1. License.

Gened date: January 30, 2015

2. Comments.
There are 3 principle reasons for this program:

1) To verify that O_2 parses a lr(1) grammar, and its parse tables accepts the language expessed
2) To give a glimpse into what the compiler writer "will do" in creating a translator/compiler
3) An example for your own learning

This "will do" activity prepares the Terminal vocabulary, and shows a simple way to fetch the input file to parse, and use the built in "error reporting" facilities within your own compiler/translator context.

The development cycle one needs to create a compiler/translator is:

1) write/compile your grammar(s) using O_2
2) using an editor, create your compiler's fsc file for O_2linker's consumption
3) write/compile your compiler and any other external functions
4) link your compiler/translator

Have a look at **/usr/local/yacco2/grammar-testsuite/makefile_testout_APPLE** bash script detailing the documentation generation and compiling/linking activities. This example is in "literate programming" genre and u can read the "testout*.w" files. If u prefer or u do not have *Cweb* installed, u can write raw C++code instead. The gened C++ files *.h and *.cpp are also in **../grammar-testsuite** to model from.

The grammar used is from David Pager — "The Lane Table Method Of Constructing LR(1) Parsers" discussing lr(1) resolution page 61. To testout pager_1.lex grammar, u can input one of the following files to *testout*:

1) ./grammar-testsuite/testout_1.dat
2) ./grammar-testsuite/testout_2.dat
3) x — no file
4) ./grammar-testsuite/testout_error.dat

An example of running the program:

cd /usr/local/yacco2/yacco2/grammar-testsuite
./testout testout_1.dat

Various Yac_2o_2 tracings are turned on to demo their activity. Please read wlibrary's pdf document for more details on their capabilities and meaning or "o2book.pdf" reference manual. Added to the tracing are file/line number source coordinates. At least u can go back to Yac_2o_2's library code when things need adjusting. "**1lrtracings.log**" contains the dynamic tracings when running the program.

General routines to get things going:

1) get file-to-parse and put into holding file
2) parse the command line containing the file to parse
3) use pager_1 grammar to parse input file's contents

⟨ accrue testout code 2 ⟩ ≡
  ⟨ Include files 3 ⟩;

See also sections 5, 7, 8, and 9.

This code is used in section 14.

3. Include files.
To start things off, these are the Standard Template Library (STL) containers needed by Yacco2's parse library definitions.

⟨ Include files 3 ⟩ ≡
```
#include "testout.h"
```
This code is used in section 2.

4. Create header file for testout environment.
The include files format is:

1) system related definitions
2) the grammar vocabulary framework
3) grammars used
4) namespace use for convenience in your C++ code

Note the namespace turn-ons for Error and Meta-terminal vocabularies. Your own language will define your vocabularies with their own namespaces. Have a look at their definition files: *testout_err_symbols.T* and *testout_terminals.T* and possibly re-read "o2book.pdf" for a refresher on their Ws. This example is skeletal for teaching purposes whereas your own vocabulary will be meater. Also note the 2 grammar headers: *pager_1.h* and the error handler *testout_err_hdlr.h*.

⟨ testout.h 4 ⟩ ≡
```
  ⟨ Preprocessor definitions ⟩
#ifndef testout__
#define testout__ 1
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "yacco2.h"
#include "testout_T_enumeration.h"
#include "yacco2_characters.h"
#include "yacco2_k_symbols.h"
#include "testout_terminals.h"
#include "testout_err_symbols.h"
#include "pager_1.h"
#include "testout_err_hdlr.h"
  using namespace std;
  using namespace NS_testout_T_enum;
  using namespace NS_yacco2_k_symbols;
  using namespace NS_testout_terminals;
  using namespace NS_testout_err_symbols;
  using namespace yacco2;
#endif
```

5. Define variables.

1) Holding file to start things off — command line data is put into this file
2) Define the O_2's tracing variable by macro
3) O_2's token containers needed

```
⟨ accrue testout code 2 ⟩ +≡
#define Max_buf_size 2 * 1024
  std::string Holding_file("testout_holding_file.cmd");
  std::string To_parse_file;
  YACCO2_define_trace_variables( );
  yacco2::TOKEN_GAGGLE JUNK_tokens;
  yacco2::TOKEN_GAGGLE IP_tokens;
  yacco2::TOKEN_GAGGLE Error_queue;
```

6. Local Routines.

`GET_CMD_LINE` is a basic copy of `O2`'s command line routine. It is changed as it just fetches the file to process. There are no other parameters to deal with. Have a read and compare `O2`'s routine against this one to see the variances. O_2's routines are written up in *common_externs.pdf* in the "./docs" folder.

`DUMP_ERROR_QUEUE` ditto's `O2`'s error processing routine. Testout's difference to O_2 is the use of its own error handler grammar. This is due to the different T vocabulary defined over O_2's vocabulary. This was done to show u what u'll be doing for your own compiler development as your T vocabulary will definitely be different from O_2. Have a look at the *testout_err_hdlr* grammar and see how it resolves the external file associated with the error T. The Error T processing is basicly a template using your specific grammar relative to its own language/Terminal Vocabulary.

As O2's *o2_error_hdlr.lex* grammar deals with its own specific errors, this is why there is not a generic error template grammar that uses only the `|+|` to trace all the T errors as anonymous entities. One could do this but u'll find that this is not too creative. Look at *o2_error_hdlr.lex*'s subrule with tandom errors — "file-inclusion" "bad filename" as an example.

Your own error reporting is open to your own code and style to format and notification. U might want to compare *testout_terminals.T* and *testout_err_symbols.T* definitions against O_2's vocabulary: *yacco2_terminals.T* and *yacco2_err_symbols.T* to get a feel on the range of Tes within O_2. What they stand for, and the different definition templates used: canned to full C++ definition / implementation.

7. GET_CMD_LINE — go fetch the fodder.
To note: the "File_to_parse" parameter is passed in as a reference which gets filled with the command line input. The file's existence check is done and if it exists, downstream parsing uses it passed in reference variable for the input tok_can container to parse its contents.

```
⟨ accrue testout code 2 ⟩ +≡
  void GET_CMD_LINE(int argc, char *argv[]
  , const char *Holding
  , std::string & File_to_parse
  , yacco2::TOKEN_GAGGLE & Errors)
  {
    using namespace std;
    using namespace NS_testout_err_symbols;
    using namespace yacco2;
    ofstream ofile;
    ofile.open(Holding, ios_base::out | ios::trunc);
    if (¬ofile) {
      CAbs_lr1_sym * sym = new Err_bad_filename(Holding);
      sym→set_external_file_id(1);
      sym→set_line_no(1);
      sym→set_pos_in_line(1);
      Errors.push_back(*sym);
      return;
    }
    if (argc ≡ 1) {
      char cmd_line[Max_buf_size];
      cout ≪ "Please␣enter␣Command␣line␣to␣process:␣";
      cin.get(cmd_line, Max_buf_size, '\n');
      ofile ≪ cmd_line;
      File_to_parse += cmd_line;
      ofile.close( );
    }
    else {
      for (int x = 1; x < argc; ++x) {
        ofile ≪ argv[x];
        File_to_parse += argv[x];
      }
      ofile.close( );
    }
    tok_can < std::ifstream > Cmd1_tokens(Holding);
      /* sets the table of files controlled by yacco2 for parsing T gpsing */
    ifstream ifile;
    ifile.open(File_to_parse.c_str( ));
    if (¬ifile) {
      CAbs_lr1_sym * sym = new Err_bad_filename(File_to_parse.c_str( ));
      sym→set_external_file_id(1);
      sym→set_line_no(1);
      sym→set_pos_in_line(1);
      Errors.push_back(*sym);
      return;
    }
    ifile.close( );
```

```
    return;
  }
```

8. DUMP_ERROR_QUEUE — the teller of horror.
The 2 **yacco2**::*PTR_LR1_eog__* added to the end-of-the-container allows the grammar's *start_rule* to be accepted.

⟨ accrue testout code 2 ⟩ +≡

```
void DUMP_ERROR_QUEUE(yacco2::TOKEN_GAGGLE & Errors)
{
  using namespace NS_yacco2_k_symbols;
  using namespace yacco2;
  Errors.push_back(*yacco2::PTR_LR1_eog__);
  Errors.push_back(*yacco2::PTR_LR1_eog__);
  using namespace NS_testout_err_hdlr;
  Ctestout_err_hdlr fsm;
  Parser pass_errors(fsm, &Errors, 0);
  pass_errors.parse( );
  yacco2::Parallel_threads_shutdown(pass_errors);
}
```

9. Mainline.
The mainline code demonstrates various ways to do "literate programming: a mixture of Literate programming code modules and standard "c" code.

Note: I checked the returned code from the parser *Parser*::*erred* as to whether a parse error occurred. This is the proper way to do things but u'll see my lazyness in O_2's program listing to just check the "Error queue". If u misused the `RSVP_FSM` macro from a monolithic grammar's *failed* directive to do this, the parser did abort, the *failed* directive fired but the "Error queue" is empty as the error T is placed within the "Accept queue" which is the context for a threaded grammar to return a result back to a calling grammar.

To see that *failed* works within the monolithic context, use "testout_error.dat" file as input.

So be forwarned. I guess this should be checked for by O_2 and to thrown an error when compiling the grammar but for the moment it does not.

```
⟨ accrue testout code 2 ⟩ +≡
  int main(int argc, char *argv[])
  {
    std::cout << "testout start" << std::endl;
    using namespace yacco2;
    ⟨ turn on some O2's tracing 11 ⟩;
    ⟨ get command line, parse it, and place contents into a holding file 12 ⟩;
    using namespace NS_pager_1;
    tok_can < std::ifstream > cmd_line(To_parse_file.c_str());
    Cpager_1 pager_1_fsm;
    Parser testout_parse(pager_1_fsm, &cmd_line, &IP_tokens, 0, &Error_queue, &JUNK_tokens, 0);
    if (testout_parse.parse() ≡ Parser::erred) {
      std::cout << "===============>ERROR OCCURRED" << std::endl;
      ⟨ if error queue not empty then deal with posted errors 13 ⟩;
    }
  exit:
    std::cout << "Exiting testout" << std::endl;
    return 0;
  }
```

10. Some Programming sections.

11. Turn on some of O_2's dynamic tracing variables.
Tracings:

T - terminals
TH - threads if their fsm-debug options turned on
MSG - messages between grammars
MU - trace mutex activity

Notice that MU was turned off but kept as a comment. Due to how *Cweb* deals with comments, the underscore "_" must be escaped. Have a look in the "testout_prog.w" file where u'll see this.

You will not see any "MSG" or "TH" tracings as there was not a threaded grammar called. So why do u include them? Just to pique your interest to O_2's delights.

⟨ turn on some O_2's tracing 11 ⟩ ≡
```
yacco2::YACCO2_T__ = 1;
yacco2::YACCO2_TH__ = 1;
yacco2::YACCO2_MSG__ = 1;      /* yacco2::YACCO2_MU_TRACING__ = 1; */
```
This code is used in section 9.

12. Get command line, parse it, and place contents into a holding file.

⟨ get command line, parse it, and place contents into a holding file 12 ⟩ ≡
```
GET_CMD_LINE(argc, argv, Holding_file.c_str( ), To_parse_file, Error_queue);
⟨ if error queue not empty then deal with posted errors 13 ⟩;
```
This code is used in section 9.

13. Do we have errors?.
Check that error queue for those errors. Note, `DUMP_ERROR_QUEUE` will also flush out any launched threads for the good housekeeping or is it housetrained seal award? Trying to do my best in the realm of short lived winddowns.

⟨ if error queue not empty then deal with posted errors 13 ⟩ ≡
```
if (Error_queue.empty( ) ≠ true) {
  DUMP_ERROR_QUEUE(Error_queue);
  return 1;
}
```
This code is used in sections 9 and 12.

14. Testout implementation.
Start the code output to *testout.cpp* by appending its include file.

⟨ `testout.cpp` 14 ⟩ ≡
⟨ accrue testout code 2 ⟩;

15. Index.

argc: 7, 9, 12.
argv: 7, 9, 12.
c_str: 7, 9, 12.
CAbs_lr1_sym: 7.
cin: 7.
close: 7.
cmd_line: 7, 9.
Cmd1_tokens: 7.
common_externs: 6.
cout: 7, 9.
Cpager_1: 9.
cpp: 14.
Ctestout_err_hdlr: 8.
Cweb: 11.
DUMP_ERROR_QUEUE: 6, 8, 13.
empty: 13.
endl: 9.
Err_bad_filename: 7.
erred: 9.
Error_queue: 5, 9, 12, 13.
Errors: 7, 8.
exit: 9.
failed: 9.
File_to_parse: 7.
fsm: 8.
get: 7.
GET_CMD_LINE: 6, 7, 12.
Holding: 7.
Holding_file: 5, 12.
ifile: 7.
ifstream: 7, 9.
ios: 7.
ios_base: 7.
IP_tokens: 5, 9.
JUNK_tokens: 5, 9.
lex: 6.
main: 9.
Max_buf_size: 5, 7.
NS_pager_1: 9.
NS_testout_err_hdlr: 8.
NS_testout_err_symbols: 4, 7.
NS_testout_T_enum: 4.
NS_testout_terminals: 4.
NS_yacco2_k_symbols: 4, 8.
ofile: 7.
ofstream: 7.
open: 7.
out: 7.
O2: 6.
o2_error_hdlr: 6.
pager_1: 4.
pager_1_fsm: 9.
Parallel_threads_shutdown: 8.
parse: 8, 9.
Parser: 8, 9.
pass_errors: 8.
pdf: 6.
PTR_LR1_eog__: 8.
push_back: 7, 8.
RSVP_FSM: 9.
set_external_file_id: 7.
set_line_no: 7.
set_pos_in_line: 7.
start_rule: 8.
std: 4, 5, 7, 9.
string: 5, 7.
sym: 7.
testout: 14.
testout__: 4.
testout_err_hdlr: 4, 6.
testout_err_symbols: 4, 6.
testout_parse: 9.
testout_terminals: 4, 6.
To_parse_file: 5, 9, 12.
tok_can: 7, 9.
TOKEN_GAGGLE: 5, 7, 8.
true: 13.
trunc: 7.
x: 7.
yacco2: 4, 5, 7, 8, 9, 11.
YACCO2_define_trace_variables: 5.
yacco2_err_symbols: 6.
YACCO2_MSG__: 11.
YACCO2_T__: 11.
yacco2_terminals: 6.
YACCO2_TH__: 11.

⟨ Include files 3 ⟩ Used in section 2.
⟨ accrue testout code 2, 5, 7, 8, 9 ⟩ Used in section 14.
⟨ get command line, parse it, and place contents into a holding file 12 ⟩ Used in section 9.
⟨ if error queue not empty then deal with posted errors 13 ⟩ Used in sections 9 and 12.
⟨ `testout.cpp` 14 ⟩
⟨ `testout.h` 4 ⟩
⟨ turn on some O_2's tracing 11 ⟩ Used in section 9.

TESTOUT

	Section	Page
License	1	1
Comments	2	2
Include files	3	3
Create header file for testout environment	4	3
Define variables	5	4
Local Routines	6	5
`GET_CMD_LINE` — go fetch the fodder	7	6
`DUMP_ERROR_QUEUE` — the teller of horror	8	8
Mainline	9	9
Some Programming sections	10	10
Turn on some of O_2's dynamic tracing variables	11	10
Get command line, parse it, and place contents into a holding file	12	10
Do we have errors?	13	10
Testout implementation	14	11
Index	15	12