1. License.

Gened date: January 30, 2015

2. Summary of Yacco2 and Linker Common External Routines.
These are the various procedures that parse Yac$_2$o$_2$'s grammar language and emit the grammar's c++ code and tex document with mpost generated diagrams. Each language construct has its appropriate external procedure that houses the monolithic grammar to start the parse. There is no namespace used to contain these routines as I felt that this was overkill. As this is a closed system, their grammars are not universal and cannot be re-cycled for others. Their only outside value are in teaching examples on "how to skin a cat" or is it "how to parse a lion?"...

External routines ratatouille:

o2externs.w - cweb generator file
o2_externs.h - header file
o2_externs.cpp - implementation

Dependency files from other Yacco2 sub-systems:

yacco2.h - basic definitions used by Yacco2
yacco2_T_enumeration.h - terminal enumeration for Yacco2's terminal grammar alphabet
yacco2_err_symbols.h - error terminal definitions from Yacco2's grammar alphabet
yacco2_characters.h - raw character definitions from Yacco2's grammar alphabet
yacco2_k_symbols.h - constant meta terminal defs from Yacco2's grammar alphabet
yacco2_terminals.h - regular terminal definitions from Yacco2's grammar alphabet
$*$. *h* - assorted grammar definitions for Yacco2's parsing
yacco2_stbl.h - symbol table defnitions

External procedures and other globals:

`GET_CMD_LINE` — global routine for public consumption
`DUMP_ERROR_QUEUE`
`DATE_AND_TIME`

3. Global definitions, External parse routines for Yacco2.
Why `CWEB_MARKER` external? It holds the *T_cweb_marker* tree containing *cweb* comments. Why? It is a holding pointer that allows *cweb* comments to be processed before the appropriate parse phrases like *process_xxx_phrase* that are triggered by `PROCESS_KEYWORD_FOR_SYNTAX_CODE`.

4. Create header file.

```
⟨ common_externs.h 4 ⟩ ≡
#ifndef common_extern_
#define common_extern_ 1
  ⟨ Preprocessor definitions ⟩
  ⟨ Files for header 5 ⟩;
#endif
```

5. Files for header.

```
⟨ Files for header 5 ⟩ ≡
#include "globals.h"
#include "/usr/local/yacco2/compiler/o2/o2_types.h"
#include "o2_lcl_opts.h"
#include "o2_lcl_opt.h"
#include "o2_err_hdlr.h"
  extern
  void GET_CMD_LINE(int argc, char *argv[], const char *File, yacco2::TOKEN_GAGGLE & Errors);
  extern
  void DUMP_ERROR_QUEUE(yacco2::TOKEN_GAGGLE & Errors);
  extern const char *DATE_AND_TIME( );
```

This code is used in section 4.

6. Include Header file.

```
⟨ Include Header file 6 ⟩ ≡
#include "common_externs.h"
```

This code is used in section 7.

7. Yacco2 external routines blueprint. Output of the code.

```
⟨ common_externs.cpp 7 ⟩ ≡
  ⟨ Include Header file 6 ⟩;
  ⟨ accrue source for emit 8 ⟩;
```

8. Accrue source for emit.

```
⟨ accrue source for emit 8 ⟩ ≡      /* add source code */
```

See also sections 10, 11, 12, and 13.

This code is used in section 7.

9. External routines.

10. Extract command line parameters: `GET_CMD_LINE`.
This is a simple routine to fetch the program's run parameters and place them into a passed holding file. If there is no command line data, it asks the user for the command line via *cin*. This holding file gets parsed first. The extraction process from the program environment is more general than required for example Linker. It will accept more than just the fsc file name. It is left to the likes of *xxxx_PARSE_CMD_LINE* and its associated grammar to determine whether the command line is acceptable.

Constraints:

ip1-2: argc and argv program run parameters
ip3: Holding file to place the command line or interactive data
ip4: Error token container for generated error token

Errors:

1) bad filename for holding file

⟨ accrue source for emit 8 ⟩ +≡

```
extern
void GET_CMD_LINE(int argc, char *argv[], const char *Holding, yacco2::TOKEN_GAGGLE & Errors)
{
  using namespace std;
  using namespace NS_yacco2_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) {      /* just program name : no parms */
    char cmd_line[Max_buf_size];
    cout ≪ "Please␣enter␣Command␣line␣to␣process:␣";
    cin.get(cmd_line, Max_buf_size, '\n');
    ofile ≪ cmd_line;
    ofile.close( );
    return;
  }
  for (int x = 1; x < argc; ++x) {
    ofile ≪ argv[x] ≪ '␣';
  }
  ofile.close( );
  return;
}
```

11. Dump Error queue tokens: DUMP_ERROR_QUEUE.
Its 5 minutes of fame is the use of a grammar to parse the error queue and format the outputted error message(s). It demonstrates how grammars can be used in various situations. The output is a tupple of 5 elements: file being parsed, the extracted source line, and its coordinates by line number and position within the line, and the error message with an arrow positioned within the above extracted souce line indicating the why-where-who-did-it. It also shutsdown the threads launched.

2 places are outputed to for gazing at those bugs. cout handles the screen addict while the Java sippers can flip thru Yacco2's log or is it a blog?:

op1: Yacco2's log file
op2: cout

Constraints:

ip1: Error token contain

⟨ accrue source for emit 8 ⟩ +≡

```
extern 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_o2_err_hdlr;
  Co2_err_hdlr fsm;
  Parser pass_errors(fsm, &Errors, 0);
  pass_errors.parse( );
  yacco2::Parallel_threads_shutdown(pass_errors);
}
```

12. DATE_AND_TIME.

⟨ accrue source for emit 8 ⟩ +≡

```
extern const char *DATE_AND_TIME( )
{
  static std::string dt;
  if (dt.empty( )) {
    time_t theTime = time(0);
    char *cTime = ctime(&theTime);
    dt += cTime;
    int n = dt.find('\n');
    dt[n] = '␣';
  }
  return dt.c_str( );
}
```

13. *XLATE_SYMBOLS_FOR_cweave*.

```
⟨ accrue source for emit 8 ⟩ +≡
  extern void XLATE_SYMBOLS_FOR_cweave(const char *Sym_to_xlate, char *Xlated_sym)
  {
    Xlated_sym[0] = (char) 0;
    int lll = strlen(Sym_to_xlate);
    int app_ptr = 0;
    for (int x = 0; x < lll; ++x, --app_ptr) {      /* app_ptr re-align to (chr)0 cuz rel to 1 so bkup */
      switch (Sym_to_xlate[x]) {
      case '{':
        {
          int sz = sizeof ("\\BRACEOPEN{}");
          strncpy(&Xlated_sym[app_ptr], "\\BRACEOPEN{}", sz);
          app_ptr += sz;
          continue;
        }
      case '}':
        {
          int sz = sizeof ("\\BRACECLOSE{}");
          strncpy(&Xlated_sym[app_ptr], "\\BRACECLOSE{}", sz);
          app_ptr += sz;
          continue;
        }
      case '<':
        {
          int sz = sizeof ("\\LTsign{}");
          strncpy(&Xlated_sym[app_ptr], "\\LTsign{}", sz);
          app_ptr += sz;
          continue;
        }
      case '>':
        {
          int sz = sizeof ("\\GTsign{}");
          strncpy(&Xlated_sym[app_ptr], "\\GTsign{}", sz);
          app_ptr += sz;
          continue;
        }
      case '@':
        {
          int sz = sizeof ("\\ATsign{}");
          strncpy(&Xlated_sym[app_ptr], "\\ATsign{}", sz);
          app_ptr += sz;
          continue;
        }
      case '|':
        {
          if (x + 2 < lll) {
            if (Sym_to_xlate[x + 2] ≠ '|') {
              int sz = sizeof ("\\Verticalbar␣");
```

```
      strncpy(&Xlated_sym[app_ptr], "\\Verticalbar␣", sz);
      app_ptr += sz;
      continue;
    }
  }
  else {
    int sz = sizeof ("\\Verticalbar␣");
    strncpy(&Xlated_sym[app_ptr], "\\Verticalbar␣", sz);
    app_ptr += sz;
    continue;
  }
  switch (Sym_to_xlate[x + 1]) {
  case '+':
    {
      int sz = sizeof ("\\ALLshift{}");
      strncpy(&Xlated_sym[app_ptr], "\\ALLshift{}", sz);
      app_ptr += sz;
      x += 2;
      continue;
    }
  case '.':
    {
      int sz = sizeof ("\\INVshift{}");
      strncpy(&Xlated_sym[app_ptr], "\\INVshift{}", sz);
      app_ptr += sz;
      x += 2;
      continue;
    }
  case '?':
    {
      int sz = sizeof ("\\QUEshift{}");
      strncpy(&Xlated_sym[app_ptr], "\\QUEshift{}", sz);
      app_ptr += sz;
      x += 2;
      continue;
    }
  case '|':
    {
      int sz = sizeof ("\\PARshift{}");
      strncpy(&Xlated_sym[app_ptr], "\\PARshift{}", sz);
      app_ptr += sz;
      x += 2;
      continue;
    }
  case 'r':
    {
      int sz = sizeof ("\\REDshift{}");
      strncpy(&Xlated_sym[app_ptr], "\\REDshift{}", sz);
      app_ptr += sz;
      x += 2;
      continue;
```

```
        }
      case 'p':
        {
          int sz = sizeof ("\\PROCshift{}");
          strncpy(&Xlated_sym[app_ptr], "\\PROCshift{}", sz);
          app_ptr += sz;
          x += 2;
          continue;
        }
      case 't':
        {
          int sz = sizeof ("\\TRAshift{}");
          strncpy(&Xlated_sym[app_ptr], "\\TRAshift{}", sz);
          app_ptr += sz;
          x += 2;
          continue;
        }
      default:
        {
          int sz = sizeof ("\\Verticalbar{}");
          strncpy(&Xlated_sym[app_ptr], "\\Verticalbar{}", sz);
          app_ptr += sz;
          continue;
        }
      }
    }
  case '+':
    {
      int sz = sizeof ("+");
      strncpy(&Xlated_sym[app_ptr], "+", sz);
      app_ptr += sz;
      continue;
    }
  case '-':
    {
      int sz = sizeof ("-");
      strncpy(&Xlated_sym[app_ptr], "-", sz);
      app_ptr += sz;
      continue;
    }
  case '*':
    {
      int sz = sizeof ("\\ASTERICsign{}");
      strncpy(&Xlated_sym[app_ptr], "\\ASTERICsign{}", sz);
      app_ptr += sz;
      continue;
    }
  case '#':
    {
      int sz = sizeof ("\\NOsign␣");
```

```
    strncpy(&Xlated_sym[app_ptr], "\\NOsign␣", sz);
    app_ptr += sz;
    continue;
  }
case '&':
  {
    int sz = sizeof ("\\AMPERsign{}");
    strncpy(&Xlated_sym[app_ptr], "\\AMPERsign{}", sz);
    app_ptr += sz;
    continue;
  }
case '%':
  {
    int sz = sizeof ("\\PERCENTsign{}");
    strncpy(&Xlated_sym[app_ptr], "\\PERCENTsign{}", sz);
    app_ptr += sz;
    continue;
  }
case '$':
  {
    int sz = sizeof ("\\DOLLARsign{}");
    strncpy(&Xlated_sym[app_ptr], "\\DOLLARsign{}", sz);
    app_ptr += sz;
    continue;
  }
case '\\':
  {    /* escaped */
    if (lll ≡ 1) {
      int sz = sizeof ("\\BKSLASHsign{}");
      strncpy(&Xlated_sym[app_ptr], "\\BKSLASHsign{}", sz);
      app_ptr += sz;
      continue;
    }
    switch (Sym_to_xlate[x + 1]) {
    case '\\':
      {
        if (lll ≡ 2) {      /* only a bkslash esc seq and not a TeX macro */
          int sz = sizeof ("\\BKSLASHsign{}");
          strncpy(&Xlated_sym[app_ptr], "\\BKSLASHsign{}", sz);
          app_ptr += sz;
          ++x;
          continue;
        }
        int sz = sizeof ("\\\\");
        strcat(Xlated_sym, "\\\\");
        app_ptr += sz;
        ++x;      /* this is adv past 2nd char; the loop handles */
        continue;
      }
    case '"':
```

```
    {
      int sz = sizeof ("\\DBLQUOTEsign{}");
      strncpy(&Xlated_sym[app_ptr], "\\DBLQUOTEsign{}", sz);
      app_ptr += sz;
      ++x;
      continue;
    }
  case '\'':
    {
      int sz = sizeof ("\\RTQUOTEsign{}");
      strncpy(&Xlated_sym[app_ptr], "\\RTQUOTEsign{}", sz);
      app_ptr += sz;
      ++x;
      continue;
    }
  case 'n':
    {
      int sz = sizeof ("\n");
      strncpy(&Xlated_sym[app_ptr], "\n", sz);
      app_ptr += sz;
      ++x;
      continue;
    }
  case 't':
    {
      int sz = sizeof ("\t");
      strncpy(&Xlated_sym[app_ptr], "\t", sz);
      app_ptr += sz;
      ++x;
      continue;
    }
  case 'r':
    {
      int sz = sizeof ("\r");
      strncpy(&Xlated_sym[app_ptr], "\r", sz);
      app_ptr += sz;
      ++x;
      continue;
    }
  case 'v':
    {
      int sz = sizeof ("\v");
      strncpy(&Xlated_sym[app_ptr], "\v", sz);
      app_ptr += sz;
      ++x;
      continue;
    }
  case 'b':
    {
      int sz = sizeof ("\b");
```

```
        strncpy(&Xlated_sym[app_ptr], "\b", sz);
        app_ptr += sz;
        ++x;
        continue;
      }
    case 'f':
      {
        int sz = sizeof ("\f");
        strncpy(&Xlated_sym[app_ptr], "\f", sz);
        app_ptr += sz;
        ++x;
        continue;
      }
    case 'a':
      {
        int sz = sizeof ("\a");
        strncpy(&Xlated_sym[app_ptr], "\a", sz);
        app_ptr += sz;
        ++x;
        continue;
      }
    case '?':
      {
        int sz = sizeof ("\?");
        strncpy(&Xlated_sym[app_ptr], "\?", sz);
        app_ptr += sz;
        ++x;
        continue;
      }
    default:
      {    /* */
        int sz = sizeof ("\\");
        strncpy(&Xlated_sym[app_ptr], "\\", sz);
        app_ptr += sz - 1;
        strncpy(&Xlated_sym[app_ptr], &Sym_to_xlate[x + 1], 1);
        ++app_ptr;
        strncpy(&Xlated_sym[app_ptr], "", 1);
        ++x;
        ++app_ptr;
        continue;
      }
    }
  }
case '^':
  {
    int sz = sizeof ("\\UPARROWsign{}");
    strncpy(&Xlated_sym[app_ptr], "\\UPARROWsign{}", sz);
    app_ptr += sz;
    continue;
  }
case '~':
```

```
        {
          int sz = sizeof ("\\TILDE{}");
          strncpy(&Xlated_sym[app_ptr], "\\TILDE{}", sz);
          app_ptr += sz;
          continue;
        }
      case '_':
        {
          int sz = sizeof ("\\_");
          strncpy(&Xlated_sym[app_ptr], "\\_", sz);
          app_ptr += sz;
          continue;
        }
      default:
        {
          strncpy(&Xlated_sym[app_ptr], &Sym_to_xlate[x], 1);
          ++app_ptr;
          strncpy(&Xlated_sym[app_ptr], "", 1);
          ++app_ptr;
          continue;
        }
      }
    }
  }
```

14. Index.

⟨ Files for header 5 ⟩ Used in section 4.
⟨ Include Header file 6 ⟩ Used in section 7.
⟨ accrue source for emit 8, 10, 11, 12, 13 ⟩ Used in section 7.
⟨ `common_externs.cpp` 7 ⟩
⟨ `common_externs.h` 4 ⟩

COMMON˙EXTERNS