r( ok, is );
        Parsers::iterator i = parsers_.begin();
        for( ; i != parsers_.end(); ++i ) {
            const StreamOff pos = is.tellg();
            if( i->Parse( is ) ) {
                matchedParsers_.insert( std::make_pair( is.tellg(), i ) );
            }
            is.seekg( pos );

        }
        matchedParser_ = matchedParsers_.empty() ?
                         parsers_.end() : ( --matchedParsers_.end() )->second;
        ok = i != parsers_.end();
        if( ok ) is.seekg( ( --matchedParsers_.end() )->first );
        }
        return ok;
    }
    /// Implementation of IParser::GetValues. Returns the values parsed by the
    /// selected parser.
    const Values& GetValues() const {
        static const Values dummy;
        return matchedParser_ != parsers_.end() ?
                                 matchedParser_->GetValues() : dummy;
    }
    /// Implementation of IParser::operator[].
    /// @throw std::logic_error if key not found.
    const ValueType& operator[]( const KeyType& k ) const {
        const Values& v = GetValues();
        Values::const_iterator i = v.find( k );
        if( i == v.end() ) throw std::logic_error( "Cannot find value" );
        return i->second;
    }
    /// Implementation of IParser::Clone.
    GreedyOrParser* Clone() const { return new GreedyOrParser( *this ); }
private:
    /// Parser list type.
    typedef std::list< Parser > Parsers;
    /// Parser list.
    Parsers parsers_;
    /// Reference to matching parser or to @c parsers_.end() if no suitable
    /// parser found.
    Parsers::const_iterator matchedParser_;
    std::map< StreamOff, Parsers::const_iterator > matchedParsers_;
};

//------------------------------------------------------------------------------
/// @brief Applies a parser until it validates, eof is reached or a terminal
/// condition parser validates.
class GreedyParser : public IParser {
public:
    typedef Values::value_type::second_type ValueType;
    typedef Values::key_type KeyType;
    /// Copy constructor.
    GreedyParser( const GreedyParser& p )
        : parser_( p.parser_ ), terminalParser_( p.terminalParser_ ) {}
```

```
    /// Constructor.
    /// @param p parser to apply
    /// @param term parser validating terminal condition
    GreedyParser( const Parser& p,
                  bool skipBlanks = true,
                  const Parser& term = Parser() )
        : parser_( p ), terminalParser_( term ), skipBlanks_( skipBlanks ) {}
    void SetParser( const Parser& p ) { parser_ = p; }
    /// Implementation of IParser::Parse. Returns @c true if and only if all
    /// the parsers in the sequnce return true; @c false otherwise.
    bool Parse( InStream& is ) {
        bool ok = !( terminalParser_.Valid() && terminalParser_.Parse( is ) )
                  && parser_.Parse( is );
        REWIND r( ok, is );
        while( !ok && !is.eof() ) {
            if( skipBlanks_ ) {
                InStream::char_type c = is.get();
                while( IsSpace( c ) && !is.eof() ) c = is.get();
                if( !is.eof() ) is.unget();
            }
            ok = !( terminalParser_.Valid() && terminalParser_.Parse( is ) )
                  && parser_.Parse( is );
        }
        return ok;
    }
    /// Implementation of IParser::GetValues.
    /// Returns all the values parsed by provided parsers. Lazy behavior: values
    /// are retrieved from parsers the first time this method is invoked.
    const Values& GetValues() const {
        return parser_.GetValues();
    }
    /// Implementation of IParser::operator[]. Invokes GetValues() first.
    /// @exception std::logic error if key not found.
    const ValueType& operator[]( const KeyType& k ) const {
        const Values& v = GetValues();
        Values::const_iterator i = v.find( k );
        if( i == v.end() ) throw std::logic_error( "Cannot find value" );
        return i->second;
    }

    /// Implementation of IParser::Clone.
    GreedyParser* Clone() const { return new GreedyParser( *this ); }

private:
    /// Parser list.
    Parser parser_;
    Parser terminalParser_;
    bool skipBlanks_;
};

} //namespace
```