The Prospective Boost.Lexer Library and its Integration with Boost.Spirit

Ben Hanson
ben@benhanson.net

Hartmut Kaiser
hartmut.kaiser@gmail.com
Center for Computation and Technology
Louisiana State University

Ben Hanson

Agenda

- The (proposed) Boost.Lexer library
  - Overview
  - Features
  - Structure
- The integration of Boost.Lexer and SpiritV2
  - Goals and Features
  - Examples

Boost.Lexer (proposed)

Some Definitions

- Token
  - Is a sequence of consecutive characters having a collective meaning
  - May have attributes specific to the token type, carrying additional information about the matched character sequence.
- Pattern
  - Is a rule for matching tokens expressed as a regular expression
  - Describes how a particular token can be formed
    - For example: `"[A-Za-z_][A-Za-z_0-9]*"` is a pattern for matching C++ identifiers
- Whitespace
  - Characters separating tokens
  - These include spaces, tabs, newlines, and form feeds.
  - Many people also count comments as whitespace
  - Depends on the language

What's Boost.Lexer

- Boost.Lexer is a lexical analyser generator inspired by flex.
- The aim is to support all the features of flex, whilst providing a more modern interface and supporting wide characters.
- Based on table based DFAs (deterministic finite automatons)
- The library aims for total flexibility by allowing the user to decide whether to use it dynamically or statically.
- Located in `namespace boost::lexer`

Boost.Lexer Features

- Build a set of lexing rules by the use of regular expressions
  - One regex for each of the tokens to be recognized
- Allows the use of regex 'macros' if required
- Allows multiple 'lex states' like flex for sophisticated lexing possibilities
  - Each lexer state encapsulates as set of related rules
  - Lexer states represented as an array of DFAs
  - DFA[0] is always the 'INITIAL' DFA
- Handles narrow (`char`) and wide character sets (`wchar_t`)

Boost.Lexer Features

- Build a (table based) state machine from the lexing rules, optionally minimizing the DFAs
- Process text using the state machine and supplied drivers (or write your own driver)
  - This creates tokens from the input stream
- Generate (C++) code from the state machine allowing the creation of static (table driven) lexical analyzers
- Save and load the `state_machine` using Boost.Serialization
- Create debug output in human readable format from generated DFAs

Supported Regex Syntax

Expression	Meaning
X	Match the character X
.	Any character
[0-9]	Digits 0 through 9
[^0-9]	Anything except 0-9
R*	Zero or more R's
R+	One or more R's
R?	Zero or one R
R{n,m}	R between n and m times
R{n,}	At least n R's
R{n}	Exactly n R's
{MACRO}	The named regex MACRO
"*+?"	The character sequence '*+?'
R|S	R or S
^R	R, but only at the beginning of a line
R$	R, but only at the end of a line

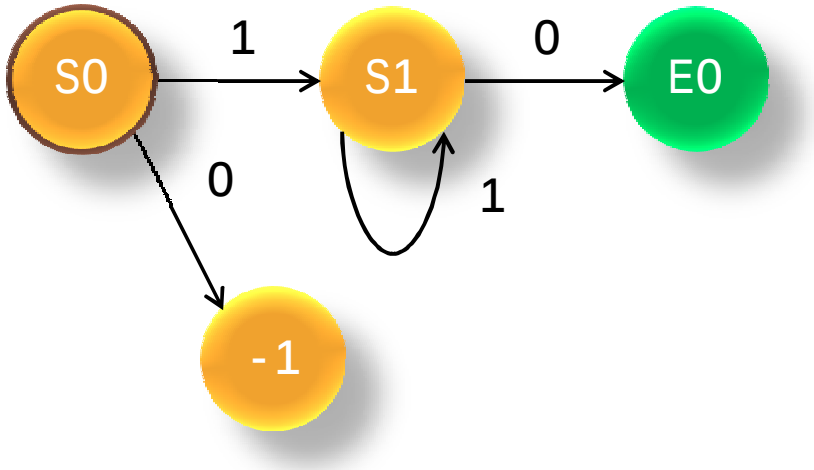
Escape sequences are also supported such as \0, \123, \x2a, \d, \D, \s, \S, \w and \W.

How Lexers use Regular Expressions

- Rules are defined in priority order, that is earlier rules over-ride later ones
- Essentially all rules are ‘or’ed together, with the id being added at the end of each expression to uniquely identify it (see the famous ‘Dragon Book’ for details)

Recap: What's a DFA

- Consider a regular expression on a two symbol alphabet ('0' and '1')
  - "1+0": arbitrary number of '1' followed by a single '0'



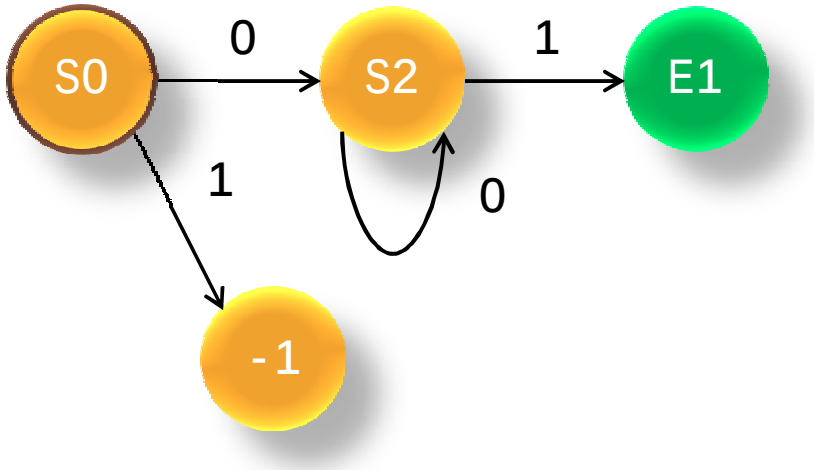
State Machine

E0 is equivalent to S0

DFA		Input	
		0	1
State	S0	-1	S1
	S1	S0	S1

Recap: What's a DFA

- Consider a second regular expression on the same alphabet
  - "0+1": arbitrary number of '0' followed by a single '1'



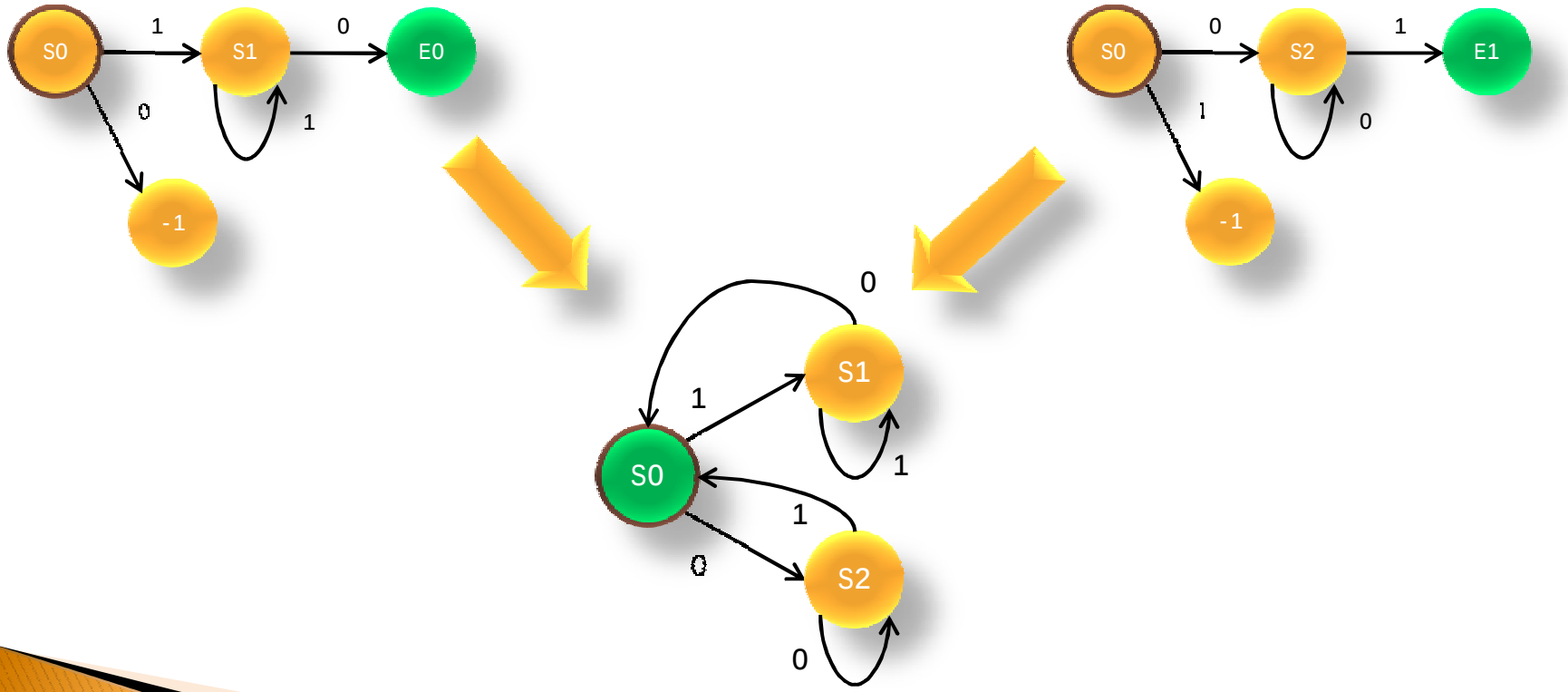
State Machine

E1 is equivalent to S0

DFA		Input	
		0	1
State	S0	S2	-1
	S2	S2	S0

Combined State Machine

- Combine the state machines for the different regular expressions into one

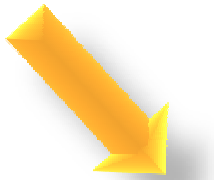


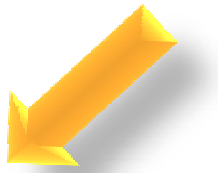
Merged and Minimized DFA

- Minimization combines the DFAs for the different regular expressions into one DFA

DFA 1		Input	
		0	1
State	S0	-1	S1
	S1	S0	S1

DFA 2		Input	
		0	1
State	S0	S2	-1
	S2	S2	S0





Minimized DFA		Input	
		0	1
State	S0	S2	S1
	S1	S0	S1
	S2	S2	S0

Building DFAs Efficiently (1)

- When building a state machine from a regex syntax tree, the text book approach is to consider every possible character one at a time and see which nodes can be reached from it.
- This approach is far too slow in practice – consider the character set '.' (every character) in wide character mode – we are talking about thousands of characters.
- Instead of dealing with each character one at a time, we deal with groups of characters instead.

Building DFAs Efficiently (2)

- We can therefore reduce the processing required per follow position to a list of character sets instead of a list of characters. For a large lex spec this reduces the processing required massively.
- To see the implementation look in `string_token.hpp` (see functions `normalise()` and `intersect()`) and `generator.hpp` (see `build_tree()`).

Boost.Lexer Structure

- There are 3 core classes:
  - `rules`
    - Helper class allowing the collection of all information needed to generate the DFAs
    - Add lexer rules (token definitions) using the `rules` class
  - `generator`
    - Helper class exposing functionality to generate and minimize, and generally manage DFAs
  - `state_machine`
    - Holds generated (and optimized) DFAs

Boost.Lexer Structure

- Additionally the library provides a couple of often used functions
  - Process text by the use of `state_machine` in conjunction with some lookup code.
  - Generate (C++) code from `state_machine` to be included in static lexical analyzer
  - Dump DFAs in human readable format (for debug purposes)
- Reserved ids:
  - 0 is reserved for end of input (like flex)
  - boost::lexer::npos (~0) is reserved for 'unknown token' (no match found)

Example 1: Build a state_machine

```
#include <iostream>
#include <boost/lexer/generator.hpp>
#include <boost/lexer/rules.hpp>
#include <boost/lexer/state_machine.hpp>

enum token_ids { whitespace = 1, identifier };

int main(int argc, char* argv[])
{
    try {
        boost::lexer::rules rules_;
        boost::lexer::state_machine state_machine_;

        rules_.add ("\\s+", whitespace);
        rules_.add ("[a-z]+", identifier);

        boost::lexer::generator::build (rules_, state_machine_);  // Build DFA from rules
    }
    catch (boost::lexer::runtime_error &e) {
        std::cout << e.what ();
    }
    return 0;
}
```

Example 2: Dump a state_machine

```
#include <iostream>
#include <boost/lexer/debug.hpp>
#include <boost/lexer/generator.hpp>
#include <boost/lexer/rules.hpp>
#include <boost/lexer/state_machine.hpp>

enum token_ids { whitespace = 1, identifier };

int main(int argc, char* argv[])
{
    try {
        boost::lexer::rules rules_;
        boost::lexer::state_machine state_machine_;

        rules_.add ("\\s+", whitespace);
        rules_.add ("[A-Za-z_][A-Za-z_0-9]*" , identifier);

        boost::lexer::generator::build (rules_, state_machine_); // Build DFA from rules
        boost::lexer::generator::minimise (state_machine_);      // Minimize the state machine
        boost::lexer::debug::dump (state_machine_, std::cout);   // Dump the state machine
    }
    catch (boost::lexer::runtime_error &e) {
        std::cout << e.what ();
    }
    return 0;
}
```

Example 3: Generating C++

```
#include <iostream>
#include <boost/lexer/generate_cpp.hpp>
#include <boost/lexer/generator.hpp>
#include <boost/lexer/state_machine.hpp>

enum token_ids { whitespace = 1, identifier };

int main(int argc, char* argv[])
{
    try {
        boost::lexer::rules rules_;
        boost::lexer::state_machine state_machine_;

        rules_.add ("\\s+", whitespace);
        rules_.add ("[a-z]+", identifier);

        boost::lexer::generator::build (rules_, state_machine_); // Build DFA from rules
        boost::lexer::generator::minimise (state_machine_);      // Minimize the state machine
        boost::lexer::generate_cpp (state_machine_, std::cout);  // Generate C++
    }
    catch (boost::lexer::runtime_error &e) {
        std::cout << e.what ();
    }
    return 0;
}
```

Example 4: Instant Lexing

```
#include <iostream>
#include <boost/lexer/generator.hpp>
#include <boost/lexer/state_machine.hpp>
#include <boost/lexer/tokeniser.hpp>

void tokenize (boost::lexer::state_machine const& state_machine_)
{
    std::string input_ ("!This is a test");
    std::string::iterator start_ = input_.begin();
    std::string::iterator start_token_ = start_;
    std::string::iterator curr_ = start_;
    std::string::iterator end_ = input_.end ();
    std::size_t id_ = boost::lexer::npos;
    std::size_t state_ = 0;

    while (id_ != 0) {
        id_ = boost::lexer::iter_tokeniser::next (state_machine_,
            state_, start_, curr_, end_);

        std::string token_ (start_token_, curr_);

        std::cout << "Id: " << id_ << ", Token: '" << token_ << "'\n";
        start_token_ = curr_;
    }
}
```

Example 4: Instant Lexing

```
enum token_ids { whitespace = 1, identifier };

int main(int argc, char* argv[])
{
    try {
        boost::lexer::rules rules_;
        boost::lexer::state_machine state_machine_;

        rules_.add ("\\s+", whitespace);
        rules_.add ("[a-z]+", identifier);

        boost::lexer::generator::build (rules_, state_machine_);

        tokenize (state_machine_);              // Tokenize using the state machine
    }
    catch (boost::lexer::runtime_error &e)
    {
        std::cout << e.what ();
    }
    return 0;
}
```

Example 5: Multi-state Lexing

```
#include <boost/lexer/generator.hpp>
#include <iostream>
#include <map>
#include <boost/lexer/state_machine.hpp>
#include <string>
#include <boost/lexer/tokeniser.hpp>

// Easier to enumerate tokens in production
   code...
enum e_URLToken {
    eEOF = 0,
    eFile,
    eQuestion,
    eParam,
    eEquals,
    eValue,
    eSeparator
};
```

```
// Data structure to parse into
struct url_t
{
    typedef std::map<std::string,
        std::string> string_string_map;
    typedef std::pair<std::string,
        std::string> string_string_pair;

    std::string m_file;
    size_t m_index;
    string_string_map m_value_map;

    void clear ()
    {
        m_file.erase ();
        m_index = std::string::npos;
        m_value_map.clear ();
    }
};
```

Example 5: Multi-state Lexing

```
std::string str ("test.html?i=0&p1=test");

// Case insensitive rules
boost::lexer::rules rules (false);
boost::lexer::state_machine sm;
url_t url;

// Start states
rules.add_state ("QUESTION");
rules.add_state ("PARAM");
rules.add_state ("EQUALS");
rules.add_state ("VALUE");
rules.add_state ("SEPARATOR");

// Context sensitive rules
// It may not be EBNF, but it's quick and easy...
rules.add ("INITIAL", "[^?]+", eFile, "QUESTION");
rules.add ("QUESTION", "\\?", eQuestion, "PARAM");
rules.add ("PARAM", "i|p\\d+", eParam, "EQUALS");
rules.add ("EQUALS", "=", eEquals, "VALUE");
rules.add ("VALUE", "[^&]+", eValue, "SEPARATOR");
rules.add ("SEPARATOR", "&", eSeparator, "PARAM");
boost::lexer::generator::build (rules, sm);
```

```
const char* pstart = str.c_str();
const char* pcurr = pstart;
const char* pend = pstart + str.size();
std::size_t state = 0;
std::size_t id = eFile;
std::string param;
std::string value;

while (id != eEOF) {
    pszStart = pcurr;
    id = ptr_tokeniser::next (sm, state,
                pcurr, pend));

    switch (id) {
    case boost::lexer::npos:
        throw runtime_error("…");
    case eFile:
        url.m_file.assign(pstart, pcurr);
        break;
    case eParam:
        param.assign (pstart, pcurr);
        break;
    case eValue:
        value.assign (pstart, pcurr);
        if (param == "i" || param == 'I')
            url.m_index = atoi(value.c_str());
        else
            url.m_value_map.insert(param, value);
        break;
    }
}
```

Performance Comparison

- Small Grammar:

```
rules_.add("[a-z]+", 1);
rules_.add("[0-9]+", 2);
rules_.add("[ \t]+", 3);
rules_.add(".", 100);
```

- Compared to flex with small grammar (looping 1000 times):
  - Boost.Lexer: 0.72 seconds
  - flex: 1.19 seconds
- Compared to flex with full set of C++ token definitions (looping 1000 times):
  - Boost.Lexer: 1.71 seconds
  - flex: 2.55 seconds

Platforms Tested On

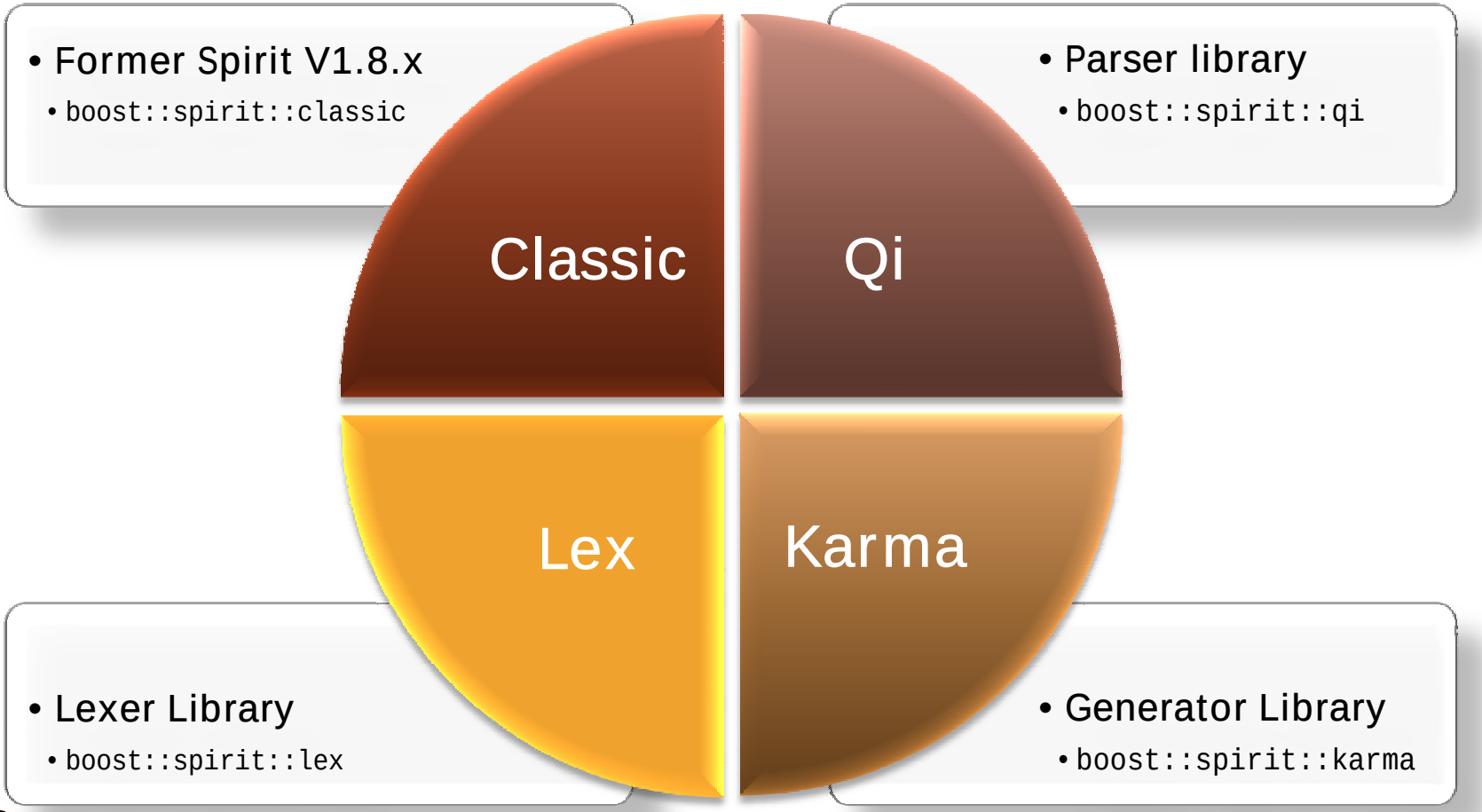
- (Tested via Spirit Regression Tests)
  - Darwin, Sandia-darwin-intel
  - Darwin, Sandia-darwin-ppc
  - HP-UX, HP-UX ia64 aCC
  - Windows, RW WinXP, VC7.1, VC8, VC9

Future Work

- Modernise the interface (iterators) in preparation for the Boost review. Having the state machine completely exposed is not going to fly!
- Introduce compression for wchar_t (table) mode.
- Make std::size_t a template parameter for table entries.
- Support more flex features (the REJECT command for example).

Spirit.Lex

Spirit's Library Structure



What's Spirit.Lex

- Lexical scanning is
  - The process of analyzing the input stream
  - Separating it into strings called tokens
  - Often separated by whitespace
- The Component doing the lexical scanning
  - Is called Lexer, Scanner, Lexical analyzer
- Spirit.Lex is a library
  - To take care of the complexities of creating a lexer
  - Takes a set of regular expressions describing the tokens
  - Generates a DFA (Deterministic Finite Automaton)
  - Integrates well with parsers built using Spirit.Qi
  - Usable as standalone lexer (without parsing)

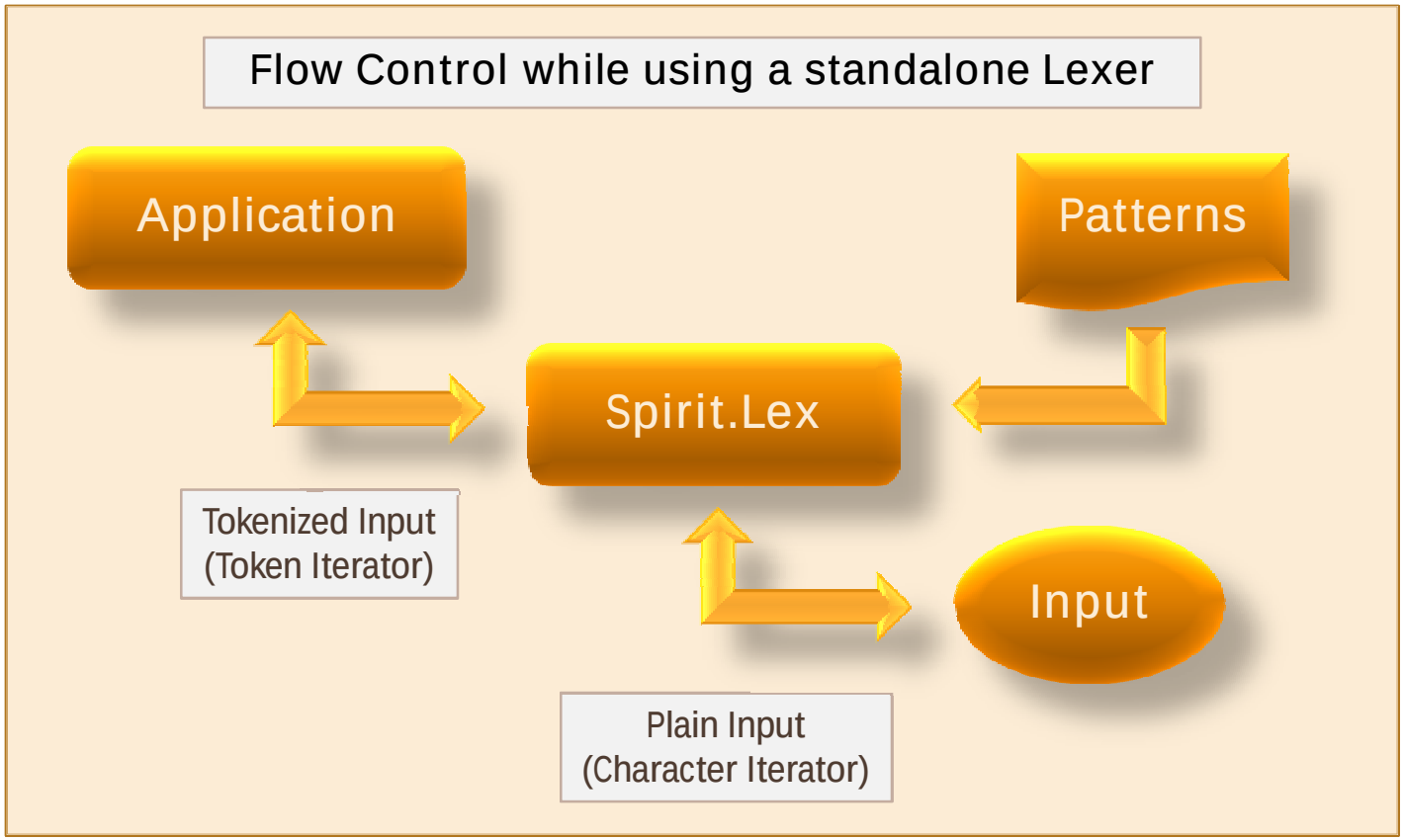
What's Spirit.Lex

- Provides facilities to
  - Define patterns for tokens to match
  - Assign token id's to token definitions
  - Associate patterns with lexer states
  - Execute code after the token has been matched
- Exposes a pair of iterators
  - Return a stream of tokens generated from the underlying character stream
- Provides unified interface for potentially different lexer libraries
  - Boost.lexer (proposed) used as proof of concept and first implementation

Library Features

- Select and customize the token type to be generated by the lexer instance
- Select and customize the token value types the generated token instances will be able to hold
- Select the iterator type of the underlying input stream, which will be used as the source for the character stream to tokenize
- Customize the iterator type returned by the lexer to enable debug support, special handling of certain input sequences, etc.
- Select the dynamic or the static runtime model for the lexical analyzer.

Library Structure



The Spirit.Lexer Components

- Facilities allowing to the definition of tokens based on regular expression strings
  - Classes: `token_def, token_set, lexer`

```
token_def<> identifier = "[a-zA-Z_][a-zA-Z0-9_]*";
self = identifier | ',' | '{' | '}';
```

- Exposes an iterator based interface for easy integration with Spirit parsers:

```
std::string str (…input…);
lexer<example1_tokens> lex(tokens);
grammar<example1_grammar> calc(def);
parse(lex.begin(str.begin(), str.end()), lex.end(), calc);
```

- Lexer related components are at the same time parser components, allowing for tight integration

```
start = '{' >> *(identifier >> -char_(',')) >> '}';
```

The Spirit.Lexer Components

- Advantages:
  - Avoid re-scanning of input stream during backtracking
    - Exposed iterator is based on the `multi_pass<>` iterator specifically developed for Spirit parsers (stores the last recognized tokens)
  - Simpler grammars for input language
  - Token values are evaluated once
  - Pattern (token) recognition is fast (uses regex's and DFAs)
- Disadvantages:
  - Additional overhead for token construction (especially if no backtracking occurs or no special token value types are used)

The Spirit.Lex Components

- Parsing using a lex is fully token based (even single characters are tokens)
- Every token may have its own (typed) value

```
token_def<std::string> identifier = "[a-zA-Z_][a-zA-Z0-9_]*";
```

- During parsing this value is available as the tokens (parser components) 'return value' (parser attribute)

```
std::vector<std::string> names;
start = '{'
  >> *(identifier >> -char_(',')) [ref(names)]
  >> '}';
```

- Token values are evaluated once and only on demand **à** no performance loss
- Tokens without a value 'return' iterator pair

Examples

Counting Words: Flex

```
// token definition
%{
  #define ID_WORD 1000
  #define ID_EOL  1001
  #define ID_CHAR 1002
  int c = 0, w = 0, l = 0;
%}
%%
[^ \t\n]+ { return ID_WORD; }
\n        { return ID_EOL; }
.         { return ID_CHAR; }
%%
```

```
bool count(int tok)
{
  switch (tok) {
  case ID_WORD:
    ++w; c += yyleng; break;
  case ID_EOL: ++l; ++c; break;
  case ID_CHAR: ++c; break;
  default: return false;
  }
  return true;
}

// main routine
void main()
{
  int tok = EOF;
  do {
    tok = yylex();
    if (!count(tok)) break;
  } while (EOF != tok);
  printf("%d %d %d\n", l, w, c);
}
```

Counting Words: Spirit.Lex

- Lexers created with Spirit.Lex are usable standalone
  - These expose a pair of forward iterators, which when dereferenced, return the next matched token
- Simplest case
  - All that needs to be done is to write a single line of code for each of the tokens to match
  - It's possible to register one function being called for all matched tokens

Counting Words: Spirit.Lex

```
int c = 0, w = 0, l = 0;


// token definition
struct word_count_tokens
  : lex::lexer_def<lex::lexer_lexer<> >
{
  template <typename Self>
  void def (Self& self)
  {
    self.add
      ("[^ \t\n]+", ID_WORD)
      ("\n", ID_EOL)
      (".", ID_CHAR)
    ;
  }
};


// lexer definition
word_count_tokens word_count;
```

```
struct counter {
  template <typename Token>
  bool operator()(Token const& t) {
    switch (t.id()) {
    case ID_WORD:
      ++w; c += t.value().size();
      break;
    case ID_EOL: ++l; ++c; break;
    case ID_CHAR: ++c; break;
    default: return false;
    }
    return true;
  }
};

// main routine
std::string str("…");
char const* p = str.c_str();
bool r =
    lex::tokenize(
        p, &p[str.size()],
        lex::make_lexer(word_count),
        bind(counter(), _1));
```

Counting Words: Spirit.Lex

```
int c = 0, w = 0, l = 0;


// token definition
struct word_count_tokens
  : lex::lexer_def<lex::lexer_lexer<> >
{
  template <typename Self>
  void def (Self& self)
  {
    self.add
      ("[^ \t\n]+", ID_WORD)
      ("\n", ID_EOL)
      (".", ID_CHAR)
    ;
  }
};


// lexer definition
word_count_tokens word_count;
```

```
struct counter {
  template <typename Token>
  bool operator()(Token const& t) {
    switch (t.id()) {
    case ID_WORD:
      ++w; c += t.value().size();
      break;
    case ID_EOL: ++l; ++c; break;
    case ID_CHAR: ++c; break;
    default: return false;
    }
    return true;
  }
};

// main routine
std::string str("…");
char const* p = str.c_str();
bool r =
    lex::tokenize(
        p, &p[str.size()],
        lex::make_lexer(word_count),
        bind(counter(), _1));
```

Lexer API

- Standalone tokenization

```
template <typename Iterator, typename Lexer>
bool tokenize(Iterator& first, Iterator last,
    Lexer const& lex);

template <typename Iterator, typename Lexer, typename F>
bool tokenize(Iterator& first, Iterator last,
    Lexer const& lex, F f);
```

Lexer API

- Tokenization and parsing

```
template <typename Iterator, typename Lexer, typename Parser>
bool tokenize_and_parse(Iterator& first, Iterator last,
    Lexer const& lex, Parser const& xpr);

template <typename Iterator, typename Lexer,
    typename Parser, typename Attribute>
bool tokenize_and_parse(Iterator& first, Iterator last,
    Lexer const& lex, Parser const& xpr, Attribute& attr);
```

Lexer API

- Tokenization and phrase parsing

```
template <typename Iterator, typename Lexer,
    typename Parser, typename Skipper>
bool tokenize_and_phrase_parse (Iterator& first,
    Iterator last, Lexer const& lex, Parser const& xpr,
    Skipper const& skip);

template <typename Iterator, typename Lexer,
    typename Parser, typename Attribute, typename Skipper>
bool tokenize_and_phrase_parse(Iterator& first,
    Iterator last, Lexer const& lex, Parser const& xpr,
    Attribute& attr, Skipper const& skip);
```

Counting Words: Version 2

- It's possible to associate code with each of the matched tokens (semantic actions)
- Information carried by tokens
  - Token id
  - Token value (always `iterator_range<>`)
  - Lexer state (optional)

Counting Words: Version 2

```
%{
  int c = 0, w = 0, l = 0;
%}
%%
[^ \t\n]+ { ++w; c += yyleng; }
\n        { ++c; ++l; }
.         { ++c; }
%%
int main()
{
  yylex();
  printf("%d %d %d\n", l, w, c);
  return 0;
}
```

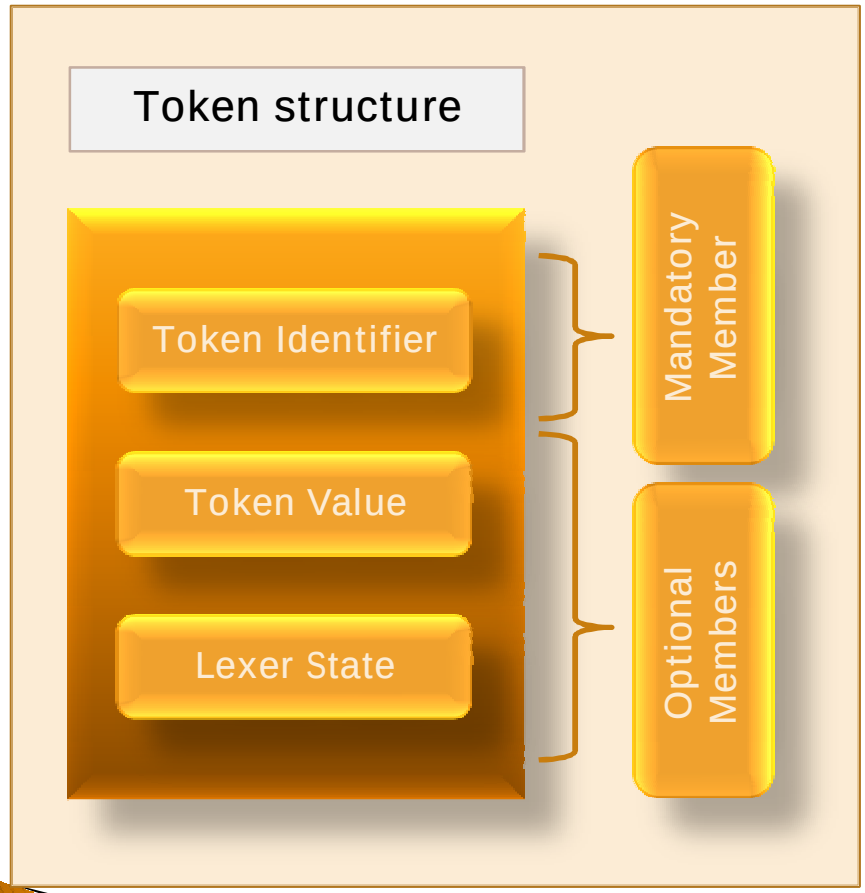
```
template <typename Lexer>
struct word_count_tokens
  : lex::lexer_def<Lexer>
{
  word_count_tokens()
   : c(0), w(0), l(0),
     word("[^ \t\n]+"),
     eol("\n"), any(".")
  {}

  template <typename Self>
  void def (Self& self)
  {
    self =  word [
              ++ref(w),
              ref(c) += distance(_1)
            ]
          |  eol [++ref(c), ++ref(l)]
          |  any [++ref(c)]
          ;
  }
  std::size_t c, w, l;
  lex::token_def<> word, eol, any;
};
```

Counting Words: Version 2

```
%{
  int c = 0, w = 0, l = 0;
%}
%%
[^ \t\n]+ { ++w; c += yyleng; }
\n        { ++c; ++l; }
.         { ++c; }
%%
int main()
{
  yylex();
  printf("%d %d %d\n", l, w, c);
  return 0;
}
```

```
template <typename Lexer>
struct word_count_tokens
  : lex::lexer_def<Lexer>
{
  word_count_tokens()
    : c(0), w(0), l(0),
      word("[^ \t\n]+"),
      eol("\n"), any(".")
  {}

  template <typename Self>
  void def (Self& self)
  {
    self =  word [
              ++ref(w),
              ref(c) += distance(_1)
            ]
          |  eol [++ref(c), ++ref(l)]
          |  any [++ref(c)]
          ;
  }
  std::size_t c, w, l;
  lex::token_def<> word, eol, any;
};
```

Token Structure




```
template <
    typename Iterator,
    typename AttributeTypes,
    typename HasState>
struct lexer_token;
```

- Iterator
  - Underlying iterator type
- AttributeTypes
  - List of possible token *value* types
  - At least `iterator_range<>`
  - `omitted`: no token value
- HasState
  - Stores lexer state

Ways to define a Token

- Several ways to define a token

```
self.add("[a-z]")("[0-9]", ID_DIGIT);
self = token_def<>("[ \t\n]");
```

  - Syntax sugar: '|'

```
self = token_def<>("[ \t\n]") | '\b' | "a+";

token_def<> ident("[a-zA-Z_]+");
self = token_def<>("[ \t\n]", ID_WHITESPACE) | ident;
```

  - Define new lexer states:

```
self("STATE") = …;
```

Semantic Actions

- Construct allowing the attachment of code to a token definition
  - Gets executed *after* a token as described by the token definition has been matched, but *before* it gets returned from the lexer
  - Allows to access the matched data
  - Allows a rule to force a lex failure
- Syntax similar to parser semantic actions

```
token_def<std::string> id("[a-zA-Z_][a-zA-Z0-9_]*");
boost::iterator_range<char const*> r;
id[ref(r) = _1]     // _1 is not std::string!
```

Semantic Actions

- Any function or function object can be called

```
void f(Attribute&, Idtype const&, bool&, Context&);
void f(Attribute&, Idtype const&, bool&);
void f(Attribute&, Idtype const&);
void f(Attribute&);
void f();
```

- Attribute
  - On this level no parser attributes (token values) are available yet, only the matched sequence: `iterator_range<base_iterator>`
  - The token instance has not been constructed yet
- Idtype
  - token identifier (`std::size_t`)
- bool
  - Allows you to make the match fail, if needed (by assigning false)
- Context
  - Internal data of token type
  - For the more adventurous, depends on token type i.e. change of lexer state or other token/lexer specific operations

Semantic Actions

- Best way is to use phoenix function objects

```
token_def<> int_ = "[0-9]+";
using boost::spirit::arg_names::_1;
self = int_[std::cout << _1, pass = val(false)];
```

- Plain function:

```
void on_int(iterator_range<char const*> r)
{ std::cout << r; }
self = int_[&on_int];
```

- But it's possible to use `boost::lambda`…

```
using boost::lambda::_1;
self = int_[std::cout << _1];
```

- … and `boost::bind` as well

```
self = int_[boost::bind(&on_int, _1)]
```

Semantic Actions

- Best way is to use phoenix function objects

```
token_def<> int_ = "[0-9]+";
self = int_[std::cout << _1, pass = val(false)];
```

`_1`:
    Matched input sequence (`iterator_range<base_iter>`)

`id`:
    Token id of matched token (`std::size_t`)

`pass`:
    Make the match fail in retrospective

`state`:
    Lexer state this token has been matched in

Static Lexing

- Default mode of operation is to generate the DFAs at runtime
  - Advantages
    - flexible
    - easy turnaround
    - no additional build step
  - Disadvantage
    - Additional runtime requirement
    - Might be a performance issue
- Spirit.Lexer supports building static lexers
  - Generate DFAs in separate step
  - Use the generated DFAs while compiling the code using the lexer

Static Lexing

```
struct word_count_tokens
  : lex::lexer_def<lexer_lexer<> >
{
  word_count_tokens()
   : c(0), w(0), l(0),
     word("[^ \t\n]+"),
     eol("\n"), any(".")
  {}

  template <typename Self>
  void def (Self& self)
  {
    self =  word [
              ++ref(w),
              ref(c) += distance(_1)
            ]
          |  eol [++ref(c), ++ref(l)]
          |  any [++ref(c)]
          ;
  }
  std::size_t c, w, l;
  lex::token_def<> word, eol, any;
};
```

```
// generate static lexer DFAs
int main(int argc, char* argv[])
{
    // create the lexer object instance
    // needed to invoke the generator

    // the token definition
    word_count_tokens word_count;

    // open the output file, where the
    // generated tokenizer function will
    // be written to
    std::ofstream out(argv[1]);

    // invoke the generator, passing the
    // token definition, the output
    // stream and the name prefix of the
    // tokenizing function to be
    // generated
    char const* function_name = argv[2];

    return generate_static(
        make_lexer(word_count), out,
        function_name) ? 0 : -1;
}
```

Counting Words: Static Lexing

```
struct word_count_tokens
  : lex::lexer_def<lexer_lexer<> >
{
  word_count_tokens()
   : c(0), w(0), l(0),
     word("[^ \t\n]+"),
     eol("\n"), any(".")
  {}

  template <typename Self>
  void def (Self& self)
  {
    self =  word [
              ++ref(w),
              ref(c) += distance(_1)
            ]
          |  eol [++ref(c), ++ref(l)]
          |  any [++ref(c)]
          ;
  }
  std::size_t c, w, l;
  lex::token_def<> word, eol, any;
};
```

```
struct word_count_tokens
  : lex::lexer_def<lexer_static_lexer<> >
{
  word_count_tokens()
   : c(0), w(0), l(0),
     word("[^ \t\n]+"),
     eol("\n"), any(".")
  {}

  template <typename Self>
  void def (Self& self)
  {
    self =  word [
              ++ref(w),
              ref(c) += distance(_1)
            ]
          |  eol [++ref(c), ++ref(l)]
          |  any [++ref(c)]
          ;
  }
  std::size_t c, w, l;
  lex::token_def<> word, eol, any;
};
```

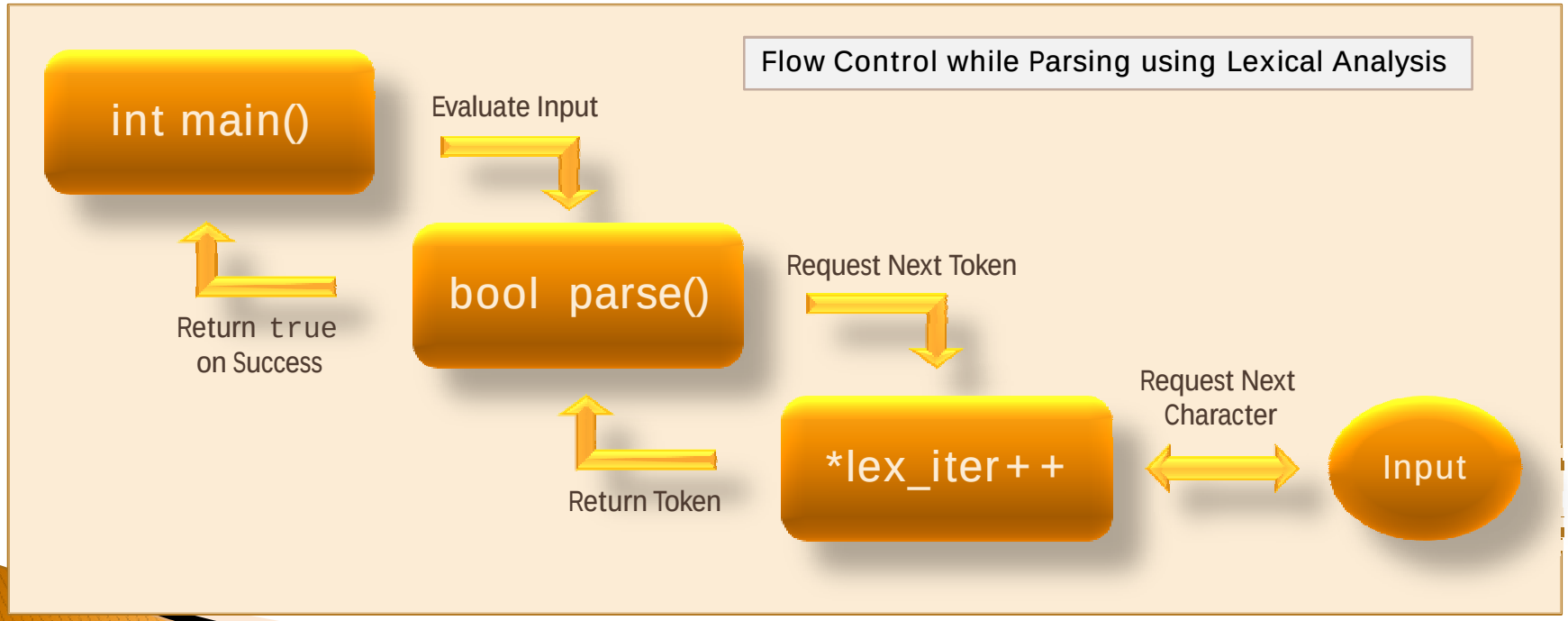
Counting Words: Static Lexing

```
struct word_count_tokens
  : lex::lexer_def<lexer_lexer<> >
{
  word_count_tokens()
   : c(0), w(0), l(0),
     word("[^ \t\n]+"),
     eol("\n"), any(".")
  {}

  template <typename Self>
  void def (Self& self)
  {
    self =  word [
              ++ref(w),
              ref(c) += distance(_1)
            ]
          |  eol [++ref(c), ++ref(l)]
          |  any [++ref(c)]
          ;
  }
  std::size_t c, w, l;
  lex::token_def<> word, eol, any;
};
```

```
struct word_count_tokens
  : lex::lexer_def<lexer_static_lexer<> >
{
  word_count_tokens()
   : c(0), w(0), l(0),
     word("[^ \t\n]+"),
     eol("\n"), any(".")
  {}

  template <typename Self>
  void def (Self& self)
  {
    self =  word [
              ++ref(w),
              ref(c) += distance(_1)
            ]
          |  eol [++ref(c), ++ref(l)]
          |  any [++ref(c)]
          ;
  }
  std::size_t c, w, l;
  lex::token_def<> word, eol, any;
};
```

Parsing using Spirit.Lex

- Spirit.Lex provides iterators directly consumable by Spirit.Qi parsers



Counting Words: Using a Parser

- Possibilities when defining tokens
  - Characters stand for themselves
    - Have to be used as `char_('\n')` or directly as `'\n'`
    - Parser attribute is character value
  - Token definition (`token_def<>`) instances are valid parser components
    - Can be used directly without wrapping
    - Parser attribute is `iterator_range<>`, explicitly specified, or none at all (omitted)
  - Token id's are directly usable as valid parser components by wrapping them in a `token(IDANY)`
    - Parser attribute is `iterator_range<>`

Counting Words: Using a Parser

```
template <typename Lexer>
struct word_count_tokens
  : lex::lexer_def<Lexer>
{
  template <typename Self>
  void def (Self& self)
  {
    self.add_pattern
      ("WORD", "[^ \t\n]+")
    ;

    word = "{WORD}";

    self.add
      (word)
      ('\n')
      (".", IDANY)
    ;
  }
  lex::token_def<string> word;
};
```

```
template <typename Iterator>
struct word_count_grammar
  : qi::grammar_def<Iterator>
{
  template <typename TokenDef>
  word_count_grammar(TokenDef const& t)
    : c(0), w(0), l(0)
  {
    start = *(
            t.word [
              ++ref(w),
              ref(c) += size(_1)
            ]
          | char_('\n') [
              ++ref(c), ++ref(l)
            ]
          | token(IDANY) [ ++ref(c) ]
        );
  }
  std::size_t c, w, l;
  qi::rule<Iterator> start;
};
```

Counting Words: Using a Parser

```
template <typename Lexer>
struct word_count_tokens
  : lex::lexer_def<Lexer>
{
  template <typename Self>
  void def (Self& self)
  {
    self.add_pattern
      ("WORD", "[^ \t\n]+")
    ;

    word = "{WORD}";

    self.add
      (word)
      ('\n')
      (".", IDANY)
    ;
  }
  lex::token_def<string> word;
};
```

```
template <typename Iterator>
struct word_count_grammar
  : qi::grammar_def<Iterator>
{
  template <typename TokenDef>
  word_count_grammar(TokenDef const& t)
    : c(0), w(0), l(0)
  {
    start = *(
            t.word [
              ++ref(w),
              ref(c) += size(_1)
            ]
          | char_('\n') [
              ++ref(c), ++ref(l)
            ]
          | token(IDANY) [ ++ref(c) ]
        );
  }
  std::size_t c, w, l;
  qi::rule<Iterator> start;
};
```

Lexer States

- More complex use cases might be simplified by using lexer states
  - Makes lexers context sensitive
- Token definitions are associated with a lexer state, which implicitly introduces a new state

```
self("WS") = tokend_def<>("[ \t\n]*");
```

- Default lexer state is "INITIAL", so the following two are equivalent:

```
self = tokend_def<>(…);
self("INITIAL") = tokend_def<>(…);
```

- Start state is "INITIAL" as well (as long as not changed)

Lexer States

- Lexer states have to be switched explicitly
  - From a lexer semantic action:

```
self = token_def<>(…)[state = val("WS")];
```

  - Using special parser directives:

```
… >> set_state("WS") >> …
… >> in_state("WS")[…some parser…] >> …
```

Parsing using a Skipper

- Skipper is used to ‘remove’ certain parts of the input before it get’s passed to the parser
  - Whitespace, comments, delimiters, etc.
- Same thing is possible while parsing tokens
  - But now we’ll have to use a token based skipper
  - Need a way to distinguish ‘good’ and ‘bad’ tokens
- Simplest way is to use lexer states, one dedicated state for skipping

Skipper Example: Token definition

- Separate class allows for encapsulated token definitions:

```
// template parameter 'Lexer' specifies the underlying lexer (library) to use
template <typename Lexer>
struct example1_tokens : lex::lexer_def<Lexer>
{
    // the 'def()' function gets passed a reference to a lexer interface object
    template <typename Self>
    void def (Self& self)
    {
        // define tokens and associate them with the lexer
        identifier = "[a-zA-Z_][a-zA-Z0-9_]*";
        self = lex::token_def<>(',') | '{' | '}' | identifier;

        // any token definition to be used as the skip parser during parsing
        // has to be associated with a separate lexer state (here 'WS')
        white_space = "[ \\t\\n]+";
        self("WS") = white_space;
    }

    // every 'named' token has to be defined explicitly
    token_def<> identifier, white_space;
};
```

Skipper Example: Token definition

- Separate class allows for encapsulated token definitions:

```
// template parameter 'Lexer' specifies the underlying lexer (library) to use
template <typename Lexer>
struct example1_tokens : lex::lexer_def<Lexer>
{
    // the 'def()' function gets passed a reference to a lexer interface object
    template <typename Self>
    void def (Self& self)
    {
        // define tokens and associate them with the lexer
        identifier = "[a-zA-Z_][a-zA-Z0-9_]*";
        self = lex::token_def<>(',') | '{' | '}' | identifier;

        // any token definition to be used as the skip parser during parsing
        // has to be associated with a separate lexer state (here 'WS')
        white_space = "[ \\t\\n]+";
        self("WS") = white_space;
    }

    // every 'named' token has to be defined explicitly
    token_def<> identifier, white_space;
};
```

Skipper Example: Grammar definition

- Grammar definition takes token definition as parameter:

```
// template parameter 'Iterator' specifies the iterator this grammar is based on
// Note: token_def<> is used as the skip parser type
template <typename Iterator>
struct example1_grammar
  : qi::grammar_def<Iterator, qi::in_state_skipper<lex::token_def<> > >
{
    // parameter 'TokenDef' is a reference to the token definition class
    template <typename TokenDef>
    example1_grammar(TokenDef const& tok)
    {
        // Note: we use the 'identifier' token directly as a parser component
        start = '{' >> *(tok.identifier >> -char_(',')) >> '}';
    }

    // usual rule declarations, token_def<> is skip parser (as for grammar)
    rule<Iterator, qi::in_state_skipper<lex::token_def<> > > start;
};
```

Skipper Example: Parsing

- Parser invocation

```
// the token and grammar definitions as shown before
example1_tokens tokens;
example1_grammar def (tokens);

// create the lexer
lexer<example1_tokens> lex(tokens);

// call parse() as usual, supplying a special skip-parser
std::string str (…input…);
phrase_parse(
    lex.begin(str.begin(), str.end()), lex.end(),
    qi::make_grammar(calc), in_state("WS")[tokens.white_space]);
```

Future work

- Boost.Lexer
  - Rethink API, improve documentation
  - Pass Boost review
- Spirit.Lex
  - Debug support for lexers
  - Other lexer backends