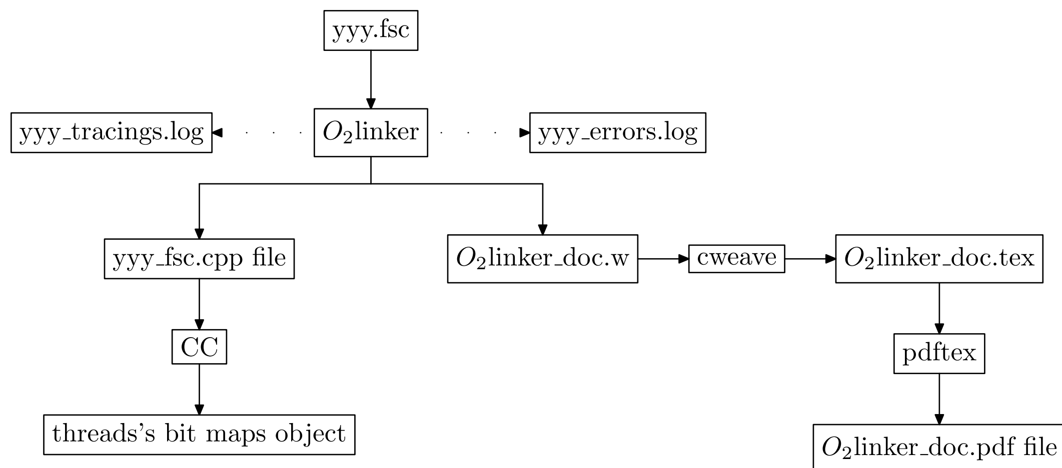
1. License.
Gened date: January 30, 2015
Copyright © 1998-2015 Dave Bone

This Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL was not distributed with this file, You can obtain one at http://mozilla.org/MPL/2.0/.

2. Summary of Yacco2's Linker — threads and their bit maps.
O_2^{linker}produces "thread bit maps" for each terminal and a global table of to-run threads, and a linker document describing all the processed grammars extracted from their grammar's "fsm-comments" and their called threads. Standalone grammars: monolithic y, are also traversed to add their thread booty to the global bit maps. As Linker is a companion program of Yacco2, please see Yacco2's documentation for appropriate development of data structures that are germain to both.

Linker receives its input either interactively or as a command line. The file to compile contains a list of grammar first set control files generated by Yacco2. Each grammar's "first set" file generated by Yacco2 uses a naming convention of the "thread name" supplied by the grammar's $fsm - filename$ with an extension of "fsc" indicating a "first set" control file. At present, Linker's inputted file is handcrafted by the grammar writer due to Yacco2 not having a generation database of compiled grammars. The below diagram shows O_2Linker's manufacturing assembly line for the "bit maps" object file within an Unix context where the "ld" linker program resolves the globals described in the following section. The "pdf" generated document provides an overview description of all the grammars in use. "CC" is the c++ compiler on my Sun Solaris work station.



The "O2linker_doc.pdf" document provides an index of all grammars compiled with comments about their claim-to-fame. It is a nice overview document of your language software when debugging or just getting a feel for all those defined grammars. It demarks the threaded grammars from their monolithic brethren.

3. Globals — those unresolved static variables used by Yacco2's library.
The following variables are made global for Yacco2's parse library usuage and placed within *yacco2* namespace:

1) `THDS_STABLE__` - vector of threads for the running
2) `T_ARRAY_HAVING_THD_IDS__` - terminals' thread sets
3) `BIT_MAPS_FOR_SALE__` - block of 32 bit words for bit map manufacture
4) `BIT_MAP_IDX__` - index used to dispense bit maps
5) `TOTAL_NO_BIT_WORDS__` - total number of words for sale

To speed things up and not be involved with the *malloc* issues of new / delete, `BIT_MAPS_FOR_SALE__` is a constant size of bit map words. Its size is controlled by the `TOTAL_NO_BIT_WORDS` macro and indicated at run-time by the `TOTAL_NO_BIT_WORDS__` global. Why not use c++ template containers? Depending on the context, they are not very efficient and in some cases they are not robust depending on whose compiler one uses. So "roll your own" is my mode of operandi. At least I can bark at myself instead of dealing with charades of good-citizenship a la Microsoft and their bug reporting - fixes: their stated policy of 24 hours response after submitting the bug report and now its 2 weeks and I'm still waiting for an acknowledgement. Have you seen their latest website for bug reporting? No direct button to send the report, only the platitude of how wastefull it is at their cost to deal with bug reports so make sure that you review their stated digest of bugs before submitting. How does democracy come into play when bugs are bugs: Fix them! Ranking bugs by developer votes is just (you substitute one of the following adjectives: stupid, senseless, witless, dull, brainless, weak-brained, muddle-headed, beef-witted, unentertaining). You get my sentiment.

Each global will be developed in their appropriate sections.

4. Some definitions within Linker's context.
What is a "first set" within the linker's context?
It is the terminals within the Start state configuration (aka "Closure only" state) of an automaton. There is some subtlety to this statement as an epsilon rule (empty right-hand-side), special terminals |+| and |.|, and threads being called from within the "Closure only" state require special treatment as their terminals are outside of this state. These situations require going into other state configurations to determine the terminals. Possibly an epsilon rule and special terminals but definitely threads presents a more difficult problem as the grammar being parsed does not have access to the called thread's first set. Why? A called thread is another grammar requiring compilation. This is why Linker is required to post process all grammars in a global context to calculate the first sets for grammars calling threads out of their "Closure only" state.

Why first sets anyway?
Originally this was not programmed. I wanted to prove my concepts first before optimizing for speed. Reality demands that a good idea be also practical. What good is parsing with threads when they are orders of magnitude slower than the current crop of parsing paradigns? So to overcome slowness, first sets provide the potential of whether a thread should run. How? If the current token to be parsed is in the first set of the thread then launch it from the running parse context. This optimization prevents false thread starts.

Why the name O_2^{linker} instead of for example "first setter" or jet?
I checked to the current name of tools required to prepare a program as an executable. In a sense Linker in my mind's eye was a concept that brought together the loose ends of the grammars and made it ready to parse. First setter is also appropriate.

How are first sets generated?
The basic problem solved by Linker is the following: within a grammar, threads can be called. If threads called are within the Start state of a grammar, the grammar's first set now has a dependence on another thread's first set. Of course this call sequence is transitive: thread A can call thread B can call thread C. Both threads B and C can have a call thread dependency within their first sets. It is this thread dependency that the Linker resolves by applying the transitive closure operation across a graph of thread nodes to outcome the explicit first set of terminals per dependent thread. The output from Linker is a global vector of threads to be run, and their global first set bit maps optimized against all terminals. That is, what threads can be run by this current terminal.

5. Overview of O_2^{linker}'s generated components.
Threads are just your normal procedures that get launched under the guidance of the threading facility native to the operating system instead of being called directly. Yacco2's parse library documentation goes into the detail between each run-time advantages threads versus procedure calls: After experimenting with a stop watch, the winner is the threads approach due to c++'s overhead of object birth-run-destructor cycle.

Now the issue becomes "how to determine and provide a thread id?" that can be referenced globally and locally. Why the distinction between global and local contexts? Local contexts are the individual grammars that can reference other threads who are defined globally outside of its domain. As raised before, Linker's raison-d'être is post-processing in a global context all the compiled threads. As the O_2 compiler / compiler processes the grammar in a local parsing context, references to threads within the grammar must be delayed thru indirection in its generated tables. This indirection comes from objects whose references are globally defined via the "extern" c(++) language facility and resolved by the appropriate language linker.

An entente between the global and local contexts was signed in 1066 whereby the charter of thread rights defined the *Thread_entry* structure. Each *Thread_entry* uses a naming convention of: Ixxx where xxx is the thread name without its namespace drawn from the grammar. O_2 generates references to these thread entries whilst Linker creates the actual thread entry entities. The *Thread_entry* contains a pointer to the literal thread name used in O_2's tracings, the address to its called procedure, and its thread id that is a key to appropriate tables and thread maps. Thread ids are positive whole numbers starting from 0. The thread names are sorted in ascending sequence. Each thread's id is its relative position within this sorted sequence.

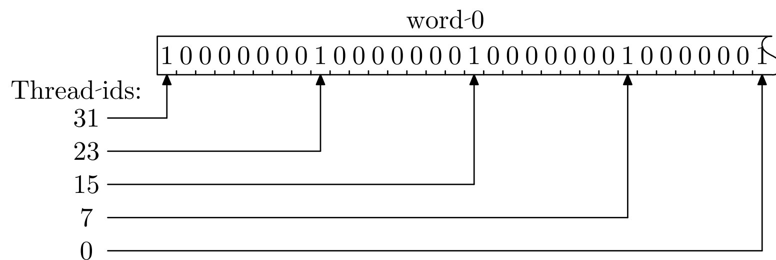
6. Thread bit maps.

To give speed and economy of space, 32 bit words are used whereby each bit within a word represents the presence or absence of a specific thread. Why 32 bit words? This falls nicely in the current crop of CPUs. As 64 bit envy becomes mainstream, the number of bits per word can be changed by *cweb*'s macro facility. The mapping of a thread id into its bit map, uses modulo 32 on the thread id. The quotient indicates which word to access and the remainder indicates which specific bit within the word represents the thread. Due to the open number of threads being defined, the total number of words making up a thread map is specific to each parsing environment. For example, the current number of threads for O_2 is under 50. So, each thread map is 2 words long.

Use of bit maps allows set processing with its operators like intersection to take place. Due to other optimizations to be discussed later, the number of words per bit map for Yac_2o_2's parsing library is calculated at runtime from the global `THDS_STABLE__` structure that carries the total number of threads being supported for the current parse environment. The globals `TOTAL_NO_OF_BIT_WORDS__`, `BIT_MAPS_FOR_SALE__`, and `BIT_MAP_IDX__` are used to manufacture bit maps. Why not manufacture bit maps at time of O_2 running? Neither the local grammar or O_2 has any notion to what its or other threads ids are. Also, I did not want to impose on the grammar writer an unique thread id assignment within each grammar nor an approximation of the total number of threads being developed.

Now, Yac_2o_2's runtime library has a just-in-time thread map manufacture process. Each parsing thread's state keeps a list of threads to possibly run but list processing to launch threads is too slow. So when a parsing thread arrives at a state configuration needing thread bit maps, Yac_2o_2's library will manufacture it using the globals `BIT_MAPS_FOR_SALE__` and `BIT_MAP_IDX__`. `BIT_MAP_IDX__` controls the current index into `BIT_MAPS_FOR_SALE__` that houses the words for the bit maps. This newly minted thread map is now available throughout the balance of the parsing of the thread by storing the map address in the state configuration.

The below diagram depicts a partial bit map configuration:



The word 0 above is in "big endian" sequence composed of 4 bytes: byte 0 is the first covering the most significant bit 31 thru to bit 24 while byte 3 contains the least significant bit 0 where its bits range from 7 thru to 0. It illustrates threads 31, 23, 15, 7, and 0 as being available for the running as their bits are "turned on" represented by "1". Word 1 would have the thread id range of 63 thru to 32. Each thread id in a word is just a replay of the modulo: Q is the word map no * 32 + the R the remainder indicating the specific bit from 0..31.

7. Linker's languages.

There are 3 separate languages to be parsed by Linker:

1) Terminal alphabet — literal names of each terminal
2) Each grammar's first set control file generated by O_2 — xxx.fsc
3) Linker's input control file

These languages are simplistic but requires proper parsing that, of course, will use Yacco2's parse library.

8. Terminal alphabet.
This is your terminal vocabulary defined by the 4 classes of terminals: LR constants, raw characters, errors, and "user created" terminals. Its content provides the literal description per terminal used for annotated comments in the emitted file. Their input order is also their enumeration. This file is outputted by O_2 when one requests the terminals to be compiled using one of its gen options: "-t" for user terminals or "-err" for errors. It uses the filename provided by the grammar's T-enumeration's file-name construct with a "fsc" extension (aka first set control). For the O_2 grammars, it looks like this:

```
T-enumeration (file-name yacco2_T_enumeration,name-space NS_yacco2_T_enum) {}
```

The outputted vocabulary file is "*yacco2_T_enumeration.fsc*" from this above example.

The language uses a bracketing construct "T-alphabet" ... "end-T-alphabet" to contain the list of terminals by their name defined by the grammar's "terminals" constructs: Lrk, raw characters, errors, and terminals. Below is an example:

```
T-alphabet
  LR1_questionable_shift_operator
  LR1_eog
  LR1_eolr
      ...
  T_file_of_T_alphabet
end-T-alphabet
```

9. Terminal enumeration.
As the inputted terminals are by name, to be more efficient, an enumeration scheme is used whereby each terminal's relative position within this list starting at 0 becomes their assigned number working out along the positive whole numbers. From the example above, the *LR1_questionable_shift_operator* terminal is assigned 0, *LR1_eog* terminal is assigned 1 and so on. In Yac_2o_2's documentation you'll find "how it applies" this enumeration to the shifting and reducing sets within the automaton tables. The storage for these sets favors space economy over speed.

10. First set declarations.
O_2 outputs per grammar their first set files having a file naming convention of the grammar name with an extension of "fsc". Below is "subrule_def.fsc" first set control file outputted by the subrule_def.lex grammar.

```
 1: /*
 2:  File: subrule_def.fsc
 3:  Date and Time: Fri Dec  7 16:28:47 2007
 4: */
 5: transitive   y
 6: grammar-name "subrule_def"
 7: name-space   "NS_subrule_def"
 8: thread-name  "TH_subrule_def"
 9: monolithic   n
10: file-name    "subrule_def.fsc"
11: no-of-T      567
12: list-of-native-first-set-terminals 0
13: end-list-of-native-first-set-terminals
14: list-of-transitive-threads 1
15:   NS_subrule_vector::TH_subrule_vector
16: end-list-of-transitive-threads
17: list-of-used-threads 10
18:   NS_cweb_marker::TH_cweb_marker
19:   NS_dbl_colon::TH_dbl_colon
20:   NS_identifier::TH_identifier
21:   NS_lint_balls::TH_lint_balls
22:   NS_o2_sdc::TH_o2_sdc
23:   NS_parallel_oper::TH_parallel_oper
24:   NS_rhs_bnd::TH_rhs_bnd
25:   NS_rhs_component::TH_rhs_component
26:   NS_rtn_component::TH_rtn_component
27:   NS_subrule_vector::TH_subrule_vector
28: end-list-of-used-threads
29: fsm-comments
30: "Parse a subrule: into the valley of sin..."
31:
```

This is a summary of what was found within the grammar. I included line numbers each delimited by a ":" to the left of the language's individual source lines as reference points for the discussion that follows. Lines 5 – 11 above are the prelude declarations of the compiled grammar. The 2 items of interest within the prelude are the keywords "transitive" and "monolithic". Both use a boolean declaration of "y" or "n". The "transitive" keyword indicates whether this grammar needs transitive processing. The "monolithic" keyword indicates whether the grammar is stand alone or threaded.

The 3 components following the prelude indicate the intent by their names. Lines 12 – 13 houses the "list-of-native-first-set-terminals" component: the grammar's explicitly used terminals within its first set where there this example has none. The "list-of-transitive-threads" component: lines 14 – 16 indicates the directly called threads from their first set. This list is recursively called by assessing their called threads's "fsc" files: the transitive adjective is a memory jog to recursion. Please see *first_set_for_threads* grammar on how the thread's first set is calculated as there is a subtle difference caused by the `|.|` in its first set calculation versus the regular first set calcaluation of a rule or lr state. Lines 17 – 28 is a cross reference list of used threads throughout the grammar and not just the first set. Lines 29 – 30 is the extracted fsm-comments from the grammar. It is used in generating the Linker's document indexing all the grammars with their comments, and their called thread graph with highlighting of nested grammars.

11. Linker's control language.
There is not much to this control file. It provides the terminal vocabulary file, the file to output by Linker, a preamble that allows the grammar writer to insert code ahead of the "to be emitted" code, and finally the list of grammars' "fsc" files to process. Here is a sample of a handcrafted Linker control file drawn from the O_2 parsing environment.

```
file-of-T-alphabet "c:/yacco2/compiler/grammars/yacco2_T_enumeration.fsc"
emitfile "/yacco2/compiler/grammars/yacco2_fsc.cpp"
preamble
#include <yacco2.h>
#include <yacco2_T_enumeration.h>
#include <yacco2_err_symbols.h>
#include <yacco2_k_symbols.h>
#include <yacco2_terminals.h>
#include <yacco2_characters.h>
using namespace NS_yacco2_T_enum;
using namespace NS_yacco2_err_symbols;
using namespace NS_yacco2_k_symbols;
using namespace NS_yacco2_terminals;
using namespace NS_yacco2_characters;
end-preamble
"/yacco2/compiler/grammars/error_symbols_phrase.fsc"
"/yacco2/compiler/grammars/error_symbols_phrase_th.fsc"
"/yacco2/compiler/grammars/angled_string.fsc"
"/yacco2/compiler/grammars/bad_char_set.fsc"
"/yacco2/compiler/grammars/c_comments.fsc"
                    ...
```

Basicly, Linker builds a graph by grammar and does its transitive moves to produce concrete first sets for each unresolved grammar or thread. To improve performance, a global thread map per terminal will be created indicating the potential threads that can be run.

12. Catalogue of Linker's files.
Linker's Input files to *cweb*:

1) *o2linker.w* — master Linker file that starts things off
2) *intro.w* — inroduction
3) *prog.w* — linker cweb code
4) *o2linker_externs.w* — external procedures
5) *pms.w* — running comments of life
6) *bugs.w* — yep i do them
7) *testsuites.w* — self proof of code
8) *sampleoutput.w*
9) *types.w* — provides a way around burping output formats
10) *includes.w* — C++ include code
11) /usr/local/yacco2/externals/common_defs.w — external procedures
12) *o2linker_defs.w* — external procedures

cweb generated files:

1) *o2linker.h* — linker definitions
2) *o2linker.cpp* — linker program
3) *xxx.cpp* — first sets for grammars provided by the input file

Yac_2o_2 library memorabilia:

1) yacco2 — library namespace
2) "/usr/local/yacco2/library" — Yac_2o_2's library directory
3) $< yacco2.h >$ — Yacco2's library header file
4) "/usr/local/yacco2/library/lib/xxxx" - xxxx is the Debug or Release of the object library
5) "yacco2build.a" - static Yac_2o_2's library

Dependency files from other Yacco2 sub-systems:

yacco2.h - basic definitions used by Yacco2
yacco2_T_enumeration.h - terminal enumeration for Yac_2o_2's terminal grammar alphabet
yacco2_err_symbols.h - error terminal definitions from Yac_2o_2's grammar alphabet
yacco2_characters.h - raw character definitions from Yac_2o_2's grammar alphabet
yacco2_k_symbols.h - constant terminal definitions from Yac_2o_2's grammar alphabet
yacco2_terminals.h - regular terminal definitions from Yac_2o_2's grammar alphabet
$*.\ h$ - assorted grammar definitions from Yac_2o_2 to parse
o2linker_externs.h - external support routines common for O_2^{linker}

Dictionaries

`T_DICTIONARY` — terminals
`GRAMMAR_DICTIONARY`
`YACCO2_STBL` — symbol table
`USED_THREADS_LIST`
`THREAD_ID_FIRST_SET`
`T_THREAD_ID_LIST`

Grammars

T_alphabet.lex
linker_pass3.lex — control file parser
link_cleanser.lex — lexical grammar stripping off comments etc
fsc_file.lex — syntactic grammar of fsc file

Document

o2linker_doc.w — overall index describing the grammars processed: cweave / pdftex

13. Global macro definitions.

```
#define SPECULATIVE_NO_BIT_WORDS 20
#define LR1_QUESTIONABLE_SHIFT_OPERATOR 0
#define LR1_EOG 1
#define LR1_EOLR 2
#define LR1_PARALLEL_OPERATOR 3
#define LR1_REDUCE_OPERATOR 4
#define LR1_PROCEDURE_CALL_OPERATOR 7
#define LR1_INVISIBLE_SHIFT_OPERATOR 5
#define LR1_ALL_SHIFT_OPERATOR 6
#define LR1_FSET_TRANSIENCE_OPERATOR 7
#define SMALL_BUFFER_4K 1024 * 4
#define BITS_PER_WORD 32
#define TOTAL_NO_BIT_WORDS 1024 * 2 * 100
```

14. External routines and globals.
General routines to get things going:

1) get control file and put into Linker's holding file
2) parse the command line
3) format errors

These are defined by including *o2_externs.h*.

The globals are:

1) *Error_queue* — global container of errors passed across all parsings
2) GRAMMAR_DICTIONARY — thread container whose contents point into the symbol table
3) T_DICTIONARY — terminal container whose contents point into the symbol table
4) T_THREAD_ID_LIST — thread id list per terminal
5) NO_OF_THREADS — found number of threads
6) NO_WORDS_FOR_BIT_MAP — calculated number of words per thread bit map
7) THREAD_ID_FS — first set per thread id

⟨ External rtns and variables 14 ⟩ ≡

```
extern int NO_OF_THREADS;
extern int NO_WORDS_FOR_BIT_MAP;
```

See also section 28.
This code is used in section 88.

15. 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 15 ⟩ ≡

```
if (Error_queue.empty() ≠ true) {
  DUMP_ERROR_QUEUE(Error_queue);
  return 1;
}
```

This code is used in sections 17, 18, 19, 21, and 66.

16. Local routines.

17. Parse linker control file.
Handcrafted file that gathers all the grammar control files together for the processing. Note the use of *tok_can* < *std* :: *ifstream* > to read raw characters and produce tokens in just-in-time.

```
⟨ parse linker control file 17 ⟩ ≡
  cout ≪ "Parse␣linker␣control␣file␣" ≪ endl;
  using namespace NS_linker_pass3;
  tok_can < std :: ifstream > cntl_file_tokens(cntl_file_name.c_str( ));
  TOKEN_GAGGLE P3_tokens;
  Clinker_pass3 linker_cntl_file_fsm;
  Parser linker_cntl_file(linker_cntl_file_fsm, &cntl_file_tokens, &P3_tokens, 0, &Error_queue, 0, 0);
  linker_cntl_file.parse( );
  ⟨ if error queue not empty then deal with posted errors 15 ⟩;
```

This code is used in section 66.

18. Parse T alphabet.
Ahh the terminal vocabulary. The symbol's position within the list is also its enumerate value starting from 0. The *T_alphabet* grammar places the terminal symbols into the symbol table as a 1:1 mapping, into the `T_DICTIONARY` as a consolidated repository and into the `T_THREAD_ID_LIST` which is the terminal to thread list relationship used to quickly determine if the current token has the potential to run as a thread. Each thread contains its first set list squirreled away in thread_attributes's *list_of_Ts_*.

```
⟨ parse T alphabet 18 ⟩ ≡
  cout ≪ "Parse␣alphabet" ≪ endl;
  using namespace NS_link_cleanser;
  tok_can < std :: ifstream > T_file_tokens(linker_cntl_file_fsm.t_alphabet_filename_.c_str( ));
  TOKEN_GAGGLE cleanser_tokens;
  Clink_cleanser cleanser_fsm;
  Parser cleanser(cleanser_fsm, &T_file_tokens, &cleanser_tokens, 0, &Error_queue, 0, 0);
  cleanser.parse( );
  using namespace NS_t_alphabet;
  TOKEN_GAGGLE T_tokens;
  Ct_alphabet T_fsm;
  Parser T_pass3(T_fsm, &cleanser_tokens, &T_tokens, 0, &Error_queue, 0, 0);
  T_pass3.parse( );
  ⟨ if error queue not empty then deal with posted errors 15 ⟩;
```

This code is used in section 66.

19. Parse fsc files.

Digest those fsc files. Burp. Better that than the other end. Now for a chiro for the stretched stomach muscles. The *fsc_file* grammar puts the digested grammar's fsc content into the symbol table of thread attribute's object and its reference into the `GRAMMAR_DICTIONARY`.

```
⟨ parse fsc files 19 ⟩ ≡
  cout ≪ "Parse␣fsc␣files" ≪ endl;
  TOKEN_GAGGLE cleanser_fsc_tokens;
  TOKEN_GAGGLE fsc_file_output_tokens;
  std::vector < std::string > ::iterator ii = linker_cntl_file_fsm.grammars_fsc_files_.begin();
  std::vector < std::string > ::iterator iie = linker_cntl_file_fsm.grammars_fsc_files_.end();
  for ( ; ii ≠ iie; ++ii) {
    tok_can < std::ifstream > T_fsc_file_tokens(ii→c_str());
    Clink_cleanser cleanser_fsc;
    Parser fsc_cleanser(cleanser_fsc, &T_fsc_file_tokens, &cleanser_fsc_tokens, 0, &Error_queue, 0, 0);
    fsc_cleanser.parse();
    ⟨ if error queue not empty then deal with posted errors 15 ⟩;
    using namespace NS_fsc_file;
    Cfsc_file fsc_file_fsm;
    Parser fsc_file_pass(fsc_file_fsm, &cleanser_fsc_tokens, &fsc_file_output_tokens, 0, &Error_queue, 0, 0);
    fsc_file_pass.parse();
    ⟨ if error queue not empty then deal with posted errors 15 ⟩;
    cleanser_fsc_tokens.clear();
    fsc_file_output_tokens.clear();
  }
  ⟨ if error queue not empty then deal with posted errors 15 ⟩;
```

This code is used in section 66.

20. *load_linkkw_into_tbl*.
These are the keywords from all the languages to be parsed. So why the loading up of keywords? Speed. *linker_id* does a symbol table lookup for O_2^{linker}. So where ever appropriate like the "first set control" files, normal boundary parsing can take place. There is your lexical grammar that discriminates characters into tokens like keywords, comments followed by a separate syntax grammar to process the language structure. See comments in Yacco2's external document regarding the reason for the kludge.

```
⟨ accrue linker code 20 ⟩ ≡
  void load_linkkw_into_tbl(yacco2::CAbs_lr1_sym *Kw)
  {
    using namespace yacco2_stbl;
    T_sym_tbl_report_card report_card;
    KCHARP kwkey = Kw→id();
    if (*kwkey ≡ '#') ++kwkey;    /* kludge: bypass 1st char eg "#fsm" */
    kw_in_stbl *kw = new kw_in_stbl(Kw);
    add_sym_to_stbl(report_card, *kwkey, *kw, table_entry::defed, table_entry::keyword);
    kw→stbl_idx(report_card.pos_);
  }
  void load_linkkws_into_tbl()
  {
    cout ≪ "Load␣linker's␣keywords␣" ≪ endl;
    load_linkkw_into_tbl(new T_transitive);
    load_linkkw_into_tbl(new T_grammar_name);
    load_linkkw_into_tbl(new T_name_space);
    load_linkkw_into_tbl(new T_thread_name);
    load_linkkw_into_tbl(new T_fsm_comments);
    load_linkkw_into_tbl(new T_monolithic);
    load_linkkw_into_tbl(new T_file_name);
    load_linkkw_into_tbl(new T_no_of_T);
    load_linkkw_into_tbl(new T_list_of_native_first_set_terminals);
    load_linkkw_into_tbl(new T_end_list_of_native_first_set_terminals);
    load_linkkw_into_tbl(new T_list_of_transitive_threads);
    load_linkkw_into_tbl(new T_end_list_of_transitive_threads);
    load_linkkw_into_tbl(new T_list_of_used_threads);
    load_linkkw_into_tbl(new T_end_list_of_used_threads);
    load_linkkw_into_tbl(new T_T_alphabet);
    load_linkkw_into_tbl(new T_end_T_alphabet);
    load_linkkw_into_tbl(new T_file_of_T_alphabet);
    load_linkkw_into_tbl(new T_emitfile);
    load_linkkw_into_tbl(new T_preamble);
    load_linkkw_into_tbl(new T_end_preamble);
  }
```

See also sections 22, 33, 38, 39, 40, 52, 53, 54, 55, 56, 60, and 66.
This code is used in section 89.

21. Verify that all threads used are defined.
Simple check! Just sequentially read the `GRAMMAR_DICTIONARY` whose elements are pointers to their symbol table registry where the entry is not "defined" but referenced by some transitive call list. The loop just pours the rogues into the error queue and uses the token co-ordinates that created the entry as "used" for the error message. This allows the error dump to pinpoint the source file and specific line that referenced the rogue thread. Correction is to add the missing grammar to the control file list or correct the grammar that issued the thread call.

```
⟨ post verify that there are no threads "used" and not "defined" 21 ⟩ ≡
  std::vector < table_entry *> ::iterator th_i = GRAMMAR_DICTIONARY.begin();
  std::vector < table_entry *> ::iterator th_ie = GRAMMAR_DICTIONARY.end();
  for ( ; th_i ≠ th_ie; ++th_i) {
    table_entry * tbl_entry = *th_i;
    if (tbl_entry→defined_ ≡ true) continue;
    CAbs_lr1_sym * sym = new Err_bad_th_in_list;
    sym→set_who_created("linker.w", __LINE__);
    thread_attributes*th_goodies=␣((th_in_stbl*)tbl_entry->symbol_)->thread_in_stbl();
    sym→set_rc(*th_goodies);
    Error_queue.push_back(*sym);
  }
  ⟨ if error queue not empty then deal with posted errors 15 ⟩;
```

This code is used in section 66.

22. Sort thread dictionary.
The grammars are divided into 2 types:
1) monolithic — stand alone grammars
2) called threads
To facilitate the emitted code, the following partial order is imposed on the fully qualified name (FQN) of the grammar composed of the grammar's namespace and name separated by "::": for example, *NS_eol* :: *TH_eol*. This is the calling handle of the thread when used with the ||| statement. The partial order is divided into 2 parts — threads followed by standalone grammars:
thread vs thread — lexicographical order on "thread name" only
thread vs mono — thread less than mono grammar
mono vs thread — thread less than mono grammar
mono vs mono — lexicographical order on FQN
The order defines explicitly the enumerate value assigned to each thread starting from 0. The standalone grammars (monolithic) are not part of the thread stable that gets emitted.

⟨ accrue linker code 20 ⟩ +≡
```
bool sort_threads_criteria(const table_entry *P1, const table_entry *P2)
{
  th_in_stbl*th_tbl1=␣(th_in_stbl*)P1->symbol_;
  th_in_stbl*th_tbl2=␣(th_in_stbl*)P2->symbol_;
  thread_attributes * p1 = th_tbl1→thread_in_stbl( );
  thread_attributes * p2 = th_tbl2→thread_in_stbl( );
  int len_a = p1→thread_name_→c_string( )→size( );
  int len_b = p2→thread_name_→c_string( )→size( );
  string ucase_a;
  for (int x = 0; x < len_a; ++x) {
    ucase_a += toupper((*p1→thread_name_→c_string( ))[x]);
  }
  string ucase_b;
  for (int x = 0; x < len_b; ++x) {
    ucase_b += toupper((*p2→thread_name_→c_string( ))[x]);
  }
  if (len_a < len_b) {
    for (int x = len_a + 1; x ≤ len_b; ++x) ucase_a += '␣';
  }
  else {
    for (int x = len_b + 1; x ≤ len_a; ++x) ucase_b += '␣';
  }
  int result;
  if (p1→monolithic_ ≡ 'n') {    /* a:thread */
    if (p2→monolithic_ ≡ 'n') {    /* a:thread b:thread */
      result = strcmp(ucase_a.c_str( ), ucase_b.c_str( ));
      if (result < 0) return true;    /* a less b */
      return false;    /* a gt b */
    }
    return true;    /* a:thread b:mono; a less b */
  }
  if (p2→monolithic_ ≡ 'n') {    /* a:mono b:thread */
    return false;    /* a gt b */
  }    /* a:mono b:mono changed to the thread name instead of fqn */
  int len_fqna = p1→thread_name_→c_string( )→size( );
```

```
    int len_fqnb = p2→thread_name_→c_string()→size();
    string ucase_fqna;
    for (int x = 0; x < len_fqna; ++x) {
      ucase_fqna += toupper((*p1→thread_name_→c_string())[x]);
    }
    string ucase_fqnb;
    for (int x = 0; x < len_fqnb; ++x) {
      ucase_fqnb += toupper((*p2→thread_name_→c_string())[x]);
    }
    if (len_fqna < len_fqnb) {
      for (int x = len_fqna + 1; x ≤ len_fqnb; ++x) ucase_fqna += '␣';
    }
    else {
      for (int x = len_fqnb + 1; x ≤ len_fqna; ++x) ucase_fqnb += '␣';
    }
    result = strcmp(ucase_fqna.c_str(), ucase_fqnb.c_str());
    if (result < 0) return true;   /* a less b */
    return false;   /* a gt b */
  }
```

23. Sort uses template algorithm.

```
⟨sort thread dictionary 23⟩ ≡
  cout ≪ "Sort␣thread␣dictionary" ≪ endl;
  stable_sort(GRAMMAR_DICTIONARY.begin(), GRAMMAR_DICTIONARY.end(), sort_threads_criteria);
```

This code is used in section 66.

24. Dump sorted dictionary.

Not another reality show? Yupp.

```
⟨dump sorted dictionary 24⟩ ≡
  yacco2::lrclog ≪ "Sorted␣thread␣dictionary" ≪ GRAMMAR_DICTIONARY.size() ≪ std::endl;
  std::vector < table_entry *> ::iterator dth_i = GRAMMAR_DICTIONARY.begin();
  std::vector < table_entry *> ::iterator dth_ie = GRAMMAR_DICTIONARY.end();
  int pos(−1);
  for ( ; dth_i ≠ dth_ie; ++dth_i) {
    ++pos;
    table_entry *tbl_entry = *dth_i;
    thread_attributes*th_goodies=␣((th_in_stbl*)tbl_entry->symbol_)->thread_in_stbl();
    yacco2::lrclog ≪ "tbl␣entry*:␣" ≪ tbl_entry ≪ "␣th_goodies*:␣" ≪ th_goodies ≪ "␣" ≪
        pos ≪ ":" ≪ th_goodies→th_enum_ ≪ "␣mono:␣" ≪ th_goodies→monolithic_ ≪
        "␣thread␣name:␣" ≪ th_goodies→thread_name_→c_string()→c_str() ≪
        "␣FQN:␣" ≪ th_goodies→fully_qualified_th_name_.c_str() ≪ "␣K:␣" ≪
        th_goodies→fsm_comments_→c_string()→c_str() ≪ std::endl;
  }
```

This code is used in section 66.

25. Count and re-align threads enumerate values to sorted position.
The NO_OF_THREADS is calculated at the same time. It is used to indicate thread presence and the numbers of threads to emit.

```
⟨count and re-align threads enumerate values to sorted position 25⟩ ≡
  std::vector < table_entry *> ::iterator ri = GRAMMAR_DICTIONARY.begin( );
  std::vector < table_entry *> ::iterator rie = GRAMMAR_DICTIONARY.end( );
  for (int p = −1; ri ≠ rie; ++ri) {
    ++p;
    table_entry * tbl_entry = *ri;
    thread_attributes*th_goodies=␣((th_in_stbl*)tbl_entry->symbol_)->thread_in_stbl();
    th_goodies→th_enum_ = p;
    if (th_goodies→monolithic_ ≡ 'n') ++NO_OF_THREADS;
  }
```

This code is used in section 66.

26. Check whether Linker has enough space to generate the thread bit maps.
See *Passover on code* for my comments.

```
⟨check whether Linker has enough space to gen thread bit maps: no throw up 26⟩ ≡
  int max_thds_supported = SPECULATIVE_NO_BIT_WORDS * BITS_PER_WORD;
  if (NO_OF_THREADS > max_thds_supported) {
    char a[SMALL_BUFFER_4K];
    ;
    KCHARP msg = "Error:␣not␣enough␣space␣for␣thread␣bit␣\
        map␣manufacture!""␣#␣threads:␣%i,␣Linker's␣maximum␣no␣of␣threads␣supported:␣%\
        i.␣\n""␣Please␣expand␣SPECULATIVE_NO_BIT_WORDS";
    sprintf(a, msg, NO_OF_THREADS, max_thds_supported);
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
```

This code is used in section 66.

27. Thread graphs: first set generation.

28. *Visit_graph*.

⟨ External rtns and variables 14 ⟩ +≡
```
extern char Visit_graph[RESERVE_FIXED_NO_THREADS];
```

29. Probagate |+|.
Not much to it. The "all shift" operator indicates all terminals. It is a wild token facility that eases the pain in using grammars. So, all terminals except some meta operators must be placed into the first set and the thread id against each terminal.

To lower the O_2^{linker} document, the |+| meta terminal representing all terminals substitutes "eolr" in the thread's first set. It certainly makes for a cleaner document without the slug fest.

⟨ probagate |+| 29 ⟩ ≡
```
INT_SET_ITER_type t_listi = Root_thread.fs_.find(LR1_EOLR);     /* substitute eolr */
if (t_listi ≡ Root_thread.fs_.end()) {
  Root_thread.fs_.insert(LR1_EOLR);
}
int no_of_T = T_DICTIONARY.size();     /* rel 1 instead of 0 */
for (int x = 0; x < no_of_T; ++x) {
  switch (x) {
  case LR1_QUESTIONABLE_SHIFT_OPERATOR: break;
  case LR1_EOG: break;
  case LR1_EOLR: continue;
  case LR1_PARALLEL_OPERATOR: continue;
  case LR1_REDUCE_OPERATOR: continue;
  case LR1_INVISIBLE_SHIFT_OPERATOR: continue;
  case LR1_ALL_SHIFT_OPERATOR: continue;
  case LR1_FSET_TRANSIENCE_OPERATOR: continue;
  default: break;
  }
  if (Visited_th.monolithic_ ≡ 'n') {
    INT_SET_type &th_list = T_THREAD_ID_LIST[x];
    if (th_list.find(Visited_th.th_enum_) ≡ th_list.end()) {
      th_list.insert(Visited_th.th_enum_);
    }
    if (th_list.find(Root_thread_id) ≡ th_list.end()) {
      th_list.insert(Root_thread_id);
    }
  }
}
```
This code is used in section 31.

30. Deal with threads having T in first set.
Please note that threads only are dealt with and not their standalone brethren. The root grammar is still traversed so that its called threads can be added to the list.

Though the monolithic grammar is not launched by its first set, I included its first set calculations to verify my work.

```
⟨ deal with threads having T in first set 30 ⟩ ≡
  INT_SET_ITER_type t_listi = Root_thread.fs_.find(t_enum);
  if (t_listi ≡ Root_thread.fs_.end()) {
    Root_thread.fs_.insert(t_enum);
  }
  if (Visited_th.monolithic_ ≡ 'n') {
    INT_SET_type & th_list = T_THREAD_ID_LIST[t_enum];
    if (th_list.find(Visited_th.th_enum_) ≡ th_list.end()) {
      th_list.insert(Visited_th.th_enum_);
    }
    if (th_list.find(Root_thread_id) ≡ th_list.end()) {
      th_list.insert(Root_thread_id);
    }
  }
```

This code is used in section 31.

31. Associate native terminals with called thread.
For the moment until i can refine the thread's first set algorithm in O_2 that generates the Tes for "list-of-native-first-set-terminals", i force "attempting to run" the threads having the |t| in their first set across all Tes. There will be thread stutters in firing them up and then shuting them down when their true first set is outside of the current token. Remember these threads are fired up when they are in the being run grammar's current lr parse state. So the speed bump shouldn't be too big.

```
⟨ associate native terminals with called thread 31 ⟩ ≡
  std::vector < int > ::iterator fi = Visited_th.list_of_Ts_.begin();
  std::vector < int > ::iterator fie = Visited_th.list_of_Ts_.end();
  for ( ; fi ≠ fie; ++fi) {
    int t_enum = *fi;
    if (t_enum ≡ LR1_ALL_SHIFT_OPERATOR) {
      ⟨ probagate |+| 29 ⟩;
    }
    ⟨ deal with threads having T in first set 30 ⟩;
  }
```

This code is used in section 33.

32. Process called thread's list.
Walk the list and recursively call that graph.

```
⟨ process called thread's list 32 ⟩ ≡
  std::vector < thread_attributes *> ::iterator li = Visited_th.list_of_transitive_threads_.begin();
  std::vector < thread_attributes *> ::iterator lie = Visited_th.list_of_transitive_threads_.end();
  for ( ; li ≠ lie; ++li) {
    thread_attributes * th_att = *li;
    lrclog ≪ "------->process␣called␣thread's␣list␣thd:␣" ≪
        th_att→thread_name_→c_string()→c_str() ≪ "␣for␣root␣thd␣id:␣" ≪ Root_thread_id ≪ endl;
    crt_fset_of_thread(*th_att, Root_thread_id, Root_thread);
  }
```

This code is used in section 33.

33. *crt_fset_of_thread*.
Recursive procedure that chews gum, pats its stomach, and belches fire.
It traverses the called threads recursively to continue the association of the inherited terminals into their thread bit maps. Uses the *Visit_graph* to prevent self looping: whichever way u call it left or right recursion depending on your context — cheers or bottoms-up.

The *Root_thread_id* associates the starting thread id to the traversed called threads' first sets' terminals. Each thread is processed individually to associate itself with their called brethern's first sets.

```
⟨ accrue linker code 20 ⟩ +≡
  void crt_fset_of_thread(thread_attributes & Visited_th, int Root_thread_id, thread_attributes & Root_thread)
  {
    if (Visit_graph[Visited_th.th_enum_] ≡ 'y') return;
    Visit_graph[Visited_th.th_enum_] = 'y';
    ⟨ associate native terminals with called thread 31 ⟩;
    ⟨ process called thread's list 32 ⟩;
  }
```

34. Allocation space for *Visit_graph*.
Before, reserve allocated its space, now MS throws an error as *push_back* not done.

```
⟨ allocation space for Visit_graph 34 ⟩ ≡
  for (int vi = 0; vi < NO_OF_THREADS; ++vi) {
    Visit_graph[vi] = 'n';
  }
```

This code is used in section 36.

35. Initialize *Visit_graph* **to "not visited".**
Each new thread having its final fist set's gened presets the graph.

```
⟨ initialize Visit_graph to not visited 35 ⟩ ≡
  for (int vi = 0; vi < NO_OF_THREADS; ++vi) {
    Visit_graph[vi] = 'n';
  }
```

This code is used in sections 36 and 40.

36. Generate those first sets.
Walk the what? u threads fulfull your first set destinies. Not very complex. A visit graph is built having its nodes equal in number to the threads in the GRAMMAR_DICTIONARY. Each node is initialized to "not visited". From here, it's just process each thread and visit its called threads to inherit their native terminals: Of course this is a transitive process. Sorry for the let down but there ain't much to it.

A secondary graph of called threads per grammar is built so that an overall linker document can be emitted with:

1) index of threads sorted by their name with its fsm's comments
2) followed by the monolithic grammars
3)) each grammar will have its called threads

```
⟨ generate threads final first sets 36 ⟩ ≡
  ⟨ allocation space for Visit_graph 34 ⟩;
  std::vector < table_entry *> ::iterator thi = GRAMMAR_DICTIONARY.begin( );
  std::vector < table_entry *> ::iterator thie = GRAMMAR_DICTIONARY.end( );
  for ( ; thi ≠ thie; ++thi) {    /* individually process each thread */
    th_in_stbl*th_tbl=␣(th_in_stbl*)(*thi)->symbol_;
    thread_attributes * th_att = th_tbl→thread_in_stbl( );
    if (th_att→monolithic_ ≡ 'n') {
      yacco2::lrclog ≪ "thread␣being␣walked:␣" ≪ th_att→thread_name_→c_string( )→c_str( ) ≪
          "␣id:␣" ≪ th_att→th_enum_ ≪ std::endl;
      ⟨ initialize Visit_graph to not visited 35 ⟩;
      crt_fset_of_thread(*th_att, th_att→th_enum_, *th_att);
    }
  }
  thi = GRAMMAR_DICTIONARY.begin( );
  thie = GRAMMAR_DICTIONARY.end( );
  for ( ; thi ≠ thie; ++thi) {    /* individually process each thread */
    th_in_stbl*th_tbl=␣(th_in_stbl*)(*thi)->symbol_;
    thread_attributes * th_att = th_tbl→thread_in_stbl( );
    ⟨ initialize Visit_graph to not visited 35 ⟩;
    crt_called_thread_graph(*th_att);
  }
```

This code is used in section 66.

37. Generate document for each grammar's called threads.

38. *crt_called_thread_list* **and** *walk_called_thread_list*.
Generate the call graph per thread for reporting purposes.

```
⟨ accrue linker code 20 ⟩ +≡
  void walk_called_thread_list(std::vector < thread_attributes *> &Thd_list, AST * Mother_thd_t)
  {
    if (Thd_list.begin() ≡ Thd_list.end()) return;
    std::vector < thread_attributes *> ::iterator li = Thd_list.begin();
    std::vector < thread_attributes *> ::iterator lie = Thd_list.end();
    for ( ; li ≠ lie; ++li) {
      thread_attributes * th_att = *li;
      if (Visit_graph[th_att→th_enum_] ≡ 'y') continue;
      Visit_graph[th_att→th_enum_] = 'y';
      AST * called_t = new AST(*th_att);
      AST::add_child_at_end(*Mother_thd_t, *called_t);
      walk_called_thread_list(th_att→list_of_transitive_threads_, called_t);
    }
  }
```

39. *crt_called_thread_graph*.

```
⟨ accrue linker code 20 ⟩ +≡
  void crt_called_thread_graph(thread_attributes & Visited_th)
  {
    if (Visit_graph[Visited_th.th_enum_] ≡ 'y') return;
    Visit_graph[Visited_th.th_enum_] = 'y';
    AST * mother_thd_t = new AST(Visited_th);
    Visited_th.called_thread_graph_ = mother_thd_t;
    walk_called_thread_list(Visited_th.list_of_transitive_threads_, mother_thd_t);
  }
```

40. *gen_each_thread_s_referenced_threads*.

```
⟨ accrue linker code 20 ⟩ +≡
  void gen_each_grammar_s_referenced_threads()
  {
    std::vector < table_entry *> ::iterator thi = GRAMMAR_DICTIONARY.begin();
    std::vector < table_entry *> ::iterator thie = GRAMMAR_DICTIONARY.end();
    for ( ; thi ≠ thie; ++thi) {      /* individually process each thread */
      th_in_stbl*th_tbl=␣(th_in_stbl*)(*thi)->symbol_;
      thread_attributes * th_att = th_tbl→thread_in_stbl();
      ⟨ initialize Visit_graph to not visited 35 ⟩;
      crt_called_thread_graph(*th_att);
    }
  }
```

41. Generate Linker's document.
A secondary graph of called threads per grammar is built so that an overall linker document can be emitted with:
1) index of threads sorted by their name with its fsm's comments
2) followed by the monolithic grammars
3)) each grammar will have its "called threads" graph

42. Make grammar's contents cweaveable and output.

```
⟨ make grammar's contents cweaveable and output 42 ⟩ ≡
  XLATE_SYMBOLS_FOR_cweave(th_att→thread_name_→c_string( )→c_str( ), xlate_gfile);
  XLATE_SYMBOLS_FOR_cweave(th_att→fsm_comments_→c_string( )→c_str( ), rebuild_comment);
  strcat(fandk, xlate_gfile);
  strcat(fandk, "␣---␣");
  strcat(fandk, rebuild_comment);
  int fandk_len = strlen(fandk);
  if (fandk_len < CWEAVE_TITLE_LIMIT) {
    if (fandk[fandk_len − 1] ≠ '.') {
      strcat(fandk, ".");
    }
  }
  else {
    if (fandk_len ≡ CWEAVE_TITLE_LIMIT) {
      if (fandk[CWEAVE_TITLE_LIMIT − 1] ≠ '.') {
        strcat(fandk, ".");
      }
    }
    else {
      fandk[CWEAVE_TITLE_LIMIT] = (char) 0;
      strcat(fandk, "$\\ldots$␣.");
    }
  }
  int x = sprintf(big_buf_, w_grammar, fandk);
  ow_linker_file_.write(big_buf_, x);
  ow_linker_file_ ≪ std::endl;
  x = sprintf(big_buf_, w_comments, rebuild_comment);
  ow_linker_file_.write(big_buf_, x);
  ow_linker_file_ ≪ std::endl;
  x = sprintf(big_buf_, w_called_threads, "␣");
  ow_linker_file_.write(big_buf_, x);
  ow_linker_file_ ≪ std::endl;
```

This code is used in section 47.

43. Output grammar's called threads list.

```
⟨ output grammar's called threads list 43 ⟩ ≡
  prt_called_thread_list_ast_functor prt_functr(&PRINT_CALLED_THREAD_LIST);
  prt_functr.o_file(&ow_linker_file_);
  ast_prefix pre(∗th_att→called_thread_graph_, &prt_functr);
  while (pre.base_stk_.cur_stk_rec( ) ≠ 0) {
    pre.exec( );
  }
```

This code is used in section 47.

44. Output grammar's used threads.

⟨ output grammar's used threads 44 ⟩ ≡

```
std::map < std::string, std::vector < std::string >> ::iterator ti =
    USED_THREADS_LIST.find(th_att→thread_name_→c_string()→c_str());
ow_linker_file_ << "{\\parindent=6pc" << endl;
ow_linker_file_ << "\\item{Used␣threads:}" << std::endl;
KCHARP used_threads = "%s\n""@.%s@>";   /* xref entry */
std::vector < std::string > &tt = ti→second;
std::vector < std::string > ::iterator tti = tt.begin();
std::vector < std::string > ::iterator ttie = tt.end();
char xlate_thnm[Max_cweb_item_size];
for ( ; tti ≠ ttie; ++tti) {
  XLATE_SYMBOLS_FOR_cweave(tti→c_str(), xlate_thnm);
  int x = sprintf(big_buf_, used_threads, xlate_thnm, xlate_thnm);
  ow_linker_file_.write(big_buf_, x);
  ow_linker_file_ << std::endl;
}
if (ti→second.empty() ≡ YES) ow_linker_file_ << "␣none" << endl;
ow_linker_file_ << "}" << endl;
```

This code is used in section 47.

45. Output grammar's first set.

Go thru the set and map the T enum into its literal value.

⟨ output grammar's first set 45 ⟩ ≡

```
ow_linker_file_ << "{\\parindent=6pc" << endl;
ow_linker_file_ << "\\item{First␣set:}" << std::endl;
KCHARP fs = "%s\n""@.%s@>";   /* xref entry */
INT_SET_ITER_type fsi = th_att→fs_.begin();
INT_SET_ITER_type fsie = th_att→fs_.end();
char xlate_tnm[Max_cweb_item_size]; for ( ; fsi ≠ fsie; ++fsi) { int tenum = *fsi;
table_entry *t_entry = T_DICTIONARY[tenum]; tth_in_stbl *t_in_stbl = ( tth_in_stbl * ) t_entry→symbol_;
T_attributes *t_att = t_in_stbl→t_in_stbl();
XLATE_SYMBOLS_FOR_cweave(t_att→fully_qualified_T_name_.c_str(), xlate_tnm);
int x = sprintf(big_buf_, fs, xlate_tnm, xlate_tnm);
ow_linker_file_.write(big_buf_, x);
ow_linker_file_ << std::endl; }
```

This code is used in section 47.

46. Output preamble of document.

⟨output preamble of document 46⟩ ≡

```
KCHARP w_doc_index = "\\input␣\"supp-pdf\"\n""\\input␣\"/usr/local/yacco2/diagrams/o2\
    mac.tex\"\n""\\IDXlinkerdoctitle{%s}{%s}{%s}";
char xlate_file[Max_cweb_item_size];
xlate_file[0] = (char) 0;
char xlate_fscfile[Max_cweb_item_size];
xlate_fscfile[0] = (char) 0;
XLATE_SYMBOLS_FOR_cweave(w_linker_filename_.c_str(), xlate_file);
XLATE_SYMBOLS_FOR_cweave(cntl_file_name.c_str(), xlate_fscfile);
int x = sprintf(big_buf_, w_doc_index, xlate_file, xlate_file, xlate_fscfile);
ow_linker_file_.write(big_buf_, x);
ow_linker_file_ << std::endl;
KCHARP w_doc_comments = "@**␣O2linker␣Index␣of␣Grammars.\\fbreak\n""The␣grammars␣are␣so\
    rted␣lexicographically␣into␣2␣parts:\n""threads␣followed␣by␣the␣stand␣alone␣gra\
    mmars.\n""Each␣grammar's␣called␣threads␣graph␣is␣determined␣from␣thei\
    r␣\n""␣``list-of-transitive-threads''\n""derived␣from␣this␣construct.%s";
x = sprintf(big_buf_, w_doc_comments, "␣");
ow_linker_file_.write(big_buf_, x);
ow_linker_file_ << std::endl;
```

This code is used in section 50.

47. Loop thru grammars to gen their local linker doc info.

⟨loop thru grammars to gen their local linker doc info 47⟩ ≡

```
KCHARP w_grammar = "@*2␣%s";
KCHARP w_comments = "\\Linkeridxentryk{%s}";
KCHARP w_called_threads = "\\Linkercalledthreadstitle%s";
char xlate_gfile[Max_cweb_item_size];
char rebuild_comment[Max_cweb_item_size];
char fandk[Max_cweb_item_size];
char xlate_thnm[Max_cweb_item_size];
char xlate_tnm[Max_cweb_item_size];
std::vector < table_entry *> ::iterator ithi = GRAMMAR_DICTIONARY.begin();
std::vector < table_entry *> ::iterator ithie = GRAMMAR_DICTIONARY.end();
for ( ; ithi ≠ ithie; ++ithi) {      /* individually process each thread */
xlate_gfile[0] = (char) 0;
rebuild_comment[0] = (char) 0;
fandk[0] = (char) 0;
xlate_thnm[0] = (char) 0;
xlate_tnm[0] = (char) 0;
table_entry * tbl_entry = *ithi;
    thread_attributes*th_att=␣((th_in_stbl*)tbl_entry->symbol_)->thread_in_stbl();
⟨make grammar's contents cweaveable and output 42⟩;
⟨output grammar's called threads list 43⟩;
⟨output grammar's first set 45⟩;
⟨output grammar's used threads 44⟩;
}
```

This code is used in section 50.

48. Output First set of linker.

⟨ output First set of linker 48 ⟩ ≡

```
KCHARP w_fsc_file_listing = "@**␣First␣set␣control␣file␣(fsc)␣listin\
    g.\\fbreak\n""File␣:␣‘‘%s’’\\fbreak\n""\\let\\setuplisting\
    hook␣=␣\\relax\n""\\listing{\"%s\"}\n";
char xlated_filename[Max_cweb_item_size];
XLATE_SYMBOLS_FOR_cweave(cntl_file_name.c_str(), xlated_filename);
x = sprintf(big_buf_, w_fsc_file_listing, xlated_filename, cntl_file_name.c_str());
ow_linker_file_.write(big_buf_, x);
ow_linker_file_ << std::endl;
```

This code is used in section 50.

49. Output Index of linker.

⟨ output Index of linker 49 ⟩ ≡

```
KCHARP w_index = "@**␣Index.%s";
x = sprintf(big_buf_, w_index, "␣");
ow_linker_file_.write(big_buf_, x);
ow_linker_file_ << std::endl;
```

This code is used in section 50.

50. Output driver of the linker document.

⟨ generate linker document 50 ⟩ ≡

```
gen_each_grammar_s_referenced_threads();
char big_buf_[BIG_BUFFER_32K];
std::string w_linker_filename_("o2linker_doc.w");
std::ofstream ow_linker_file_;
ow_linker_file_.open(w_linker_filename_.c_str(), ios_base::out | ios::trunc);
⟨ output preamble of document 46 ⟩;
⟨ loop thru grammars to gen their local linker doc info 47 ⟩;
⟨ output First set of linker 48 ⟩;
⟨ output Index of linker 49 ⟩;
ow_linker_file_.close();
```

This code is used in section 66.

51. Emit code.
There is not too much to emit.

1) cpp preamble code — time of day, and the grammar writer's preamble
2) threads include files and their namespace statement
3) global bit maps
4) global thread stable
5) global terminals and their thread bit maps

As an afterthought, the situation of having no threads to emit has been added.

⟨ emit code 51 ⟩ ≡
```
cout ≪ "Emit␣file␣name:␣" ≪ linker_cntl_file_fsm.emitfile_filename_.c_str() ≪ endl;
ofstream ofile(linker_cntl_file_fsm.emitfile_filename_.c_str(), ios::out);
if (¬ofile) {
  cout ≪ "Error␣-␣can't␣open␣emit␣file:␣" ≪ linker_cntl_file_fsm.emitfile_filename_.c_str() ≪ endl;
  return 1;
}
emit_cpp_preamble(ofile, linker_cntl_file_fsm.emitfile_filename_.c_str(),
    linker_cntl_file_fsm.preamble_srce_→syntax_code()→c_str());
emit_global_thread_include_files(ofile);
emit_global_bit_maps(ofile);
if (NO_OF_THREADS ≡ 0) {
  emit_no_threads(ofile);
}
else {
  emit_global_thread_stable(ofile);
  emit_T_fs_of_potential_threads(ofile);
}
ofile.close();
```
This code is used in section 66.

52. Emit no threads.
A situation where the grammar writer has only standalone grammars; there are no parallel statements used within the grammars: ||| ”returned token” called ”thread”.

⟨ accrue linker code 20 ⟩ +≡
```
void emit_no_threads(ofstream &ofile)
{
  ofile ≪ "//␣There␣are␣NO␣THREADS␣emitted" ≪ endl;
  ofile ≪ "void*␣yacco2::THDS_STABLE__␣=␣0;" ≪ endl;
  ofile ≪ "void*␣yacco2::T_ARRAY_HAVING_THD_IDS__␣=␣0;" ≪ endl;
}
```

53. Emit cpp preamble.

⟨ accrue linker code 20 ⟩ +≡

```
void emit_cpp_preamble(ofstream &ofile, const char *OFile, const char *Preamble)
{
  ofile << "//" << endl;
  ofile << "// File: " << OFile << endl;
  ofile << "// Generated by linker.exe" << endl;
  ofile << "// Date and Time: " << DATE_AND_TIME() << endl;
  ofile << "//" << endl;
  ofile << endl;
  ofile << "// Preamble code" << endl;
  ofile << Preamble << endl;
}
```

54. Emit thread include files.

Unfortunately, a lot of verbage to resolve the thread's procedure. OhHum.

Note, standalone grammars are not emitted. Why process them anyway? They provide thread calls that are added to the terminal's thread bit map.

⟨ accrue linker code 20 ⟩ +≡

```
void emit_global_thread_include_files(ofstream &ofile)
{
  ofile << "// thread include and namespace" << std::endl;
  char a[SMALL_BUFFER_4K];
  KCHARP thread_include_ns = "#include \"%s.h\"";
  std::vector < table_entry *> ::iterator thi = GRAMMAR_DICTIONARY.begin();
  std::vector < table_entry *> ::iterator thie = GRAMMAR_DICTIONARY.end();
  for ( ; thi ≠ thie; ++thi) {
    th_in_stbl*th_tbl= (th_in_stbl*)(*thi)->symbol_;
    thread_attributes * th_att = th_tbl→thread_in_stbl();
    if (th_att→monolithic_ ≡ 'y') break;
    int x = sprintf(a, thread_include_ns, th_att→grammar_file_name_→c_string()→c_str());
    ofile.write(a, x);
    ofile << std::endl;
  }
}
```

55. Emit global bit maps.

⟨ accrue linker code 20 ⟩ +≡

```
void emit_global_bit_maps(ofstream &ofile)
{
  ofile << "// BIT MAPS" << std::endl;
  ofile << "#define TOTAL_NO_BIT_WORDS 2*1024*50" << std::endl;
  ofile << "int yacco2::TOTAL_NO_BIT_WORDS__(TOTAL_NO_BIT_WORDS);" << std::endl;
  ofile << "yacco2::ULINT  bit_maps[TOTAL_NO_BIT_WORDS];" << std::endl;
  ofile << "void* yacco2::BIT_MAPS_FOR_SALE__ = (void*)&bit_maps;" << std::endl;
  ofile << "int yacco2::BIT_MAP_IDX__(0);" << std::endl;
}
```

56. Emit global thread stable THDS_STABLE__.
Read the GRAMMAR_DICTIONARY and emit a sorted *Thread_entry* list where each thread (excluded are the standalone grammars) has a global naming convention of Ixxx where the xxx is the thread name. The *Thread_entry* provides its literate name, the thread function to be spawned, and its enumerate value used as an index into the array of threads.

THDS_STABLE__ is a global referenced by Yacco2's runtime library. It is a structure indicating the number of threads within its array and the array of addresses to each just-gened thread's *Thread_entry*. HoHum — is this better than fe-fi-foe-fum I smell the blood of an ...?

```
⟨ accrue linker code 20 ⟩ +≡
  void emit_global_thread_stable(ofstream & ofile)
  {
    ofile ≪ "//␣THREAD␣STABLE" ≪ std::endl;
    char a[BIG_BUFFER_32K];
    KCHARP thread_entry = "yacco2::Thread_entry␣I%s␣=␣{%s,%s,%i,%s::PROC_%s};";
    string quoted_name;
    ⟨ gen thread list 57 ⟩;
    ⟨ gen global thread array 58 ⟩;
  }
```

57. The threading stew.

```
⟨ gen thread list 57 ⟩ ≡
  std::vector < table_entry *> ::iterator thi = GRAMMAR_DICTIONARY.begin();
  std::vector < table_entry *> ::iterator thie = GRAMMAR_DICTIONARY.end();
  for ( ; thi ≠ thie; ++thi) {
    th_in_stbl*th_tbl=␣(th_in_stbl*)(*thi)->symbol_;
    thread_attributes * th_att = th_tbl→thread_in_stbl();
    if (th_att→monolithic_ ≡ 'y') break;
    quoted_name.clear();
    const char *th_name = th_att→thread_name_→c_string()→c_str();
    quoted_name += '"';
    quoted_name += th_name;
    quoted_name += '"';
    int x = sprintf(a, thread_entry, th_name, quoted_name.c_str(), th_att→fully_qualified_th_name_.c_str(),
        th_att→th_enum_, th_att→name_space_name_→c_string()→c_str(),
        th_att→thread_name_→c_string()→c_str());
    ofile.write(a, x);
    ofile ≪ std::endl;
  }
```

This code is used in section 56.

58. The table hôte.
The thoroughbreds waiting for the "and they'rrrre off".

⟨ gen global thread array 58 ⟩ ≡
```
div_t c = div(NO_OF_THREADS, BITS_PER_WORD);
if (c.rem ≠ 0) ++c.quot;
NO_WORDS_FOR_BIT_MAP = c.quot;
KCHARP thread_array = "struct␣thd_array_type␣{\n""␣␣yacco2::USINT␣no_\
    entries__;\n""␣␣yacco2::Thread_entry*␣first_entry__[%i];""};\n""thd_array_type␣thd_\
    array␣=␣{\n""␣%i\n""␣,\n""␣␣{\n";
int x = sprintf(a, thread_array, NO_OF_THREADS, NO_OF_THREADS);
ofile.write(a, x);
ofile ≪ std::endl;
bool first_entry(true);
thi = GRAMMAR_DICTIONARY.begin();
thie = GRAMMAR_DICTIONARY.end();
KCHARP thread_entry_name = "&I%s";
for ( ; thi ≠ thie; ++thi) {
  th_in_stbl*th_tbl=␣(th_in_stbl*)(*thi)->symbol_;
  thread_attributes * th_att = th_tbl→thread_in_stbl();
  if (th_att→monolithic_ ≡ 'y') break;
  if (first_entry ≡ true) {
    first_entry = false;
    ofile ≪ "␣␣␣␣";
  }
  else {
    ofile ≪ "␣␣␣,";
  }
  int x = sprintf(a, thread_entry_name, th_att→thread_name_→c_string()→c_str());
  ofile.write(a, x);
  ofile ≪ std::endl;
}
ofile ≪ "␣␣}\n};" ≪ endl;
⟨ announce the stable to the world 59 ⟩;
```
This code is used in section 56.

59. Announce the stable to the world.

⟨ announce the stable to the world 59 ⟩ ≡
```
ofile ≪ "void*␣yacco2::THDS_STABLE__␣=␣(void*)&thd_array;" ≪ endl;
```
This code is used in section 58.

60. Emit global Terminals' thread bit maps.
This is the inverse to first sets: these are the threads that can run from the specific terminal.

This global optimization determines whether the finite state table has the potential to run a thread. How so? Firstly, the local grammar determines whether threading is taking place in its current state configuration. If so, the current token is checked to see whether there are threads to possibly run using the global thread bit map specific to itself. With these potental threads the local state configuration is measured for activity. Then and only then will the just-in-time dynamics of building the grammar's local thread map occur and the found threads launched.

This optimization stops stuttering: how so? Only threads having the current token in their first set get launched. The jiggles now are only real potential prefixes to parse by each launched thread. Remember, common prefixes get resolved by arbitration within the launching grammar specific to the current finite state configuration.

Why the output to another file? The flatulence of Microsoft's compiler: an INTERNAL COMPILER ERROR "C1001" message. Well I found the typo that causes this draconian behavior: "endl::endl" instead of "std::endl". This congers up speculative thoughts on how Microsoft's compiler is written. Enough of my racket: Back to appending to the same file.

```
⟨ accrue linker code 20 ⟩ +≡
  void emit_T_fs_of_potential_threads(ofstream &ofile)
  {
    ofile ≪ "//␣Terminal␣thread␣sets" ≪ std::endl;
    int no_of_T = T_DICTIONARY.size();
    char a[SMALL_BUFFER_4K];
    KCHARP T_list_to_thd_list_type = "struct␣T_%i_type{\n""␣yacco2::ULINT␣firs\
        t_entry__[%i];\n""};\n";
    KCHARP T_list_to_thd_list_var = "T_%i_type␣T_%i␣=␣{//␣for␣T:␣%s";
    KCHARP thd_id_in_list = "//%i:␣%s";
    INT_SET_LIST_ITER_type i = T_THREAD_ID_LIST.begin();
    INT_SET_LIST_ITER_type ie = T_THREAD_ID_LIST.end();
    int terminal_id(−1);
    for ( ; i ≠ ie; ++i) {
      ++terminal_id;
      INT_SET_type &th_list = *i;
      if (th_list.empty() ≡ true) continue;
      int x = sprintf(a, T_list_to_thd_list_type, terminal_id, NO_WORDS_FOR_BIT_MAP);
      ofile.write(a, x);
      ofile ≪ std::endl;
      ⟨ create terminal's thread bit map 61 ⟩;
    }
    ⟨ emit array of Terminals' thread bit maps 64 ⟩;
  }
```

61. Create terminal's thread bit map.
As the number of threads are unknown, I use `SPECULATIVE_NO_BIT_WORDS` to reserve the space to manufacture the thread maps. See my comments in *Passover on code*.

```
⟨ create terminal's thread bit map 61 ⟩ ≡
  int no_thds_ids = th_list.size();
  table_entry *t_entry = T_DICTIONARY[terminal_id]; tth_in_stbl *t_in_stbl = (tth_in_stbl *) t_entry→symbol_;
  T_attributes *t_att = t_in_stbl→t_in_stbl();
  x = sprintf(a, T_list_to_thd_list_var, terminal_id, terminal_id, t_att→fully_qualified_T_name_.c_str());
  ofile.write(a, x);
  ofile << std::endl;
  ULINT word_map[SPECULATIVE_NO_BIT_WORDS] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
  INT_SET_ITER_type j = th_list.begin();
  INT_SET_ITER_type je = th_list.end();
  ⟨ calculate terminal's thread bit map 62 ⟩;
  ⟨ emit terminal's thread bit map 63 ⟩;
```

This code is used in section 60.

62. Calculate terminal's thread bit map. And print out their contents as comments.

```
⟨ calculate terminal's thread bit map 62 ⟩ ≡
  for ( ; j ≠ je; ++j) {
    int th_id = *j;
    table_entry *tbl_entry = GRAMMAR_DICTIONARY[th_id];
    th_in_stbl*th_tbl= (th_in_stbl*)tbl_entry->symbol_;
    thread_attributes *th_att = th_tbl→thread_in_stbl();
    div_t bb = div(th_id, BITS_PER_WORD);
    ULINT bit_pos_value = 1 << bb.rem;
    word_map[bb.quot] |= bit_pos_value;
    int x = sprintf(a, thd_id_in_list, th_id, th_att→thread_name_→c_string()→c_str());
    ofile.write(a, x);
    ofile << std::endl;
  }
```

This code is used in section 61.

63. Emit terminal's thread bit map.

```
⟨ emit terminal's thread bit map 63 ⟩ ≡
  for (int dd = 1; dd ≤ NO_WORDS_FOR_BIT_MAP; ++dd) {
    if (dd ≡ 1) ofile << " {";
    else ofile << "  ,";
    ofile << word_map[dd − 1] << endl;
  }
  ofile << " }" << endl;
  ofile << "};" << endl;
```

This code is used in section 61.

64. Emit Terminals' thread bit maps and global T_ARRAY_HAVING_THD_IDS__.
Go tell it to the ? or is it Yacco2?

```
⟨ emit array of Terminals' thread bit maps 64 ⟩ ≡
  KCHARP T_array_type = "struct␣t_array_type␣{\n""␣␣yacco2::USINT␣no_\
      entries__;\n""␣␣yacco2::thd_ids_having_T*␣first_entry__[%i];\n""};";
  int x = sprintf(a, T_array_type, no_of_T);
  ofile.write(a, x);
  ofile ≪ std::endl;
  KCHARP T_array = "t_array_type␣t_array␣=␣{\n""␣␣%i\n""␣␣,{";
  x = sprintf(a, T_array, no_of_T);
  ofile.write(a, x);
  ofile ≪ std::endl;
  i = T_THREAD_ID_LIST.begin();
  ie = T_THREAD_ID_LIST.end();
  ⟨ print out each thread set 65 ⟩;
  ofile ≪ "␣␣␣}\n};" ≪ endl;
  ofile ≪ "void*␣yacco2::T_ARRAY_HAVING_THD_IDS__␣=␣(void*)&t_array;" ≪ endl;
```

This code is used in section 60.

65. Print each entry.
More coughing. Oh well.

```
⟨ print out each thread set 65 ⟩ ≡
  bool first_item(true);    /* regulates if a comma should be emitted */
  for (int t_id = −1; i ≠ ie; ++i) { ++t_id;
  table_entry *t_entry = T_DICTIONARY[t_id];
  tth_in_stbl *t_in_stbl = ( tth_in_stbl * ) t_entry→symbol_;
  T_attributes *t_att = t_in_stbl→t_in_stbl();
  INT_SET_type &th_list = *i;
  if (th_list.empty() ≡ true) {    /* no thds with this T as first set */
    if (first_item ≡ true) first_item = false;
    else ofile ≪ "␣␣␣␣,";
    KCHARP T_array_entries = "%s//␣%s";
    int x = sprintf(a, T_array_entries, "0", t_att→fully_qualified_T_name_.c_str());
    ofile.write(a, x);
    ofile ≪ std::endl;
    continue;
  }
  else {
    if (first_item ≡ true) first_item = false;
    else ofile ≪ "␣␣␣␣,";
    KCHARP T_list_to_thd_list_var = "(yacco2::thd_ids_having_T*)&T_%i␣//␣%s";
    int x = sprintf(a, T_list_to_thd_list_var, t_id, t_att→fully_qualified_T_name_.c_str());
    ofile.write(a, x);
    ofile ≪ std::endl;
  }
  }
```

This code is used in section 64.

66. Main line of Linker.

```
⟨ accrue linker code 20 ⟩ +≡
  YACCO2_define_trace_variables( );
  yacco2::TOKEN_GAGGLE Error_queue;
  STBL_T_ITEMS_type STBL_T_ITEMS;
  std::vector < NS_yacco2_terminals::table_entry *> GRAMMAR_DICTIONARY;
  std::map < std::string, std::vector < std::string >> USED_THREADS_LIST;
  std::vector < NS_yacco2_terminals::table_entry *> T_DICTIONARY;
  INT_SET_LIST_type T_THREAD_ID_LIST;
  int NO_OF_THREADS(0);
  int NO_WORDS_FOR_BIT_MAP(0);
  char Visit_graph[RESERVE_FIXED_NO_THREADS];
  extern void XLATE_SYMBOLS_FOR_cweave(const char *Sym_to_xlate, char *Xlated_sym);
  extern void PRINT_CALLED_THREAD_LIST(yacco2::AST * Node, std::ofstream * Ow_linker_file, int
      Recursion_level);
  string cntl_file_name;
  yacco2::CHARPRT_SW('n');
  int main(int argc, char *argv[ ])
  {
    cout << yacco2::O2linker_VERSION << std::endl;
    using namespace yacco2;
    using namespace std;
    load_linkkws_into_tbl( );
    cout << "Get␣command␣line␣and␣parse␣it␣" << endl;
    GET_CMD_LINE(argc, argv, Linker_holding_file, Error_queue);
    ⟨ if error queue not empty then deal with posted errors 15 ⟩;
    LINKER_PARSE_CMD_LINE(Linker_holding_file, cntl_file_name, Error_queue);
    ⟨ if error queue not empty then deal with posted errors 15 ⟩;
    lrclog << yacco2::O2linker_VERSION << std::endl;
    ⟨ parse linker control file 17 ⟩;
    ⟨ parse T alphabet 18 ⟩;
    ⟨ parse fsc files 19 ⟩;
    ⟨ post verify that there are no threads "used" and not "defined" 21 ⟩;
    ⟨ sort thread dictionary 23 ⟩;
    ⟨ dump sorted dictionary 24 ⟩;
    ⟨ count and re-align threads enumerate values to sorted position 25 ⟩;
    ⟨ check whether Linker has enough space to gen thread bit maps: no throw up 26 ⟩;
    ⟨ generate threads final first sets 36 ⟩;
    ⟨ emit code 51 ⟩;
    ⟨ generate linker document 50 ⟩;    /* yacco2::Parallel_threads_shutdown(linker_cntl_file); */
    return 0;
  }
```

67. Structure implementation.

68. *prt_called_thread_list_ast_functor* **implementation.**

⟨ Structure implementations 68 ⟩ ≡

```
extern void PRINT_CALLED_THREAD_LIST(AST * Node, std::ofstream * Ow_linker_file, int Idx){ char
      big_buf_[BIG_BUFFER_32K]; thread_attributes * ta = ( thread_attributes * ) AST::content(*Node);
   KCHARP w_called_threads = "\\Linkercalledthreads{%s}{%i}";
   char xlate_gfile[Max_cweb_item_size];
   XLATE_SYMBOLS_FOR_cweave(ta->thread_name_->c_string( )->c_str( ), xlate_gfile);
   int x = sprintf(big_buf_, w_called_threads, xlate_gfile, Idx);
   (*Ow_linker_file).write(big_buf_, x);
   (*Ow_linker_file) << endl;
   KCHARP w_called_threads_index = "@.%s@>";
   x = sprintf(big_buf_, w_called_threads_index, xlate_gfile);
   (*Ow_linker_file).write(big_buf_, x);
   (*Ow_linker_file) << endl; }
   yacco2::functor_result_type
   prt_called_thread_list_ast_functor::operator( )(yacco2::ast_base_stack * Stk_env)
   {
     stk_env_ = Stk_env;
     srec_ = stk_env_->cur_stk_rec_;
     idx_ = stk_env_->idx_;
     yacco2::INT pidx = idx_ - 1;
     cnode_ = srec_->node_;
     if (pidx <= 0) goto prt_prefix;
     {
       ast_base_stack::s_rec * psrec = stk_env_->stk_rec(pidx);
     }
   prt_prefix:
     ⟨ acquire trace mu 69 ⟩;
     yacco2::INT no_lt(0);
     for (yacco2::INT x = 0; x <= idx_; ++x)
       if (stk_env_->stk_rec(x)->act_ == ast_base_stack::left) ++no_lt;
     ⟨ release trace mu 70 ⟩;
   call_prt_func:
     (*prt_funct_)(cnode_, ow_linker_file_, no_lt + 1);
     return accept_node;   /* continue looping thru ast */
   }
  prt_called_thread_list_ast_functor::prt_called_thread_list_ast_functor(PFF Func):
        prt_funct_(Func), cnt_(0), ow_linker_file_(0)
   { }
   void prt_called_thread_list_ast_functor::reset_cnt( )
   {
     cnt_ = 0;
   }
   void prt_called_thread_list_ast_functor::o_file(std::ofstream * Ow_linker_file)
   {
     ow_linker_file_ = Ow_linker_file;
   }
```

This code is used in section 71.

69. Acquire trace mu.
Used to serialize trace output. Sometimes the traced output is skewed due to the threading. The output to a global container is not thread safe, so make it by use of a mutex.

```
⟨ acquire trace mu 69 ⟩ ≡
  LOCK_MUTEX(yacco2::TRACE_MU);
  if (yacco2::YACCO2_MU_TRACING__) {
    yacco2::lrclog ≪ "YACCO2_MU_TRACING__::Acquired␣trace␣mu" ≪ std::endl;
  }
```

This code is used in section 68.

70. Release trace mu.
Used to serialize trace output.

```
⟨ release trace mu 70 ⟩ ≡
  if (yacco2::YACCO2_MU_TRACING__) {
    yacco2::lrclog ≪ "YACCO2_MU_TRACING__::Releasing␣trace␣mu" ≪ std::endl;
  }
  UNLOCK_MUTEX(yacco2::TRACE_MU);
```

This code is used in section 68.

71. Write out *o2linker_defs.cpp* Structure implementations.

```
⟨ o2linker_defs.cpp 71 ⟩ ≡
#include "o2linker.h"
  ⟨ Structure implementations 68 ⟩;
```

72. PMS — Post meta syndrome.
Post thoughts of improvement before using Linker.

73. What happens when there are no threads to produce?.
Deal with iT? Sorry for the yelling...haha This is a legitimate situation.

So `THDS_STABLE__` and `T_ARRAY_HAVING_THD_IDS__` have NULL pointers emitted. This situation does not have to be checked for in Yacco2 parse library as there are no threads in the local state configuration to be tried. Yacco2 parsing checks to see if the ||| symbol is present in the current state configuration before trying to parallel parse.
22 April 2005

74. Passover on code.
Spring cleaning so put in those last gasp constraints. I know C++'s containers could be used to generate bit maps but... Enough of my rantings and mistrust on others software. So there's a hardcoded number of threads supported. The number of threads supported is `SPECULATIVE_NO_BIT_WORDS` * `BITS_PER_WORD`. Under current definitions this works out to 20*32 threads that can be gened. A bit of overkill but u never know who's out there. Now u can rant at me if this limit is surpassed. I would really like to know what u're doing to exceed this speed bump: Could be an interesting conversation.
24 Apr. 2005

75. Add test suites.
25 Apr. 2005

76. Bugs — ugh. Do i make them? Sure do! They have all the makings of Darwinism with sloppiness due to work overload and not enough self evaluation ... Forget the platitudes Dude. Hey it's Dave. The testsuite raised 3 basic programming misconceptions that should enlighten u the reader in your use of Yacco2.

1) posting of errors within a thread versus a standalone grammar
2) off by xxx in error token co-ordinates
3) refining of lookahead expression for common prefix recognition

77. Posting of errors within a grammar.
There are 2 ways to post an error within a grammar:

1) `ADD_TOKEN_TO_ERROR_QUEUE` macro or parser's *add_token_to_error_queue* procedure
2) `RSVP` macro to post the error or parser's *add_token_to_producer* procedure

The `ADD_TOKEN_TO_ERROR_QUEUE` route should be used only within a standalone grammar. Why? Due to parallelism (nondeterminism), threads can misfire but the overall parsing is fine because some thread will accept its parse or the calling grammar that set things in motion also uses the conditional parsing features of shifting over reduce.

Putting an error token into the *accept queue* via `RSVP` allows the calling grammar to discriminate by arbitration as to what token to accept. Then within the grammar's subrule that handles the error token, the syntax-directed code would post the accepted error as an error and possbily shutdown the parse by parser's *set_stop_parse*(*true*) routine. This allows the grammar writer more flexibitity in error handling.

So the moral of this story is "know your parsing run context" when posting of errors. So where's my mistake? I used the `RSVP` facility within a standalone grammar. This just shunts the error token into the output token queue instead of the error queue.

78. Off by xxx in error token co-ordinates.
When posting an error, the newly created error token needs to be associated with the source file position. So how is this accomplished? Typically one uses a previous token with established co-ordinates to set the error token's co-ordinates via the parser's *set_rc* routine.

So why the off by xxx syndrone? What co-ordinate do u associate an error with when the grammar's subrule uses the wild card facility |+| to catch an error. Is it the *current_token* routine? Not really. Depending on the syntax-directed code context, this can be the lookahead token after the shift operation. That is, it is one past the currently shifted token on the parse stack represented symbolicly by |+|. So use the appropriate *Sub_rule_xxx.p1__* parameter in the syntax-directed procedure to reference the stacked token where xxx is the subrule number of this subrule. *p1__* is the first component of the subrule's right-hand-side expression. If there are other components that are to be referenced, *px__* will have the appropriate component number replacing x.

There are other facilities open to the grammar writer like the *start_token* routine that provides the starting token passed to the thread for parsing for co-ordinate references. One can also store token references within the grammar rules as parsing takes place so that a "roll your own" fiddling within contexts can be programmed: To each their own.

79. Refining of lookahead expression for keyword recognition.
I describe this problem using a concrete example but the problem is generic when there are competing threads that can have common prefixes. If a keyword is "emitfile" then should it accept "emitfilex" when the balance "x" follows? No it should not. This is the common prefix problem where the lookahead boundary to accept or abort the parse was wrongly programmed. As theads can fine tune their lookahead expression by the grammar's "parallel-la-boundary" statement, the correction is to properly program this expression. See the *yacco2_linker_keywords.lex* grammar for the proper lookahead expression to abort such a parse.

80. Take 2: Teaching the teacher — why off by xxx in error co-ordinates?.
In testing linker's grammars, the following came up. What co-ordinate does one associate with an error token when the "eog" token produces the error? Remember, the meta-terminals like eog, |.|, | + | have no co-ordinates associated with them as they are shared across all token containers.

Quick review of co-ordinates: There are 4 parts to a token's co-ordinates:

1) a file number kept by Yacco2 giving the filename source of token
2) line number within the file
3) character position on the line
4) character position within the physical file returned by the I/O routine

Points 1 − 3 are used for now to print out errors.

Well I had to check the source code of *set_rc* procedure. It was programmed properly and had comments to boot describing the problem. *set_rc* marches back thru the token container looking for a token having physical co-ordinates. Call the number of moves thru the container the displacement. This displacement is summed with the character position of the referenced token to point to a spot on the source line. The other co-ordinate attributes are just copied. Depending on the token's length in display characters, the displacement will be xxx moves to the right of the referenced token's co-ordinates. This is why the error token graphic ↑ is offset within the error message displaying against the source line and character spot. Depending on the length of the token in display characters, the offset could be within the interior of the token or to the right of it. This offset shows that a potential overflow was prevented when the "eog" token halts the parse. Have a look at testsuite's "no end-T-alphabet keyword" or the last test "end-list-of-transitive-threads keyword not present " illustrating this type of situation.

What about the other meta-terminals? This is a rool-your-own-laddie. Grab whatever context u want to associate the co-ordinates against the error token. In the case of the |.|, the meta-epsilon token, u can use the *current_token*( ) as its co-ordinate context.

Well go tell it to the Yacco2 parse library. See *set_rc* writeup. The moral of this tidbit? A user manual must be written. I know but this will be my next venture.
4 May 2005.

81. Missed associating the root node thread to its called threads' terminals.
This shows how the forest and the trees (pun intended) got blurred. The scoping of a screen just does not allow one to view well a general perspective: batteries not required nor a camel to carry paper and a café au lait.

Now the 2 errors:
The *visit_graph* reserved the right amount of space but I mistakenly thought that the container's size procedure gave the number of elements reserved — nope. So the *visit_graph*'s check on visited became true immediately.

2nd mistake had a compound logic error: the root node thread was not associated with the called threads' first sets — which is the raison d'être for all this recursion. The more subtle error was checking in the global terminal's set for the called thread existance. If it was present, the root node thread was bypassed in checking whether it should be entered into the global terminal set. Boy sometimes you're dump Dave — spelling correct at the time: haha.

This error should have been caught in the test suites. It requires a proper set of faked grammars to test out transitive closure: i.e., what about nested calling grammars?, various flavors of epsilon: a complete pass thru grammar, |.| to boundary detection, etc.

Plain and simple I was procastinating as real work on other things is building. The scaffolding for these other suites requires a bit more thought and much more effort. Ahhh self analysis watching the watched watching — I leave this to Freud but it still cost 3 hours of time lost to ferret out these bloopers.

Okay I'm crafting more test suites to cruise-control these esoteric conditions. It demonstrates the requirement of well calibrated test suites as a necessary dimension to programming along with comments as in literate programming, and pre / post constraint declarations within code to catch realtime strange-ites. Without these tested dimensions, what assurances or confidence does one have to say that the code is correct? Not much but hand wavings and hot ... Let's hear it for QA and its exercising regime to come out and strut itself.
11 May 2005.

82. Dynamic reserve space for *Visit_graph*.
The **std** :: *vector* < **char** > *Visit_graph*( ); was originally calculating the number of threads before reserving space via "reserve(xxx)" procedure. This is a simple way to associate a visited node for my recursive graph walks: stop those revisits. Somehow this space was getting reallocated and destroying my trees. So out damm dynamics and in with static array no template please...
Nov. 2007

*

83. Test suites: Check out those flabby grammar muscles.
Exercise, exercise, excerise, perhaps to ... Linker's languages. Microsoft's batch facility is used to do the sweating. The test suites are inputted via the command line option on Linker.

The batch file *linker_testsuite.bat* contains the all the test suites. Here are the test results with some editorial liberties:

```
C:\yacco2\linker>rem file: testsuite.bat
C:\yacco2\linker>rem test suite for Linker
C:\yacco2\linker>cd "c:\yacco2\linker\release\"
######## Command line edits #######
######## perfect score: compile linker and yacco2 first set
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/compiler/grammars/yacco2.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Sort thread dictionary
Emit file name: c:/yacco2/compiler/grammars/yacco2_fsc.cpp

######## Command line error: bad file name inputted
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/compiler/grammars/yacco2.fscc"
Load linker's keywords
Get command line and parse it
Error in file#: 1 "linkercmd.tmp"
c:/yacco2/compiler/grammars/yacco2.fscc
^
        fpos: 0 line#: 1 cpos: 1
        bad-filename filename: "c:/yacco2/compiler/grammars/yacco2.fscc" does not exist

######## File control file error: no preamble construct
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_1.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Error in file#: 2 "c:/yacco2/linker/TS_1.fsc"
emitfile "TS_1_fsc.tmp"
                        ^
        fpos: 121 line#: 7 cpos: 25
        preamble keyword not present

######## File control file error: no end-preamble present
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_2.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Error in file#: 2 "c:/yacco2/linker/TS_2.fsc"
#include <yacco2.h>
                   ^
        fpos: 151 line#: 9 cpos: 20
        end-preamble keyword not present
```

```
######## File control file error: no preamble code present
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_3.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Error in file#: 2 "c:/yacco2/linker/TS_3.fsc"
preamble
        ^
        fpos: 131 line#: 8 cpos: 9
        preamble source code not present

######## File control file error: no file-of-T-alphabet keyword present
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_4.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Error in file#: 2 "c:/yacco2/linker/TS_4.fsc"
file-of-T-alphabe "c:/yacco2/linker/TS_4.fsc"
^
        fpos: 57 line#: 6 cpos: 1
        file-of-T-alphabet keyword not present

######## File control file error: no file-of-T-alphabet file present
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_5.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Error in file#: 2 "c:/yacco2/linker/TS_5.fsc"
emitfile "TS_5_fsc.tmp"
^
        fpos: 82 line#: 7 cpos: 1
        T-alphabet file not present

######## File control file error: no emitfile keyword present
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_6.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Error in file#: 2 "c:/yacco2/linker/TS_6.fsc"
emitfilee "TS_6_fsc.tmp"
^
        fpos: 102 line#: 7 cpos: 1
        emitfile keyword not present

######## File control file error: no emitfile file present
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_7.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Error in file#: 2 "c:/yacco2/linker/TS_7.fsc"
preamble
^
```

```
        fpos: 108 line#: 8 cpos: 1
        emitfile file not present

######## File control file error: no fsc file present
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_8.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Error in file#: 2 "c:/yacco2/linker/TS_8.fsc"
end-preamble
            ^
        fpos: 175 line#: 10 cpos: 13
        fsc control file not present

######## File control file error: bad fsc filename
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_9.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Error in file#: 2 "c:/yacco2/linker/TS_9.fsc"
"c:/yacco2/linker/TS_9xx.fsc"
^
        fpos: 174 line#: 11 cpos: 1
        fsc control file does not exist

######## T-alphabet edits #######
######## T-alphabet: no T-alphabet keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_10.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Error in file#: 3 "c:/yacco2/linker/TS_10t1.fsc"
T-alphabett
^
        fpos: 75 line#: 5 cpos: 1
        T-alphabet keyword not present

######## T-alphabet: no end-T-alphabet keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_11.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Error in file#: 3 "c:/yacco2/linker/TS_11t1.fsc"
end-T-alphabett
  ^
        fpos: 125 line#: 10 cpos: 3
        end-T-alphabet keyword not present

######## T-alphabet: duplicate t definition
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_12.fsc"
```

```
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Error in file#: 3 "c:/yacco2/linker/TS_12t1.fsc"
LR1_eof
^
        fpos: 122 line#: 10 cpos: 1
        dup-entry-in-sym-table

######## T-alphabet: no t definition
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_13.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Error in file#: 3 "c:/yacco2/linker/TS_13t1.fsc"
end-T-alphabet
^
        fpos: 68 line#: 6 cpos: 1
        no terminals in T-alphabet list

######## T-alphabet: comment overrun
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_14.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Error in file#: 2 "c:/yacco2/linker/TS_14.fsc"
/*
^
        fpos: 0 line#: 1 cpos: 1
        comment-overrun

######## fsc control files #######
######## fsc control files: not defined terminal used in fsc control file

C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_15.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/compiler/grammars/error_symbols_phrase.FSC"
no-of-T      513
             ^
        fpos: 195 line#: 7 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 4 "c:/yacco2/compiler/grammars/error_symbols_phrase.FSC"
   LR1_fset_transience_operator
   ^
        fpos: 239 line#: 9 cpos: 4
        bad terminal in list, not defined in T-alphabet
```

```
######## fsc control files: thread used but not defined in fsc control file
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_16.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_16t1.fsc"
  NS_bad_char_set::TH_bad_char_set
  ^
        fpos: 378 line#: 18 cpos: 3
        bad thread in transitive list, not defined by fsc files
Error in file#: 4 "c:/yacco2/linker/TS_16t1.fsc"
  NS_c_comments::TH_c_comments
  ^
        fpos: 413 line#: 19 cpos: 3
        bad thread in transitive list, not defined by fsc files
Error in file#: 4 "c:/yacco2/linker/TS_16t1.fsc"
  NS_c_literal::TH_c_literal
  ^
        fpos: 444 line#: 20 cpos: 3
        bad thread in transitive list, not defined by fsc files
Error in file#: 4 "c:/yacco2/linker/TS_16t1.fsc"
  NS_c_string::TH_c_string
  ^
        fpos: 473 line#: 21 cpos: 3
        bad thread in transitive list, not defined by fsc files
Error in file#: 4 "c:/yacco2/linker/TS_16t1.fsc"
  NS_eol::TH_eol
  ^
        fpos: 500 line#: 22 cpos: 3
        bad thread in transitive list, not defined by fsc files
Error in file#: 4 "c:/yacco2/linker/TS_16t1.fsc"
  NS_identifier::TH_identifier
  ^
        fpos: 517 line#: 23 cpos: 3
        bad thread in transitive list, not defined by fsc files
Error in file#: 4 "c:/yacco2/linker/TS_16t1.fsc"
  NS_ws::TH_ws
  ^
        fpos: 548 line#: 24 cpos: 3
        bad thread in transitive list, not defined by fsc files

######## fsc control files: bad native first set number
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_17.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_17t1.fsc"
```

```
no-of-T      502
             ^
        fpos: 205 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 4 "c:/yacco2/linker/TS_17t1.fsc"
list-of-native-first-set-terminals -3
                                   ^
        fpos: 244 line#: 12 cpos: 36
        list-of-native-terminals... number not present

######## fsc control files: native first set number mismatch to no in list
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_18.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_18t1.fsc"
no-of-T      502
             ^
        fpos: 229 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 4 "c:/yacco2/linker/TS_18t1.fsc"
list-of-native-first-set-terminals 2
                                   ^
        fpos: 268 line#: 12 cpos: 36
        no terminals in list not equal, chk items in list

######## fsc control files: thread list number mismatch to no in list
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_19.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_19t1.fsc"
no-of-T      502
             ^
        fpos: 224 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 4 "c:/yacco2/linker/TS_19t1.fsc"
list-of-transitive-threads 2
                           ^
        fpos: 383 line#: 17 cpos: 28
        no threads in list not equal, chk items in list

######## fsc control files: bad thread list number
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_20.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
```

```
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_20t1.fsc"
no-of-T      502
             ^
        fpos: 205 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 4 "c:/yacco2/linker/TS_20t1.fsc"
list-of-transitive-threads -7
                           ^
        fpos: 364 line#: 17 cpos: 28
        list-of-transitive-threads... number not present

######## fsc control files: no thread list number present
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_21.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_21t1.fsc"
no-of-T      502
             ^
        fpos: 212 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 4 "c:/yacco2/linker/TS_21t1.fsc"
  NS_bad_char_set::TH_bad_char_set
  ^
        fpos: 374 line#: 18 cpos: 3
        list-of-transitive-threads... number not present

######## fsc control files: no native first set number present
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_22.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_22t1.fsc"
no-of-T      502
             ^
        fpos: 222 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 3 "c:/yacco2/compiler/grammars/yacco2_T_enumeration.fsc"
raw_at_sign
^
        fpos: 746 line#: 74 cpos: 1
        list-of-native-terminals... number not present

######## fsc control files: no-of-T # not matched against T-alphabet
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_23.fsc"
Load linker's keywords
Get command line and parse it
```

```
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_23t1.fsc"
no-of-T      1502
             ^
        fpos: 223 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar

######## fsc control files: no transitive keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_24.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_24t1.fsc"
transitivee   y
^
        fpos: 69 line#: 5 cpos: 1
        transitive keyword not present

######## fsc control files: bad transitive value
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_25.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_25t1.fsc"
transitive   x
             ^
        fpos: 81 line#: 5 cpos: 14
        linker's transitive value not n or y

######## fsc control files: bad grammar-name keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_26.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_26t1.fsc"
grammar-namee "pass3"
^
        fpos: 87 line#: 6 cpos: 1
        grammar-name keyword not present

######## fsc control files: bad grammar-name value
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_27.fsc"
Load linker's keywords
Get command line and parse it
```

```
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_27t1.fsc"
grammar-name pass3
             ^
        fpos: 98 line#: 6 cpos: 14
        grammar-name value not present or quoted value

######## fsc control files: bad name-space keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_28.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_28t1.fsc"
name-spacee   "NS_pass3"
^
        fpos: 106 line#: 7 cpos: 1
        name-space keyword not present

######## fsc control files: bad name-space value
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_29.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_29t1.fsc"
name-space   NS_pass3
             ^
        fpos: 117 line#: 7 cpos: 14
        name-space value not present or quoted value

######## fsc control files: bad thread-name keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_30.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_30t1.fsc"
thread-namee  "Cpass3"
^
        fpos: 131 line#: 8 cpos: 1
        thread-name keyword not present

######## fsc control files: bad thread-name value
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_31.fsc"
Load linker's keywords
Get command line and parse it
```

```
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_31t1.fsc"
thread-name  Cpass3
             ^
        fpos: 142 line#: 8 cpos: 14
        thread-name value not present or quoted value

######## fsc control files: bad monolithic keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_32.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_32t1.fsc"
monolithicc   y
^
        fpos: 152 line#: 9 cpos: 1
        monolithic keyword not present

######## fsc control files: bad monolithic value
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_33.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_33t1.fsc"
monolithic   Y
             ^
        fpos: 163 line#: 9 cpos: 14
        linker's monolithic value not n or y

######## fsc control files: bad file-name keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_34.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_34t1.fsc"
file-namee    "pass3.fsc"
^
        fpos: 166 line#: 10 cpos: 1
        file-name keyword not present

######## fsc control files: bad file-name value
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_35.fsc"
Load linker's keywords
Get command line and parse it
```

```
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_35t1.fsc"
file-name    monolithic.fsc
             ^
        fpos: 177 line#: 10 cpos: 14
        file-name value not present or quoted value

######## fsc control files: bad no-of-T keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_36.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_36t1.fsc"
no-of-Tt      504
^
        fpos: 189 line#: 11 cpos: 1
        no-of-T keyword not present

######## fsc control files: bad no-of-T value
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_37.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_37t1.fsc"
no-of-T      +504
             ^
        fpos: 200 line#: 11 cpos: 14
        no-of-T value not present

######## fsc control files: bad list-of-native-first-set-terminals keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_38.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_38t1.fsc"
no-of-T      504
             ^
        fpos: 229 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 4 "c:/yacco2/linker/TS_38t1.fsc"
transitive 3
^
        fpos: 233 line#: 12 cpos: 1
        list-of-native-terminals keyword not present
```

```
######## fsc control files: bad end-list-of-native-first-set-terminals keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_39.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_39t1.fsc"
no-of-T      504
             ^
        fpos: 233 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 4 "c:/yacco2/linker/TS_39t1.fsc"
end-list-of-native-first-set-terminalss
^
        fpos: 326 line#: 16 cpos: 1
        bad terminal in list, not defined in T-alphabet

######## fsc control files: bad list-of-transitive-threads keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_40.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_40t1.fsc"
no-of-T      504
             ^
        fpos: 221 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 4 "c:/yacco2/linker/TS_40t1.fsc"
list-of-transitive-threadss 0
^
        fpos: 353 line#: 17 cpos: 1
        list-of-transitive-threads keyword not present

######## fsc control files: bad end-list-of-transitive-threads keyword
C:\yacco2\linker\Release>lnk.exe "c:/yacco2/linker/TS_41.fsc"
Load linker's keywords
Get command line and parse it
Parse linker control file
Parse alphabet
Parse fsc files
Error in file#: 4 "c:/yacco2/linker/TS_41t1.fsc"
no-of-T      504
             ^
        fpos: 225 line#: 11 cpos: 14
        T-alphabet list vs  no. of T not eq. re-compile grammar
Error in file#: 4 "c:/yacco2/linker/TS_41t1.fsc"
end-list-of-transitive-threadss
  ^
```

```
      fpos: 387 line#: 18 cpos: 3
      end-list-of-transitive-threads keyword not present

C:\yacco2\linker\Release>del c:\yacco2\linker\ts*.tmp
```

84. Sample output from Linker.
A sampling from Yacco2's grammars. Have a look in the source code below where the following external variables are defined at line references 41, 43, 44, 72, and 193:

yacco2 :: `TOTAL_NO_BIT_WORDS__`
yacco2 :: `BIT_MAPS_FOR_SALE__`
yacco2 :: `BIT_MAP_IDX__`
yacco2 :: `THDS_STABLE__`
yacco2 :: `T_ARRAY_HAVING_THD_IDS__`

Yacoo2's parse library references them and they get resolved by the language linker.

```
 1:  //
 2:  // File: c:/yacco2/compiler/grammars/yacco2_fsc.cpp
 3:  // Generated by linker.exe
 4:  // Date and Time: Fri May 06 16:30:11 2005
 5:  //
 6:
 7:  // Preamble code
 8:  #include <yacco2.h>
 9:  #include <yacco2_T_enumeration.h>
10:  #include <yacco2_err_symbols.h>
11:  #include <yacco2_k_symbols.h>
12:  #include <yacco2_terminals.h>
13:  #include <yacco2_characters.h>
14:  using namespace NS_yacco2_T_enum;
15:  using namespace NS_yacco2_err_symbols;
16:  using namespace NS_yacco2_k_symbols;
17:  using namespace NS_yacco2_terminals;
18:  using namespace NS_yacco2_characters;
19:  // thread include and namespace
20:  #include <T_enum_phrase_th.h>
21:  using namespace NS_T_enum_phrase_th;
22:  #include <angled_string.h>
23:  using namespace NS_angled_string;
24:  #include <bad_char_set.h>
25:  using namespace NS_bad_char_set;
26:  #include <c_comments.h>
27:  using namespace NS_c_comments;
28:  #include <c_literal.h>
29:  using namespace NS_c_literal;
30:       ...
31:  #include <yacco2_code_end.h>
32:  using namespace NS_yacco2_code_end;
33:  #include <yacco2_lcl_option.h>
34:  using namespace NS_yacco2_lcl_option;
35:  #include <yacco2_linker_keywords.h>
36:  using namespace NS_yacco2_linker_keywords;
37:  #include <yacco2_syntax_code.h>
38:  using namespace NS_yacco2_syntax_code;
39:
40:  // BIT MAPS
41:  #define TOTAL_NO_BIT_WORDS 2*1024*50
42:  int yacco2::TOTAL_NO_BIT_WORDS__(TOTAL_NO_BIT_WORDS);
43:  yacco2::ULINT  bit_maps[TOTAL_NO_BIT_WORDS];
```

```
44:  void* yacco2::BIT_MAPS_FOR_SALE__ = (void*)&bit_maps;
45:  int yacco2::BIT_MAP_IDX__(0);
46:  // THREAD STABLE
47:  yacco2::Thread_entry ITH_T_enum_phrase_th =
48:      {"TH_T_enum_phrase_th",NS_T_enum_phrase_th::TH_T_enum_phrase_th,0
49:      ,NS_T_enum_phrase_th::PROC_TH_T_enum_phrase_th};
50:  yacco2::Thread_entry ITH_angled_string =
51:      {"TH_angled_string",NS_angled_string::PROC_TH_angled_string,1
52:      ,NS_angled_string::TH_angled_string};
53:                                  ...
54:  yacco2::Thread_entry ITH_yacco2_syntax_code =
55:      {"TH_yacco2_syntax_code",NS_yacco2_syntax_code::TH_yacco2_syntax_code
56:      ,52,NS_yacco2_syntax_code::PROC_TH_yacco2_syntax_code};
57:  struct thd_array_type {
58:    yacco2::USINT no_entries__;
59:    yacco2::Thread_entry* first_entry__[53];
60:  };
61:  thd_array_type thd_array = {
62:   53
63:   ,
64:    {
65:      &ITH_T_enum_phrase_th
66:     ,&ITH_angled_string
67:     ,&ITH_bad_char_set
68:     ,&ITH_c_comments
69:             ...
70:     ,&ITH_yacco2_code_end
71:     ,&ITH_yacco2_lcl_option
72:     ,&ITH_yacco2_linker_keywords
73:     ,&ITH_yacco2_syntax_code
74:    }
75:  };
76:  void* yacco2::THDS_STABLE__ = (void*)&thd_array;
77:  // Terminal thread sets
78:  struct T_0_type{
79:   yacco2::ULINT first_entry__[2];
80:  };
81:  T_0_type T_0 = {// for T: LR1_eof
82:  //20: TH_linker_preamble_code
83:  //31: TH_rhs_bnd
84:  //32: TH_rhs_component
85:  //47: TH_unquoted_string
86:  //52: TH_yacco2_syntax_code
87:   {2148532224
88:    ,1081345
89:   }
90:  };
91:  struct T_1_type{
92:   yacco2::ULINT first_entry__[2];
93:  };
94:  T_1_type T_1 = {// for T: LR1_eog
95:  //20: TH_linker_preamble_code
```

```
 96:  //31: TH_rhs_bnd
 97:  //32: TH_rhs_component
 98:  //47: TH_unquoted_string
 99:  //52: TH_yacco2_syntax_code
100:   {2148532224
101:    ,1081345
102:   }
103:  };
104:  struct T_6_type{
105:   yacco2::ULINT first_entry__[2];
106:  };
107:  T_6_type T_6 = {// for T: LR1_all_shift_operator
108:  //20: TH_linker_preamble_code
109:  //32: TH_rhs_component
110:  //47: TH_unquoted_string
111:  //52: TH_yacco2_syntax_code
112:   {1048576
113:    ,1081345
114:   }
115:  };
116:  struct T_7_type{
117:   yacco2::ULINT first_entry__[2];
118:  };
119:  T_7_type T_7 = {// for T: LR1_fset_transience_operator
120:  //0: TH_T_enum_phrase_th
121:  //9: TH_error_symbols_phrase_th
122:  //14: TH_fsm_class_phrase_th
123:  //15: TH_fsm_phrase_th
124:  //21: TH_lr1_k_phrase_th
125:  //24: TH_parallel_control
126:  //28: TH_parallel_parser_phrase_th
127:  //29: TH_prefile_include
128:  //30: TH_rc_phrase_th
129:  //33: TH_rule_def_phrase
130:  //35: TH_rules_phrase_th
131:  //39: TH_subrules_phrase
132:  //42: TH_terminal_def_phrase
133:  //44: TH_terminals_phrase_th
134:   {1897972225
135:    ,5258
136:   }
137:  };
138:            ...
139:  struct T_510_type{
140:   yacco2::ULINT first_entry__[2];
141:  };
142:  T_510_type T_510 = {// for T: thread_attributes
143:  //20: TH_linker_preamble_code
144:  //32: TH_rhs_component
145:  //47: TH_unquoted_string
146:  //52: TH_yacco2_syntax_code
147:   {1048576
```

```
148:    ,1081345
149:   }
150:  };
151:  struct T_511_type{
152:   yacco2::ULINT first_entry__[2];
153:  };
154:  T_511_type T_511 = {// for T: th_in_stbl
155:  //20: TH_linker_preamble_code
156:  //32: TH_rhs_component
157:  //47: TH_unquoted_string
158:  //52: TH_yacco2_syntax_code
159:   {1048576
160:    ,1081345
161:   }
162:  };
163:  struct T_512_type{
164:   yacco2::ULINT first_entry__[2];
165:  };
166:  T_512_type T_512 = {// for T: kw_in_stbl
167:  //20: TH_linker_preamble_code
168:  //32: TH_rhs_component
169:  //47: TH_unquoted_string
170:  //52: TH_yacco2_syntax_code
171:   {1048576
172:    ,1081345
173:   }
174:  };
175:  struct t_array_type {
176:    yacco2::USINT no_entries__;
177:    yacco2::thd_ids_having_T* first_entry__[513];
178:  };
179:  t_array_type t_array = {
180:    513
181:    ,{(yacco2::thd_ids_having_T*)&T_0 // LR1_eof
182:      ,(yacco2::thd_ids_having_T*)&T_1 // LR1_eog
183:      ,0// LR1_eolr
184:      ,0// LR1_parallel_operator
185:      ,0// LR1_parallel_procedure_call_operator
186:      ,0// LR1_invisible_shift_operator
187:      ,(yacco2::thd_ids_having_T*)&T_6 // LR1_all_shift_operator
188:      ,(yacco2::thd_ids_having_T*)&T_7 // LR1_fset_transience_operator
189:      ,(yacco2::thd_ids_having_T*)&T_8 // raw_nul
190:                  ...
191:      ,(yacco2::thd_ids_having_T*)&T_509 // T_attributes
192:      ,(yacco2::thd_ids_having_T*)&T_510 // thread_attributes
193:      ,(yacco2::thd_ids_having_T*)&T_511 // th_in_stbl
194:      ,(yacco2::thd_ids_having_T*)&T_512 // kw_in_stbl
195:     }
196:  };
197:  void* yacco2::T_ARRAY_HAVING_THD_IDS__ = (void*)&t_array;
198:
```

85. Sample 2: No threads outputted just a stand alone grammar.

```
 1:  //
 2:  // File: c:/yacco2/linker/ts_0_fsc.cpp
 3:  // Generated by linker.exe
 4:  // Date and Time: Wed May 11 16:06:43 2005
 5:  //
 6:
 7:  // Preamble code
 8:  #include <yacco2.h>
 9:  #include <yacco2_T_enumeration.h>
10:  #include <yacco2_err_symbols.h>
11:  #include <yacco2_k_symbols.h>
12:  #include <yacco2_terminals.h>
13:  #include <yacco2_characters.h>
14:  using namespace NS_yacco2_T_enum;
15:  using namespace NS_yacco2_err_symbols;
16:  using namespace NS_yacco2_k_symbols;
17:  using namespace NS_yacco2_terminals;
18:  using namespace NS_yacco2_characters;
19:  // thread include and namespace
20:  // BIT MAPS
21:  #define TOTAL_NO_BIT_WORDS 2*1024*50
22:  int yacco2::TOTAL_NO_BIT_WORDS__(TOTAL_NO_BIT_WORDS);
23:  yacco2::ULINT  bit_maps[TOTAL_NO_BIT_WORDS];
24:  void* yacco2::BIT_MAPS_FOR_SALE__ = (void*)&bit_maps;
25:  int yacco2::BIT_MAP_IDX__(0);
26:  // There are NO THREADS emitted
27:  void* yacco2::THDS_STABLE__ = 0;
28:  void* yacco2::T_ARRAY_HAVING_THD_IDS__ = 0;
29:
```

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

⟨ Include files 86 ⟩ ≡
```
#include "globals.h"
#include "yacco2_stbl.h"
  using namespace yacco2_stbl;
#include "o2linker_externs.h"
#include "link_cleanser.h"
#include "t_alphabet.h"
#include "fsc_file.h"
```
This code is used in section 88.

87. Include O_2^{linker} header.

⟨ io2 87 ⟩ ≡
```
#include "o2linker.h"
```
This code is used in section 89.

88. Create header file for O_2linker environment.
Note, the "include search" directories for the c++ compiler has to be supplied to the compiler environment used. This must include Yac_2o_2's library.

```
⟨ o2linker.h 88 ⟩ ≡
  ⟨ Preprocessor definitions ⟩
#ifndef o2linker__
#define o2linker__ 1
  ⟨ Include files 86 ⟩;
  ⟨ External rtns and variables 14 ⟩;
#endif
```

89. O_2linker implementation.
Start the code output to *o2linker.cpp* by appending its include file.

```
⟨ o2linker.cpp 89 ⟩ ≡
  ⟨ io2 87 ⟩;
  ⟨ accrue linker code 20 ⟩;
```

90. Index.

__FILE__: 26.
__LINE__: 21, 26.
a: 26, 54, 56, 60.
accept: 77.
accept_node: 68.
*act_*: 68.
add_child_at_end: 38.
add_sym_to_stbl: 20.
add_token_to_error_queue: 77.
ADD_TOKEN_TO_ERROR_QUEUE: 77.
add_token_to_producer: 77.
argc: 66.
argv: 66.
AST: 38, 39, 66, 68.
ast_base_stack: 68.
ast_prefix: 43.
*base_stk_*: 43.
bat: 83.
bb: 62.
begin: 19, 21, 23, 24, 25, 31, 32, 36, 38, 40, 44, 45, 47, 54, 57, 58, 60, 61, 64.
*big_buf_*: 42, 44, 45, 46, 48, 49, 50, 68.
BIG_BUFFER_32K: 50, 56, 68.
BIT_MAP_IDX__: 3, 6, 84.
BIT_MAPS_FOR_SALE__: 3, 6, 84.
bit_pos_value: 62.
BITS_PER_WORD: 13, 26, 58, 62, 74.
bugs: 12.
c: 58.
c_str: 17, 18, 19, 22, 24, 32, 36, 42, 44, 45, 46, 48, 50, 51, 54, 57, 58, 61, 62, 65, 68.
c_string: 22, 24, 32, 36, 42, 44, 54, 57, 58, 62, 68.
CAbs_lr1_sym: 20, 21.
call_prt_func: 68.
called_t: 38.
*called_thread_graph_*: 39, 43.
Cfsc_file: 19.
CHAR: 66.
cleanser: 18.
cleanser_fsc: 19.
cleanser_fsc_tokens: 19.
cleanser_fsm: 18.
cleanser_tokens: 18.
clear: 19, 57.
Clink_cleanser: 18, 19.
Clinker_pass3: 17.
close: 50, 51.
*cnode_*: 68.
*cnt_*: 68.
cntl_file_name: 17, 46, 48, 66.
cntl_file_tokens: 17.
code: 26, 61.
content: 68.
cout: 17, 18, 19, 20, 23, 51, 66.
cpp: 12, 71, 89.
crt_called_thread_graph: 36, 39, 40.
crt_called_thread_list: 38.
crt_fset_of_thread: 32, 33, 36.
Ct_alphabet: 18.
cur_stk_rec: 43.
*cur_stk_rec_*: 68.
current_token: 78, 80.
CWEAVE_TITLE_LIMIT: 42.
cweb: 6, 12.
DATE_AND_TIME: 53.
dd: 63.
defed: 20.
*defined_*: 21.
div: 58, 62.
dth_i: 24.
dth_ie: 24.
DUMP_ERROR_QUEUE: 15.
emit_cpp_preamble: 51, 53.
emit_global_bit_maps: 51, 55.
emit_global_thread_include_files: 51, 54.
emit_global_thread_stable: 51, 56.
emit_no_threads: 51, 52.
emit_T_fs_of_potential_threads: 51, 60.
*emitfile_filename_*: 51.
empty: 15, 44, 60, 65.
end: 19, 21, 23, 24, 25, 29, 30, 31, 32, 36, 38, 40, 44, 45, 47, 54, 57, 58, 60, 61, 64.
endl: 17, 18, 19, 20, 23, 24, 32, 36, 42, 44, 45, 46, 48, 49, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 68, 69, 70.
Err_bad_th_in_list: 21.
Error: not enough space for thread bit map manufacture!:
Error_queue: 14, 15, 17, 18, 19, 21, 66.
exec: 43.
exit: 26.
false: 22, 58, 65.
fandk: 42, 47.
fandk_len: 42.
fi: 31.
fie: 31.
filename: 2.
find: 29, 30, 44.
first_entry: 58.
first_item: 65.
first_set_for_threads: 10.
fs: 45.
*fs_*: 29, 30, 45.

fsc: 8.
fsc_cleanser: 19.
fsc_file: 12, 19.
fsc_file_fsm: 19.
fsc_file_output_tokens: 19.
fsc_file_pass: 19.
fsi: 45.
fsie: 45.
fsm: 2.
*fsm_comments_*: 24, 42.
*fully_qualified_T_name_*: 45, 61, 65.
*fully_qualified_th_name_*: 24, 57.
Func: 68.
functor_result_type: 68.
gen_each_grammar_s_referenced_threads: 40, 50.
gen_each_thread_s_referenced_threads: 40.
GET_CMD_LINE: 66.
GRAMMAR_DICTIONARY: 12, 14, 19, 21, 23, 24, 25, 36, 40, 47, 54, 56, 57, 58, 62, 66.
*grammar_file_name_*: 54.
*grammars_fsc_files_*: 19.
id: 20.
Idx: 68.
*idx_*: 68.
ie: 60, 64, 65.
ifstream: 17, 18, 19.
ii: 19.
iie: 19.
includes: 12.
insert: 29, 30.
INT: 68.
INT_SET_ITER_type: 29, 30, 45, 61.
INT_SET_LIST_ITER_type: 60.
INT_SET_LIST_type: 66.
INT_SET_type: 29, 30, 60, 65.
intro: 12.
ios: 50, 51.
ios_base: 50.
iterator: 19, 21, 24, 25, 31, 32, 36, 38, 40, 44, 47, 54, 57.
ithi: 47.
ithie: 47.
je: 61, 62.
KCHARP: 20, 26, 44, 45, 46, 47, 48, 49, 54, 56, 58, 60, 64, 65, 68.
keyword: 20.
kw: 20.
Kw: 20.
kw_in_stbl: 20.
kwkey: 20.
left: 68.
len_a: 22.
len_b: 22.
len_fqna: 22.
len_fqnb: 22.
lex: 12, 79.
li: 32, 38.
lie: 32, 38.
link_cleanser: 12.
linker_cntl_file: 17, 66.
linker_cntl_file_fsm: 17, 18, 19, 51.
Linker_holding_file: 66.
linker_id: 20.
LINKER_PARSE_CMD_LINE: 66.
linker_pass3: 12.
linker_testsuite: 83.
*list_of_transitive_threads_*: 32, 38, 39.
*list_of_Ts_*: 18, 31.
load_linkkw_into_tbl: 20.
load_linkkws_into_tbl: 20, 66.
LOCK_MUTEX: 69.
lrclog: 24, 32, 36, 66, 69, 70.
LR1_ALL_SHIFT_OPERATOR: 13, 29, 31.
LR1_EOG: 13, 29.
LR1_eog: 9.
LR1_EOLR: 13, 29.
LR1_FSET_TRANSIENCE_OPERATOR: 13, 29.
LR1_INVISIBLE_SHIFT_OPERATOR: 13, 29.
LR1_PARALLEL_OPERATOR: 13, 29.
LR1_PROCEDURE_CALL_OPERATOR: 13.
LR1_QUESTIONABLE_SHIFT_OPERATOR: 13, 29.
LR1_questionable_shift_operator: 9.
LR1_REDUCE_OPERATOR: 13, 29.
main: 66.
malloc: 3.
map: 44, 66.
Max_cweb_item_size: 44, 45, 46, 47, 48, 68.
max_thds_supported: 26.
*monolithic_*: 22, 24, 25, 29, 30, 36, 54, 57, 58.
mother_thd_t: 39.
Mother_thd_t: 38.
msg: 26.
*name_space_name_*: 57.
no_lt: 68.
no_of_T: 29, 60, 64.
NO_OF_THREADS: 14, 25, 26, 34, 35, 51, 58, 66.
no_thds_ids: 61.
NO_WORDS_FOR_BIT_MAP: 14, 58, 60, 63, 66.
Node: 66, 68.
*node_*: 68.
NS_eol: 22.
NS_fsc_file: 19.
NS_link_cleanser: 18.
NS_linker_pass3: 17.

NS_t_alphabet: 18.
NS_yacco2_terminals: 66.
o_file: 43, 68.
OFile: 53.
ofile: 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65.
ofstream: 50, 51, 52, 53, 54, 55, 56, 60, 66, 68.
on: 26, 61.
open: 50.
out: 50, 51.
Ow_linker_file: 66, 68.
*ow_linker_file_*: 42, 43, 44, 45, 46, 48, 49, 50, 68.
o2_externs: 14.
o2linker: 12, 89.
o2linker__: 88.
o2linker_defs: 12, 71.
o2linker_doc: 12.
o2linker_externs: 12.
O2linker_VERSION: 66.
p: 25.
Parallel_threads_shutdown: 66.
parse: 17, 18, 19.
Parser: 17, 18, 19.
Passover: 26, 61.
PFF: 68.
pidx: 68.
pms: 12.
pos: 24.
*pos_*: 20.
pre: 43.
Preamble: 53.
*preamble_srce_*: 51.
PRINT_CALLED_THREAD_LIST: 43, 66, 68.
prog: 12.
prt_called_thread_list_ast_functor: 43, 68.
*prt_funct_*: 68.
prt_functr: 43.
prt_prefix: 68.
PRT_SW: 66.
psrec: 68.
push_back: 21, 34.
px__: 78.
p1: 22.
P1: 22.
p1__: 78.
p2: 22.
P2: 22.
P3_tokens: 17.
queue: 77.
quot: 58, 62.
quoted_name: 56, 57.
rebuild_comment: 42, 47.
Recursion_level: 66.
rem: 58, 62.
report_card: 20.
RESERVE_FIXED_NO_THREADS: 28, 66.
reset_cnt: 68.
result: 22.
ri: 25.
rie: 25.
Root_thread: 29, 30, 32, 33.
Root_thread_id: 29, 30, 32, 33.
RSVP: 77.
s_rec: 68.
sampleoutput: 12.
second: 44.
set_rc: 21, 78, 80.
set_stop_parse: 77.
set_who_created: 21.
size: 22, 24, 29, 60, 61.
SMALL_BUFFER_4K: 13, 26, 54, 60.
sort_threads_criteria: 22, 23.
SPECULATIVE_NO_BIT_WORDS: 13, 26, 61, 74.
sprintf: 26, 42, 44, 45, 46, 48, 49, 54, 57, 58, 60, 61, 62, 64, 65, 68.
*srec_*: 68.
stable_sort: 23.
start_token: 78.
stbl_idx: 20.
STBL_T_ITEMS: 66.
STBL_T_ITEMS_type: 66.
std: 17, 18, 19, 21, 24, 25, 31, 32, 36, 38, 40, 42, 44, 45, 46, 47, 48, 49, 50, 54, 55, 56, 57, 58, 60, 61, 62, 64, 65, 66, 68, 69, 70, 82.
Stk_env: 68.
*stk_env_*: 68.
stk_rec: 68.
strcat: 42.
strcmp: 22.
string: 19, 22, 44, 50, 56, 66.
strlen: 42.
Sub_rule_xxx: 78.
sym: 21.
Sym_to_xlate: 66.
*symbol_*: 45, 61, 65.
syntax_code: 51.
T_alphabet: 12, 18.
*t_alphabet_filename_*: 18.
T_array: 64.
T_array_entries: 65.
T_ARRAY_HAVING_THD_IDS__: 3, 64, 73, 84.
T_array_type: 64.
t_att: 45, 61, 65.
T_attributes: 45, 61, 65.

T_DICTIONARY: 12, 14, 18, 29, 45, 60, 61, 65, 66.
T_emitfile: 20.
T_end_list_of_native_first_set_terminals: 20.
T_end_list_of_transitive_threads: 20.
T_end_list_of_used_threads: 20.
T_end_preamble: 20.
T_end_T_alphabet: 20.
t_entry: 45, 61, 65.
t_enum: 30, 31.
T_file_name: 20.
T_file_of_T_alphabet: 20.
T_file_tokens: 18.
T_fsc_file_tokens: 19.
T_fsm: 18.
T_fsm_comments: 20.
T_grammar_name: 20.
t_id: 65.
t_in_stbl: 45, 61, 65.
T_list_of_native_first_set_terminals: 20.
T_list_of_transitive_threads: 20.
T_list_of_used_threads: 20.
T_list_to_thd_list_type: 60.
T_list_to_thd_list_var: 60, 61, 65.
t_listi: 29, 30.
T_monolithic: 20.
T_name_space: 20.
T_no_of_T: 20.
T_pass3: 18.
T_preamble: 20.
T_sym_tbl_report_card: 20.
T_T_alphabet: 20.
T_THREAD_ID_LIST: 12, 14, 18, 29, 30, 60, 64, 66.
T_thread_name: 20.
T_tokens: 18.
T_transitive: 20.
ta: 68.
table_entry: 20, 21, 22, 24, 25, 36, 40, 45, 47, 54, 57, 61, 62, 65, 66.
tbl_entry: 21, 24, 25, 47, 62.
tenum: 45.
terminal_id: 60, 61.
testsuites: 12.
th_att: 32, 36, 38, 40, 42, 43, 44, 45, 54, 57, 58, 62.
*th_enum_*: 24, 25, 29, 30, 33, 36, 38, 39, 57.
TH_eol: 22.
th_goodies: 21, 24, 25.
th_i: 21.
th_id: 62.
th_ie: 21.
th_list: 29, 30, 60, 61, 65.
th_name: 57.
th_tbl: 36, 40, 54, 57, 58, 62.
th_tbl1: 22.
th_tbl2: 22.
thd_id_in_list: 60, 62.
Thd_list: 38.
THDS_STABLE__: 3, 6, 56, 73, 84.
thi: 36, 40, 54, 57, 58.
thie: 36, 40, 54, 57, 58.
thread_array: 58.
thread_attributes: 22, 32, 33, 36, 38, 39, 40, 54, 57, 58, 62, 68.
Thread_entry: 5, 56.
thread_entry: 56, 57.
thread_entry_name: 58.
THREAD_ID_FIRST_SET: 12.
THREAD_ID_FS: 14.
thread_in_stbl: 22, 36, 40, 54, 57, 58, 62.
thread_include_ns: 54.
*thread_name_*: 22, 24, 32, 36, 42, 44, 57, 58, 62, 68.
ti: 44.
tok_can: 17, 18, 19.
TOKEN_GAGGLE: 17, 18, 19, 66.
TOTAL_NO_BIT_WORDS: 3, 13.
TOTAL_NO_BIT_WORDS__: 3, 84.
TOTAL_NO_OF_BIT_WORDS__: 6.
toupper: 22.
TRACE_MU: 69, 70.
true: 15, 21, 22, 58, 60, 65, 77.
trunc: 50.
tt: 44.
tth_in_stbl: 45, 61, 65.
tti: 44.
ttie: 44.
types: 12.
ucase_a: 22.
ucase_b: 22.
ucase_fqna: 22.
ucase_fqnb: 22.
ULINT: 61, 62.
UNLOCK_MUTEX: 70.
used_threads: 44.
USED_THREADS_LIST: 12, 44, 66.
vector: 19, 21, 24, 25, 31, 32, 36, 38, 40, 44, 47, 54, 57, 66, 82.
vi: 34, 35.
visit_graph: 81.
Visit_graph: 28, 33, 34, 35, 38, 39, 66, 82.
Visited_th: 29, 30, 31, 32, 33, 39.
w_called_threads: 42, 47, 68.
w_called_threads_index: 68.
w_comments: 42, 47.
w_doc_comments: 46.
w_doc_index: 46.

w_fsc_file_listing: 48.
w_grammar: 42, 47.
w_index: 49.
*w_linker_filename_*: 46, 50.
walk_called_thread_list: 38, 39.
word_map: 61, 62, 63.
write: 42, 44, 45, 46, 48, 49, 54, 57, 58, 60, 61, 62, 64, 65, 68.
x: 22, 29, 42, 44, 45, 46, 54, 57, 58, 60, 62, 64, 65, 68.
xlate_file: 46.
xlate_fscfile: 46.
xlate_gfile: 42, 47, 68.
XLATE_SYMBOLS_FOR_cweave: 42, 44, 45, 46, 48, 66, 68.
xlate_thnm: 44, 47.
xlate_tnm: 45, 47.
xlated_filename: 48.
Xlated_sym: 66.
xxx: 12.
yacco2: 3, 12, 20, 24, 36, 66, 68, 69, 70, 84.
yacco2_characters: 12.
YACCO2_define_trace_variables: 66.
yacco2_err_symbols: 12.
Yacco2_faulty_precondition: 26.
yacco2_k_symbols: 12.
yacco2_linker_keywords: 79.
YACCO2_MU_TRACING__: 69, 70.
yacco2_stbl: 20, 86.
YACCO2_STBL: 12.
yacco2_T_enumeration: 8, 12.
yacco2_terminals: 12.
YES: 44.

⟨ External rtns and variables 14, 28 ⟩ Used in section 88.
⟨ Include files 86 ⟩ Used in section 88.
⟨ Structure implementations 68 ⟩ Used in section 71.
⟨ accrue linker code 20, 22, 33, 38, 39, 40, 52, 53, 54, 55, 56, 60, 66 ⟩ Used in section 89.
⟨ acquire trace mu 69 ⟩ Used in section 68.
⟨ allocation space for *Visit_graph* 34 ⟩ Used in section 36.
⟨ announce the stable to the world 59 ⟩ Used in section 58.
⟨ associate native terminals with called thread 31 ⟩ Used in section 33.
⟨ calculate terminal's thread bit map 62 ⟩ Used in section 61.
⟨ check whether Linker has enough space to gen thread bit maps: no throw up 26 ⟩ Used in section 66.
⟨ count and re-align threads enumerate values to sorted position 25 ⟩ Used in section 66.
⟨ create terminal's thread bit map 61 ⟩ Used in section 60.
⟨ deal with threads having T in first set 30 ⟩ Used in section 31.
⟨ dump sorted dictionary 24 ⟩ Used in section 66.
⟨ emit array of Terminals' thread bit maps 64 ⟩ Used in section 60.
⟨ emit code 51 ⟩ Used in section 66.
⟨ emit terminal's thread bit map 63 ⟩ Used in section 61.
⟨ gen global thread array 58 ⟩ Used in section 56.
⟨ gen thread list 57 ⟩ Used in section 56.
⟨ generate linker document 50 ⟩ Used in section 66.
⟨ generate threads final first sets 36 ⟩ Used in section 66.
⟨ if error queue not empty then deal with posted errors 15 ⟩ Used in sections 17, 18, 19, 21, and 66.
⟨ initialize *Visit_graph* to not visited 35 ⟩ Used in sections 36 and 40.
⟨ io2 87 ⟩ Used in section 89.
⟨ loop thru grammars to gen their local linker doc info 47 ⟩ Used in section 50.
⟨ make grammar's contents cweaveable and output 42 ⟩ Used in section 47.
⟨ `o2linker.cpp` 89 ⟩
⟨ `o2linker.h` 88 ⟩
⟨ `o2linker_defs.cpp` 71 ⟩
⟨ output First set of linker 48 ⟩ Used in section 50.
⟨ output Index of linker 49 ⟩ Used in section 50.
⟨ output grammar's called threads list 43 ⟩ Used in section 47.
⟨ output grammar's first set 45 ⟩ Used in section 47.
⟨ output grammar's used threads 44 ⟩ Used in section 47.
⟨ output preamble of document 46 ⟩ Used in section 50.
⟨ parse T alphabet 18 ⟩ Used in section 66.
⟨ parse fsc files 19 ⟩ Used in section 66.
⟨ parse linker control file 17 ⟩ Used in section 66.
⟨ post verify that there are no threads "used" and not "defined" 21 ⟩ Used in section 66.
⟨ print out each thread set 65 ⟩ Used in section 64.
⟨ probagate $|+|$ 29 ⟩ Used in section 31.
⟨ process called thread's list 32 ⟩ Used in section 33.
⟨ release trace mu 70 ⟩ Used in section 68.
⟨ sort thread dictionary 23 ⟩ Used in section 66.

O2LINKER

	Section	Page
License	1	1
Summary of Yacco2's Linker — threads and their bit maps	2	2
Globals — those unresolved static variables used by Yacco2's library	3	3
Some definitions within Linker's context	4	4
Overview of O_2^{linker}'s generated components	5	4
Thread bit maps	6	5
Linker's languages	7	5
Terminal alphabet	8	6
Terminal enumeration	9	6
First set declarations	10	7
Linker's control language	11	8
Catalogue of Linker's files	12	9
Global macro definitions	13	10
External routines and globals	14	10
Local routines	16	11
Parse linker control file	17	11
Parse T alphabet	18	11
Parse fsc files	19	12
load_linkkw_into_tbl	20	13
Verify that all threads used are defined	21	14
Sort thread dictionary	22	15
Sort uses template algorithm	23	16
Dump sorted dictionary	24	16
Count and re-align threads enumerate values to sorted position	25	17
Thread graphs: first set generation	27	18
Visit_graph	28	18
Probagate $\vert + \vert$	29	18
Deal with threads having T in first set	30	19
Associate native terminals with called thread	31	19
Process called thread's list	32	19
crt_fset_of_thread	33	20
Allocation space for *Visit_graph*	34	20
Initialize *Visit_graph* to "not visited"	35	20
Generate those first sets	36	21

Generate document for each grammar's called threads . . . 37 22
crt_called_thread_list and *walk_called_thread_list* . . . 38 22
crt_called_thread_graph . . . 39 22
gen_each_thread_s_referenced_threads . . . 40 22
Generate Linker's document . . . 41 23
Make grammar's contents cweaveable and output . . . 42 23
Output grammar's called threads list . . . 43 23
Output grammar's used threads . . . 44 24
Output grammar's first set . . . 45 24
Output preamble of document . . . 46 25
Loop thru grammars to gen their local linker doc info . . . 47 25
Output First set of linker . . . 48 26
Output Index of linker . . . 49 26
Output driver of the linker document . . . 50 26
Emit code . . . 51 27
Emit no threads . . . 52 27
Emit cpp preamble . . . 53 28
Emit thread include files . . . 54 28
Emit global bit maps . . . 55 28
Emit global thread stable `THDS_STABLE__` . . . 56 29
The threading stew . . . 57 29
The table hôte . . . 58 30
Announce the stable to the world . . . 59 30
Emit global Terminals' thread bit maps . . . 60 31
Create terminal's thread bit map . . . 61 32
Calculate terminal's thread bit map . . . 62 32
Emit terminal's thread bit map . . . 63 32
Emit Terminals' thread bit maps and global `T_ARRAY_HAVING_THD_IDS__` . . . 64 33
Print each entry . . . 65 33
Main line of Linker . . . 66 34
Structure implementation . . . 67 35
prt_called_thread_list_ast_functor implementation . . . 68 35
Acquire trace mu . . . 69 36
Release trace mu . . . 70 36
Write out *o2linker_defs.cpp* **Structure implementations** . . . 71 37
PMS — Post meta syndrome . . . 72 38
Bugs — ugh . . . 76 39
Test suites: Check out those flabby grammar muscles . . . 83 42
Sample output from Linker . . . 84 55
Sample 2: No threads outputted just a stand alone grammar . . . 85 59
Include files . . . 86 59
Create header file for O_2linker environment . . . 88 60
O_2linker implementation . . . 89 60
Index . . . 90 61