1. License.

2. About this document. The original intent was that this document would evolve toward a book describing the code, in roughly the same form as those that Don Knuth produces using this system. But in fact this document has evolved into very heavily commented source code. There are lots and lots of notes, many quite detailed, but little thought is being given to the overall "structure" that a book would need. Maybe someday.

3. One focus is on those sections which have caused the most trouble – I make it a habit to think the ideas through and record my thoughts here. That means that those sections which never cause me any problems are very lightly documented.

4. A second focus is on matters that are unlikely to emerge from the code itself. The matters include

- Alternative implementations, and the reasons they might be worse and/or better;
- Analysis of time and space complexity;
- Where needed, proofs of correctness; and
- Other mathematical or theoretical considerations.

5. This document and this way of documenting has proved invaluable for me in keeping up what has become a mass of complex code. I fear, though, it is less helpful for any other reader, even a technically very savvy one.

6. Marpa is a very unusual C library – no system calls, no floating point and almost no arithmetic. A lot of data structures and pointer twiddling. I have found that a lot of good coding practices in other contexts are not in this one.

7. As one example, I intended to fully to avoid abbreviations. This is good practice – in most cases all abbreviations save is some typing, at a very high cost in readability. In `libmarpa`, however, spelling things out usually does **not** make them more readable. To be sure, when I say

`To_AHFA_of_YIM_by_NSYID`

that is pretty incomprehensible. But is

Aycock_Horspool_FA_To_State_of_Earley_Item_by_Internal_Symbol_ID

, where "Finite Automaton" must still be abbreviated as "FA" to allow the line to fit into 80 characters, really any better? My experience say no.

8. I have a lot of practice coming back to pages of both, cold, and trying to figure them out. Both are daunting, but the abbreviations are more elegant, and look better on the page, while unabbreviated names routinely pose almost insoluble problems for Cweb's TEX typesetting.

Whichever is used, it must be kept systematic and documented, and that is easier with the abbreviations. In general, I believe abbreviations are used in code far more than they should be. But they have their place and `libmarpa` is one of them.

Because I realized that abbreviations were going to be not just better, but almost essential if I ever was to finish this project, I changed from a "no abbreviation" policy to one of "abbreviate when necessary and it is necessary a lot" half way through. Thus the code is highly inconsistent in this respect. At the moment, that's true of a lot of my other coding conventions.

9. The reader should be aware that the coding conventions may not be consistent internally, or consistent with their documentation.

10. Design.

11. Object pointers. The major objects of the `libmarpa` layer are passed to upper layers as pointer, which hopefully will be treated as opaque. `libmarpa` objects are reference-counted.

12. Inlining. Most of this code in `libmarpa` will be frequently executed. Inlining is used a lot. Enough so that it is useful to define a macro to let me know when inlining is not used in a private function.

#define PRIVATE_NOT_INLINE static
#define PRIVATE static inline

13. Marpa global Setup.

Marpa has only a few non-constant globals as of this writing. All of them are exclusively for debugging. For thread-safety, among other reasons, all other globals are constants.

The debugging-related globals include a pointer debugging handler, and the debug level. It is assumed that the application will change these in a thread-safe way before starting threads.

14. Complexity. Considerable attention is paid to time and, where it is a serious issue, space complexity. Complexity is considered from three points of view. **Practical worst-case complexity** is the complexity of the actual implementation, in the worst-case. **Practical average complexity** is the complexity of the actual implementation under what are expected to be normal circumstances. Average complexity is of most interest to the typical user, but worst-case considerations should not be ignored — in some applications, one case of poor performance can outweigh any number of of excellent "average case" results.

15. Finally, there is **theoretical complexity**. This is the complexity I would claim in a write-up of the Marpa algorithm for a Theory of Computation article. Most of the time, I am conservative, and do not claim a theoretical complexity better than the practical worst-case complexity. Often, however, for theoretical complexity I consider myself entitled to claim the time complexity for a better algorithm, even though that is not the one used in the actual implementation.

16. Sorting is a good example of a case where I take the liberty of claiming a time complexity better than the one I actually implemented. In many places in `libmarpa`, for sorting, the most reasonable practical implementation (sometimes the only reasonable practical implementation) is an $O(n^2)$ sort. When average list size is small, for example, a hand-optimized insertion sort is often clearly superior to all other alternatives. Where average list size is larger, a call to `qsort` is the appropriate response. `qsort` is the result of considerable thought and experience, the GNU project has decided to base it on quicksort, and I do not care to second-guess them on this. But quicksort and insertion sorts are both, theoretically, $O(n^2)$.

17. Clearly, in both cases, I could drop in a merge sort and achieve a theoretical time complexity $O(n \log n)$ in the worst case. Often it is just as clear that, in practice, the merge sort would be inferior.

18. When I claim a complexity from a theoretical choice of algorithm, rather than the actually implemented one, the following will always be the case:

- The existence of the theoretical algorithm must be generally accepted.
- The complexity I claim for it must be generally accepted.
- It must be clear that there are no serious obstacles to using the theoretical algorithm.

19. I am a big believer in theory. Often practical considerations didn't clearly indicate a choice of algorithm . In those circumstances, I usually allowed theoretical superiority to be the deciding factor.

20. But there were cases where the theoretically superior choice was clearly going to be inferior in practice. Sorting was one of them. It would be possible to go through `libmarpa` and replace all sorts with a merge sort. But a slower library would be the result.

21. Coding conventions.

22. External functions.
All libmarpa's external functions, without exception, begin with the prefix `marpa_`. All libmarpa's external functions fall into one of three classes:

- Version-number-related function follow GNU naming conventions.
- The functions for libmarpa's obstacks have name with the prefix `marpa_obs_`. These are not part of libmarpa's external interface, but so they do have external linkage so that they can be compiled separately.
- Function for one of libmarpa's objects, which begin with the prefix `marpa_X_`, where X is a one-letter code which designates one of libmarpa's objects.

23. Objects.
When I find it useful, libmarpa uses an object-oriented approach. One such case is the classification and naming of external functions. This can be seen as giving libmarpa an object-oriented structure, overall. The classes of object used by libmarpa have one letter codes.

- g: grammar.
- r: recognizer.
- b: bocage.
- o: ordering.
- t: tree.
- v: evaluator.

24. Reserved locals. Certain symbol names are reserved for certain purposes. They are not necessarily defined, but if defined, and once initialized, they must be used for the designated purpose. An example is g, which is the grammar of most interest in the context. (In fact, no marpa routine uses more than one grammar.) It is expected that the routines which refer to a grammar will set g to that value. This convention saves a lot of clutter in the form of macro and subroutine arguments.

- g is the grammar of most interest in the context.
- r is the recognizer of most interest in the context.
- `irl_count` is the number of internal rules in g.
- `xrl_count` is the number of external rules in g.

25. Mixed case macros. In programming in general, accessors are very common. In `libmarpa`, the percentage of the logic that consists of accessors is even higher than usual, and their variety approaches the botanical. Most of these accessors are simple or even trivial, but some are not. In an effort to make the code readable and maintainable, I use macros for all accessors.

26. The standard C convention is that macros are all caps. This is a good convention. I believe in it and usually follow it. But in this code I have departed from it.

27. As has been noted in the email world, when most of a page is in caps, that page becomes much harder and less pleasant to read. So in this code I have made macros mixed case. Marpa's mixed case macros are easy to spot — they always start with a capital, and the "major words" also begin in capital letters. "Verbs" and "coverbs" in the macros begin with a lower case letter. All words are separated with an underscore, as is the currently accepted practice to enhance readability.

28. The "macros are all caps" convention is a long standing one. I understand that experienced C programmers will be suspicious of my claim that this code is special in a way that justifies breaking the convention. Frankly, if I were a new reader coming to this code, I would be suspicious as well. But I would ask anyone who wishes to criticize to first do the following: Look at one of the many macro-heavy pages in this code and ask yourself – do you genuinely wish more of this page was in caps?

29. External names. External Names have `marpa_` or `MARPA_` as their prefix, as appropriate under the capitalization conventions. Many names begin with one of the major "objects" of Marpa: grammars, recognizers, symbols, etc. Names of functions typically end with a verb.

30. Booleans. Names of booleans are often of the form `is_x`, where x is some property. For example, the element of the symbol structure which indicates whether the symbol is a terminal or not, is `is_terminal`. Boolean names are chosen so that the true or false value corresponds correctly to the question implied by the name. Names should be as accurate as possible consistent with brevity. Where possible, consistent with brevity and accuracy, positive names (`is_found`) are preferred to negative names (`is_not_lost`).

31. Abbreviations and vocabulary.

32. Unexplained abbreviations and non-standard vocabulary pose unnecessary challenges. Particular obstacles to those who are not native speakers of English, they are annoying to the natives as well. This section is intended eventually to document all abbreviations, as well as non-standard vocabulary. By "non-standard vocabulary", I mean terms that can not be found in a general dictionary or in the standard reference works. Non-standard vocabulary may be ommitted if it is explained in detail where it occurs.

33. As of this writing, this section is very incomplete and possibly obsolete.

34.
- alloc: Allocate.
- AHFA: Aycock-Horspool Finite Automaton.
- AHM: Aycock-Horspool item.
- AIMID: a legacy term for AHM ID, preserved for backward compatibility.
- assign: Find something, creating it when necessary.
- bv: Bit Vector.
- cmp: Compare. Usually as `_cmp`, the suffix or "verb" of a function name.

- _Object: As a suffix of a type name, this means an object, as opposed to a pointer. When there is a choice, most complex types are considered to be pointers to structures or unions, rather than the structure or union itself. When it's necessary to have a type which refers to the actual structure or union **directly**, not via a pointer, that type is called the "object" form of the type. As an example, look at the definitions of *YIM* and `YIM_Object`. (These begin with a 'Y' because C89 reserves names starting with 'E'.)
- eim: Earley item. Used for clarity in a few places where C89 reserved names are not an issue.
- es: Earley set. Used for clarity in a few places were
- g: Grammar.
- IRL: Internal Rule.
- `_ix`, `_IX`, ix, IX: Index. Often used as a suffix.
- JEARLEME: Used instead of `EARLEME` because C89 reserves names starting with a capital 'E'.
- Leo base item: The Earley item which "causes" a Leo item to be added. If a Leo chain in reconstructed from the Leo item,
- Leo completion item: The Earley item which is the "successor" of a Leo item to be added.
- Leo LHS symbol: The LHS of a Leo completion item (see which).
- Leo item: A "transition item" as described in Leo1991. These stand in for a Leo chain of one or more Earley tems. Leo items can stand in for all the Earley items of a right recursion, and it is the use of Leo items which makes this algorithm $O(n)$ for all LR-regular grammars. In an Earley implementation without Leo items, a parse with right recursion can have the time complexity $O(n^2)$.
- LBV: Lightweight Boolean Vector.
- LBW: LBV Word.
- LIM: Leo item.
- NOOK, nook: any node of a parse tree, a pun on both "node" and "fork".
- NSY, nsy: Internal symbol. This is inconsistent with the use of 'I' for internal, as in `IRL`, for internal rule. C89 reserves names beginning in 'is', making this inconsistency necessary.
- `ord_`, `Ord_`, `_ord`, `_Ord`, ord, Ord: ordinal of the Earley set. Often used as a prefix or a suffix.
- p: A Pointer. Often as `_p`, as the end of a variable name, or as `p_` at the beginning of one.
- pp: A Pointer to pointer. Often as `_pp`, as the end of a variable name.
- PIM, pim: Postdot item.
- PSI: Per Set and Item – a container of data per Earley Set and, within that, Earley Item.
- R, r: Recognizer.
- RECCE, recce: Recognizer. Originally British military slang for a reconnaissance.
- -s, -es: Plural. Note that the `es` suffix is often used even when it is not good English, because it is easier to spot in text. For example, the plural of *YS* is `YSes`.

- `s_`: Prefix for a structure tag. Cweb does not format C code well unless tag names are distinct from other names.
- SRCL: Source Link.
- `t_`: Prefix for an element tag. Cweb does not format C code well unless tag names are distinct from others. Since each structure and union in C has a different namespace, this does not suffice to make different tags unique, but it does suffice to let Cweb distinguish tags from other items, and that is the object.
- tkn: Token. Needed because C89 reserves names beginning with 'to'.
- `u_`: Prefix for a union tag. Cweb does not format C code well unless tag names are distinct from other names.
- UR: Ur-nodes, precursors of and-nodes and or-nodes.
- URS: UR Stack.
- `YIM_Object`: Earley item (object). 'Y' is used instead of 'E' because C89 reserveds names starting with a capital 'E'.
- XRL: External Rule.
- XSY: External Symbol.
- YIX: Earley item index.
- YS: Earley set.

35. Maintenance notes.

36. Where is the source?.

Most of the source code for `libmarpa` in the Cweb file `marpa`.*w*, which is also the source for this document. But error codes and public function prototypes are taken from `api.texi`, the API document. (This helps keep the API documentation in sync with the source code.) To change error codes or public function prototypes, look at `api.texi` and the scripts which process it.

37. The public header file.

38. Version constants.

39. This macro checks that the header version numbers (MARPA_xxx_VERSION) and the library version numbers (MARPA_LIB_xxx_VERSION) are identical. It is a sanity check. The best argument for the cost-effectiveness here is that the check is almost certainly cost-free at runtime – it is all compile-time constants, which I can reasonably expect to be optimized out.

```
#define HEADER_VERSION_MISMATCH (MARPA_LIB_MAJOR_VERSION ≠
        MARPA_MAJOR_VERSION ∨ MARPA_LIB_MINOR_VERSION ≠
        MARPA_MINOR_VERSION ∨ MARPA_LIB_MICRO_VERSION ≠
        MARPA_MICRO_VERSION)
```

40. Set globals to the library version numbers, so that they can be found at runtime.

```
⟨ Global constant variables 40 ⟩ ≡
  const int marpa_major_version ⟸ MARPA_LIB_MAJOR_VERSION;
  const int marpa_minor_version ⟸ MARPA_LIB_MINOR_VERSION;
  const int marpa_micro_version ⟸ MARPA_LIB_MICRO_VERSION;
```

See also sections 829, 884, and 1125.

This code is used in section 1382.

41. Check the arguments, which will usually be the version numbers from macros in the public header file, against the compiled-in version number. Currently, we don't support any kind of backward or forward compatibility here.

```
⟨ Function definitions 41 ⟩ ≡
  Marpa_Error_Codemarpa_check_version(int required_major, int required_minor, int
        required_micro)
  {
    if (required_major ≠ marpa_major_version)
      return MARPA_ERR_MAJOR_VERSION_MISMATCH;
    if (required_minor ≠ marpa_minor_version)
      return MARPA_ERR_MINOR_VERSION_MISMATCH;
    if (required_micro ≠ marpa_micro_version)
      return MARPA_ERR_MICRO_VERSION_MISMATCH;
    return MARPA_ERR_NONE;
  }
```

See also sections 42, 45, 46, 51, 55, 57, 58, 63, 65, 66, 67, 74, 76, 80, 81, 94, 95, 99, 102, 116, 117, 118, 119, 139, 140, 146, 147, 149, 152, 153, 163, 164, 165, 168, 171, 174, 177, 181, 182, 185, 188, 189, 190, 193, 194, 195, 198, 199, 200, 201, 207, 211, 213, 220, 221, 222, 223, 226, 229, 232, 235, 240, 243, 248, 249, 252, 258, 259, 261, 262, 266, 269, 270, 271, 272, 273, 278, 279, 282, 283, 290, 293, 298, 302, 306, 309, 312, 316, 319, 322, 324, 333, 335, 337, 343, 346, 352, 355, 358, 361, 364, 368, 379, 412, 461, 478, 479, 481, 483, 491, 542, 543, 544, 545, 551, 555, 556, 557, 567, 568, 571, 572, 575, 582, 583, 586, 588, 590, 592, 604, 605, 612, 639, 640, 641, 642, 643, 653, 654, 659, 671, 672, 689, 690, 691, 692, 694, 704, 706, 707, 709, 710, 711, 719, 737, 754, 755, 756, 757, 773, 802, 819, 821, 822, 831, 832, 833, 835, 836, 837, 861, 862, 863, 864, 865, 866, 868, 869, 871, 896, 906, 907, 915, 920, 921, 922, 925, 942, 955, 959, 963, 964, 966, 970, 977, 981, 982, 983, 987, 991, 994, 995, 999, 1006, 1007, 1008, 1024, 1025, 1030, 1031, 1032, 1037, 1038, 1039, 1046, 1047, 1065, 1066, 1083, 1087, 1088, 1089, 1096, 1099, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1112, 1117, 1118, 1119, 1120, 1122, 1123, 1126, 1127, 1129, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1139, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151, 1156, 1161, 1163, 1165, 1166, 1167, 1168, 1170, 1172, 1174, 1175, 1187, 1188, 1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1212, 1213, 1214, 1215, 1218, 1220, 1222, 1223, 1224, 1252, 1253, 1257, 1262, 1263, 1264, 1266, 1271, 1273, 1275, 1276, 1278, 1279, 1280, 1283, 1285, 1286, 1287, 1292, 1295, 1297, 1300, 1302, 1305, 1307, 1308, 1309, 1311, 1313, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1329, 1330, 1332, 1334, 1335, 1336, 1337, 1338, 1339, 1342, 1343, 1344, 1345, 1346, 1347, 1348, 1351, 1353, 1355, 1361, 1363, 1365, and 1366.

This code is used in section 1384.

42. Returns the compiled-in version – not the one in the headers. Always succeeds at this point.

⟨ Function definitions 41 ⟩ +≡

```
Marpa_Error_Codemarpa_version(int *version)
{
  *version++ ⟸ marpa_major_version;
  *version++ ⟸ marpa_minor_version;
  *version ⟸ marpa_micro_version;
  return 0;
}
```

43. Config (C) code.

44. ⟨ Public structures 44 ⟩ ≡

```
struct marpa_config {
  int t_is_ok;
  Marpa_Error_Code t_error;
  const char *t_error_string;
};
typedef struct marpa_config Marpa_Config;
```

See also sections 110, 828, 1072, 1349, and 1358.

This code is used in section 1387.

45. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_c_init(Marpa_Config *config)
{
  config→t_is_ok ⇐ I_AM_OK;
  config→t_error ⇐ MARPA_ERR_NONE;
  config→t_error_string ⇐ Λ;
  return 0;
}
```

46. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Error_Code marpa_c_error(Marpa_Config *config, const char
        **p_error_string)
{
  const Marpa_Error_Code error_code ⇐ config→t_error;
  const char *error_string ⇐ config→t_error_string;
  if (p_error_string) {
    *p_error_string ⇐ error_string;
  }
  return error_code;
}
const char *_marpa_tag(void)
{
#if defined (MARPA_TAG)
  return STRINGIFY(MARPA_TAG);
#elif defined (__GNUC__)
  return __DATE__ "␣" __TIME__;
#else
  return "[no␣tag]";
#endif
}
```

47. Grammar (GRAMMAR) code.

⟨ Public incomplete structures 47 ⟩ ≡

```
struct marpa_g;
struct marpa_avl_table;
typedef struct marpa_g *Marpa_Grammar;
```

See also sections 548, 667, 935, 971, 972, 1020, and 1068.

This code is used in section 1387.

48. ⟨ Private structures 48 ⟩ ≡

```
struct marpa_g {
  ⟨ First grammar element 133 ⟩
  ⟨ Widely aligned grammar elements 59 ⟩
  ⟨ Int aligned grammar elements 53 ⟩
  ⟨ Bit aligned grammar elements 97 ⟩
};
```

See also sections 111, 144, 217, 254, 326, 378, 453, 534, 537, 618, 629, 630, 661, 664, 699, 856, 857, 880, 881, 882, 883, 905, 931, 947, 973, 1022, 1159, 1180, 1206, and 1208.

This code is used in section 1381.

49. ⟨ Private typedefs 49 ⟩ ≡

```
typedef struct marpa_g *GRAMMAR;
```

See also sections 142, 216, 255, 328, 470, 529, 536, 549, 625, 627, 652, 670, 679, 682, 823, 875, 903, 929, 1014, 1116, 1124, 1182, and 1186.

This code is used in section 1381.

50. Constructors.

51. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Grammar marpa_g_new(Marpa_Config *configuration)
{
  GRAMMAR g;
  if (configuration ∧ configuration→t_is_ok ≠ I_AM_OK) {
    configuration→t_error ⟸ MARPA_ERR_I_AM_NOT_OK;
    return Λ;
  }
  g ⟸ my_malloc(sizeof(struct marpa_g));
    /* Set t_is_ok to a bad value, just in case */
  g→t_is_ok ⟸ 0;
  ⟨ Initialize grammar elements 54 ⟩
    /* Properly initialized, so set t_is_ok to its proper value */
  g→t_is_ok ⟸ I_AM_OK;
  return g;
}
```

52. Reference counting and destructors.

53. ⟨ Int aligned grammar elements 53 ⟩ ≡

```
int t_ref_count;
```

See also sections 78, 82, 85, 88, 92, 136, 161, 457, and 471.

This code is used in section 48.

54. ⟨ Initialize grammar elements 54 ⟩ ≡

```
g→t_ref_count ⇐ 1;
```

See also sections 60, 69, 79, 83, 86, 89, 93, 98, 101, 104, 106, 113, 121, 125, 128, 137, 162, 459, 531, and 539.

This code is used in section 51.

55. Decrement the grammar reference count. GNU practice seems to be to return *void*, and not the reference count. True, that would be mainly useful to help a user shot himself in the foot, but it is in a long-standing UNIX tradition to allow the user that choice.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void grammar_unref(GRAMMAR g)
{
  MARPA_ASSERT(g→t_ref_count > 0)g→t_ref_count−−;
  if (g→t_ref_count ≤ 0) {
    grammar_free(g);
  }
}
void marpa_g_unref(Marpa_Grammar g)
{
  grammar_unref(g);
}
```

56. Increment the grammar reference count.

57. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE GRAMMAR grammar_ref(GRAMMAR g)
{
  MARPA_ASSERT(g→t_ref_count > 0)g→t_ref_count++;
  return g;
}
Marpa_Grammar marpa_g_ref(Marpa_Grammar g)
{
  return grammar_ref(g);
}
```

58. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void grammar_free(GRAMMAR g)
{
  ⟨ Destroy grammar elements 61 ⟩
  my_free(g);
}
```

59. The grammar's symbol list. This lists the symbols for the grammar, with their *Marpa_Symbol_ID* as the index.

⟨ Widely aligned grammar elements 59 ⟩ ≡

```
MARPA_DSTACK_DECLARE(t_xsy_stack);
MARPA_DSTACK_DECLARE(t_nsy_stack);
```

See also sections 68, 103, 105, 112, 120, 124, 127, 135, 456, 530, and 538.

This code is used in section 48.

60. ⟨ Initialize grammar elements 54 ⟩ +≡

```
MARPA_DSTACK_INIT2(g→t_xsy_stack, XSY);
MARPA_DSTACK_SAFE(g→t_nsy_stack);
```

61. ⟨ Destroy grammar elements 61 ⟩ ≡

```
{
  MARPA_DSTACK_DESTROY(g→t_xsy_stack);
  MARPA_DSTACK_DESTROY(g→t_nsy_stack);
}
```

See also sections 70, 114, 123, 126, 129, 460, 532, 540, and 541.

This code is used in section 58.

62. Symbol count accesors.

```
#define XSY_Count_of_G(g) (MARPA_DSTACK_LENGTH((g)→t_xsy_stack))
```

63. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_highest_symbol_id(Marpa_Grammar g)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  return XSY_Count_of_G(g) − 1;
}
```

64. Symbol by ID.

```
#define XSY_by_ID(id) (∗MARPA_DSTACK_INDEX(g→t_xsy_stack, XSY, (id)))
```

65. Adds the symbol to the list of symbols kept by the Grammar object.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void symbol_add(GRAMMAR g, XSYsymbol)
{
  const XSYIDnew_id ⟸ MARPA_DSTACK_LENGTH((g)→t_xsy_stack);
  ∗MARPA_DSTACK_PUSH((g)→t_xsy_stack, XSY) ⟸ symbol;
  symbol→t_symbol_id ⟸ new_id;
}
```

66. Check that external symbol is in valid range.

```
#define XSYID_is_Malformed(xsy_id) ((xsy_id) < 0)
#define XSYID_of_G_Exists(xsy_id) ((xsy_id) < XSY_Count_of_G(g))
```

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int xsy_id_is_valid(GRAMMAR g, XSYIDxsy_id)
{
  return ¬XSYID_is_Malformed(xsy_id) ∧ XSYID_of_G_Exists(xsy_id);
}
```

67. Check that internal symbol is in valid range.

```
#define NSYID_is_Malformed(nsy_id) ((nsy_id) < 0)
#define NSYID_of_G_Exists(nsy_id) ((nsy_id) < NSY_Count_of_G(g))
```

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int nsy_is_valid(GRAMMAR g, NSYIDnsyid)
{
  return nsyid ≥ 0 ∧ nsyid < NSY_Count_of_G(g);
}
```

68. The grammar's rule list. `t_xrl_stack` lists the rules for the grammar, with their *Marpa_Rule_ID* as the index. The `rule_tree` is a tree for detecting duplicates.

⟨ Widely aligned grammar elements 59 ⟩ +≡

```
MARPA_DSTACK_DECLARE(t_xrl_stack);
MARPA_DSTACK_DECLARE(t_irl_stack);
```

69. ⟨ Initialize grammar elements 54 ⟩ +≡

```
MARPA_DSTACK_INIT2(g→t_xrl_stack, RULE);
MARPA_DSTACK_SAFE(g→t_irl_stack);
```

70. ⟨ Destroy grammar elements 61 ⟩ +≡

```
MARPA_DSTACK_DESTROY(g→t_irl_stack);
MARPA_DSTACK_DESTROY(g→t_xrl_stack);
```

71. Rule count accessors.

72.
```
#define XRL_Count_of_G(g) (MARPA_DSTACK_LENGTH((g)→t_xrl_stack))
```

73.
```
#define IRL_Count_of_G(g) (MARPA_DSTACK_LENGTH((g)→t_irl_stack))
```

74. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_highest_rule_id(Marpa_Grammar g)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  return XRL_Count_of_G(g) − 1;
}
```

```
int _marpa_g_irl_count(Marpa_Grammar g)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  return IRL_Count_of_G(g);
}
```

75. Internal accessor to find a rule by its id.

```
#define XRL_by_ID(id) (*MARPA_DSTACK_INDEX((g)→t_xrl_stack, XRL, (id)))
#define IRL_by_ID(id) (*MARPA_DSTACK_INDEX((g)→t_irl_stack, IRL, (id)))
```

76. Adds the rule to the list of rules kept by the Grammar object.

```
⟨ Function definitions 41 ⟩ +≡
  PRIVATE void rule_add(GRAMMAR g, RULE rule)
  {
    const RULEID new_id ⟸ MARPA_DSTACK_LENGTH((g)→t_xrl_stack);
    *MARPA_DSTACK_PUSH((g)→t_xrl_stack, RULE) ⟸ rule;
    rule→t_id ⟸ new_id;
    External_Size_of_G(g) += 1 + Length_of_XRL(rule);
    g→t_max_rule_length ⟸ MAX(Length_of_XRL(rule), g→t_max_rule_length);
  }
```

77. Check that rule is in valid range.

```
#define XRLID_is_Malformed(rule_id) ((rule_id) < 0)
#define XRLID_of_G_Exists(rule_id) ((rule_id) < XRL_Count_of_G(g))
#define IRLID_of_G_is_Valid(irl_id) ((irl_id) ≥ 0 ∧ (irl_id) < IRL_Count_of_G(g))
```

78. Start symbol.

```
⟨ Int aligned grammar elements 53 ⟩ +≡
  XSYIDt_start_xsy_id;
```

79. ⟨ Initialize grammar elements 54 ⟩ +≡

```
  g→t_start_xsy_id ⟸ −1;
```

80. ⟨ Function definitions 41 ⟩ +≡

```
  Marpa_Symbol_ID marpa_g_start_symbol(Marpa_Grammar g)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    if (g→t_start_xsy_id < 0) {
      MARPA_ERROR(MARPA_ERR_NO_START_SYMBOL);
      return −1;
    }
    return g→t_start_xsy_id;
  }
```

81. We return a soft failure on an attempt to set the start symbol to a non-existent symbol. The idea with other methods is they can act as a test for a non-existent symbol. That does not really make sense here, but we let consistency prevail.

⟨Function definitions 41⟩ +≡

```
Marpa_Symbol_ID marpa_g_start_symbol_set(Marpa_Grammar g, Marpa_Symbol_ID
        xsy_id)
{
  ⟨Return −2 on failure 1229⟩
  ⟨Fail if fatal error 1249⟩
  ⟨Fail if precomputed 1230⟩
  ⟨Fail if xsy_id is malformed 1232⟩
  ⟨Soft fail if xsy_id does not exist 1233⟩
  return g→t_start_xsy_id ⟸ xsy_id;
}
```

82. Start rules. These are the start rules, after the grammar is augmented. Only one of these needs to be non-NULL. A productive grammar with no proper start rule is considered trivial.

```
#define G_is_Trivial(g) (¬(g)→t_start_irl)
```

⟨Int aligned grammar elements 53⟩ +≡

```
IRLt_start_irl;
```

83. ⟨Initialize grammar elements 54⟩ +≡

```
g→t_start_irl ⟸ Λ;
```

84. The grammar's size. Intuitively, I define a grammar's size as the total size, in symbols, of all of its rules. This includes both the LHS symbol and the RHS symbol. Since every rule has exactly one LHS symbol, the grammar's size is always equal to the total of all the rules lengths, plus the total number of rules.

```
#define External_Size_of_G(g) ((g)→t_external_size)
```

85. ⟨Int aligned grammar elements 53⟩ +≡

```
int t_external_size;
```

86. ⟨Initialize grammar elements 54⟩ +≡

```
External_Size_of_G(g) ⟸ 0;
```

87. The maximum rule length. This is a high-ball estimate of the length of the longest rule in the grammar. The actual value will always be this number or smaller.

The value is used for allocating resources. Unused rules are not included in the theoretical number, but Marpa does not adjust this number as rules are marked useless.

88. ⟨Int aligned grammar elements 53⟩ +≡

```
int t_max_rule_length;
```

89. ⟨ Initialize grammar elements 54 ⟩ +≡

```
g→t_max_rule_length ⇐ 0;
```

90. The default rank. The default rank for rules and symbols. For minimum rank we want negative numbers rounded toward 0, not down.

91. *#define* MAXIMUM_RANK (INT_MAX/4)
#define MINIMUM_RANK (INT_MIN/4 + (INT_MIN % 4 > 0 ? 1 : 0))

⟨ Public typedefs 91 ⟩ ≡

```
typedef int Marpa_Rank;
```

See also sections 108, 134, 141, 215, 253, 327, 452, 533, 624, 626, 649, 668, 874, 928, 1013, 1111, and 1259.
This code is used in section 1387.

92. *#define* Default_Rank_of_G(g) ((g)→t_default_rank)

⟨ Int aligned grammar elements 53 ⟩ +≡

```
Marpa_Rank t_default_rank;
```

93. ⟨ Initialize grammar elements 54 ⟩ +≡

```
g→t_default_rank ⇐ 0;
```

94. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Rank marpa_g_default_rank(Marpa_Grammar g)
{
  ⟨ Return −2 on failure 1229 ⟩
  clear_error(g);
  ⟨ Fail if fatal error 1249 ⟩
  return Default_Rank_of_G(g);
}
```

95. Returns the symbol ID on success, −2 on failure.

⟨ Function definitions 41 ⟩ +≡

```
Marpa_Rank marpa_g_default_rank_set(Marpa_Grammar g, Marpa_Rank rank)
{
  ⟨ Return −2 on failure 1229 ⟩
  clear_error(g);
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if precomputed 1230 ⟩
  if (_MARPA_UNLIKELY(rank < MINIMUM_RANK)) {
    MARPA_ERROR(MARPA_ERR_RANK_TOO_LOW);
    return failure_indicator;
  }
  if (_MARPA_UNLIKELY(rank > MAXIMUM_RANK)) {
    MARPA_ERROR(MARPA_ERR_RANK_TOO_HIGH);
    return failure_indicator;
  }
  return Default_Rank_of_G(g) ⇐ rank;
}
```

96. Grammar is precomputed?.

97. *#define* `G_is_Precomputed`(g) ((g)→`t_is_precomputed`)

⟨ Bit aligned grammar elements 97 ⟩ ≡
  BITFIELD `t_is_precomputed`:1;

See also section 100.
This code is used in section 48.

98. ⟨ Initialize grammar elements 54 ⟩ +≡
  g→`t_is_precomputed` ⟸ 0;

99. ⟨ Function definitions 41 ⟩ +≡
  int `marpa_g_is_precomputed`(*Marpa_Grammar* g){ ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    return `G_is_Precomputed`(g); }

100. Grammar has loop?.

⟨ Bit aligned grammar elements 97 ⟩ +≡
  BITFIELD `t_has_cycle`:1;

101. ⟨ Initialize grammar elements 54 ⟩ +≡
  g→`t_has_cycle` ⟸ 0;

102. ⟨ Function definitions 41 ⟩ +≡
  int `marpa_g_has_cycle`(*Marpa_Grammar* g){ ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    return g→`t_has_cycle`; }

103. Terminal boolean vector. A boolean vector, with bits set if the symbol is a terminal. This is not used as the working vector while doing the census, because not all symbols have been added at that point. At grammar initialization, this vector cannot be sized. It is initialized to Λ so that the destructor can tell if there is a boolean vector to be freed.

⟨ Widely aligned grammar elements 59 ⟩ +≡
  Bit_Vector `t_bv_nsyid_is_terminal`;

104. ⟨ Initialize grammar elements 54 ⟩ +≡
  g→`t_bv_nsyid_is_terminal` ⟸ Λ;

105. Event boolean vectors. A boolean vector, with bits set if there is an event on completion of a rule with that symbol on the LHS. At grammar initialization, this vector cannot be sized. It is initialized to Λ so that the destructor can tell if there is a boolean vector to be freed.

⟨ Widely aligned grammar elements 59 ⟩ +≡
  Bit_Vector `t_lbv_xsyid_is_completion_event`;
  Bit_Vector `t_lbv_xsyid_completion_event_starts_active`;

```
Bit_Vector t_lbv_xsyid_is_nulled_event;
Bit_Vector t_lbv_xsyid_nulled_event_starts_active;
Bit_Vector t_lbv_xsyid_is_prediction_event;
Bit_Vector t_lbv_xsyid_prediction_event_starts_active;
```

106. ⟨ Initialize grammar elements 54 ⟩ +≡

```
g→t_lbv_xsyid_is_completion_event ⇐= Λ;
g→t_lbv_xsyid_completion_event_starts_active ⇐= Λ;
g→t_lbv_xsyid_is_nulled_event ⇐= Λ;
g→t_lbv_xsyid_nulled_event_starts_active ⇐= Λ;
g→t_lbv_xsyid_is_prediction_event ⇐= Λ;
g→t_lbv_xsyid_prediction_event_starts_active ⇐= Λ;
```

107. The event stack. Events are designed to be fast, but are at the moment not expected to have high volumes of data. The memory used is that of the high water mark, with no way of freeing it.

⟨ Private incomplete structures 107 ⟩ ≡

```
struct s_g_event;
typedef struct s_g_event *GEV;
```

See also sections 143, 454, 528, 535, 628, 650, 660, 663, 698, 855, 876, 904, 930, 936, 946, 1015, 1021, 1069, 1179, 1185, 1205, and 1207.

This code is used in section 1381.

108. ⟨ Public typedefs 91 ⟩ +≡

```
struct marpa_event;
typedef int Marpa_Event_Type;
```

109. ⟨ Public defines 109 ⟩ ≡

```
#define marpa_g_event_value(event) ((event)→t_value)
```

See also sections 295, 299, 1073, 1350, and 1359.

This code is used in section 1387.

110. ⟨ Public structures 44 ⟩ +≡

```
struct marpa_event {
  Marpa_Event_Type t_type;
  int t_value;
};
typedef struct marpa_event Marpa_Event;
```

111. ⟨ Private structures 48 ⟩ +≡

```
struct s_g_event {
  int t_type;
  int t_value;
};
typedef struct s_g_event GEV_Object;
```

112. *#define* `G_EVENT_COUNT(g) MARPA_DSTACK_LENGTH((g)→t_events)`

⟨ Widely aligned grammar elements 59 ⟩ +≡
```
MARPA_DSTACK_DECLARE(t_events);
```

113.

#define `INITIAL_G_EVENTS_CAPACITY (1024/sizeof(int))`

⟨ Initialize grammar elements 54 ⟩ +≡
```
MARPA_DSTACK_INIT(g→t_events, GEV_Object, INITIAL_G_EVENTS_CAPACITY);
```

114. ⟨ Destroy grammar elements 61 ⟩ +≡
```
MARPA_DSTACK_DESTROY(g→t_events);
```

115. Callers must be careful. A pointer to the new event is returned, but it must be written to before another event is added, because that may cause the locations of *MARPA_DSTACK* elements to change.

#define `G_EVENTS_CLEAR(g) MARPA_DSTACK_CLEAR((g)→t_events)`
#define `G_EVENT_PUSH(g) MARPA_DSTACK_PUSH((g)→t_events, GEV_Object)`

116. ⟨ Function definitions 41 ⟩ +≡
```
PRIVATE void event_new(GRAMMAR g, int type)
{
    /* may change base of dstack */
  GEV end_of_stack ⇐ G_EVENT_PUSH(g);
  end_of_stack→t_type ⇐ type;
  end_of_stack→t_value ⇐ 0;
}
```

117. ⟨ Function definitions 41 ⟩ +≡
```
PRIVATE void int_event_new(GRAMMAR g, int type, int value)
{   /* may change base of dstack */

GEV end_of_stack ⇐ G_EVENT_PUSH(g);
  end_of_stack→t_type ⇐ type;
  end_of_stack→t_value ⇐ value;
}
```

118. ⟨ Function definitions 41 ⟩ +≡
```
Marpa_Event_Type marpa_g_event(Marpa_Grammar g, Marpa_Event
        *public_event, int ix)
{
  ⟨ Return −2 on failure 1229 ⟩
  MARPA_DSTACK events ⇐ &g→t_events;
  GEV internal_event;
  int type;
```

```
    if (ix < 0) {
      MARPA_ERROR(MARPA_ERR_EVENT_IX_NEGATIVE);
      return failure_indicator;
    }
    if (ix ≥ MARPA_DSTACK_LENGTH(*events)) {
      MARPA_ERROR(MARPA_ERR_EVENT_IX_OOB);
      return failure_indicator;
    }
    internal_event ⟸ MARPA_DSTACK_INDEX(*events, GEV_Object, ix);
    type ⟸ internal_event→t_type;
    public_event→t_type ⟸ type;
    public_event→t_value ⟸ internal_event→t_value;
    return type;
  }
```

119. ⟨ Function definitions 41 ⟩ +≡

```
  Marpa_Event_Type marpa_g_event_count(Marpa_Grammar g)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    return MARPA_DSTACK_LENGTH(g→t_events);
  }
```

120. The rule duplication tree. This AVL tree is kept, before precomputation, to help detect BNF rules.

⟨ Widely aligned grammar elements 59 ⟩ +≡

```
  MARPA_AVL_TREE t_xrl_tree;
```

121. ⟨ Initialize grammar elements 54 ⟩ +≡

```
  (g)→t_xrl_tree ⟸ _marpa_avl_create(duplicate_rule_cmp, Λ);
```

122. ⟨ Clear rule duplication tree 122 ⟩ ≡

```
  {
    _marpa_avl_destroy((g)→t_xrl_tree);
    (g)→t_xrl_tree ⟸ Λ;
  }
```

This code is used in sections 123, 368, and 541.

123. ⟨ Destroy grammar elements 61 ⟩ +≡

```
  ⟨ Clear rule duplication tree 122 ⟩
```

124. The grammar obstacks. Obstacks with the same lifetime as the grammar. This is a very efficient way of allocating memory which won't be resized and which will have the same lifetime as the grammar. The XRL obstack is dedicated to XRL's, which it is convenient to build on the obstack. A dedicated obstack ensures that an in-process

XRL will not be overwritten by code using the obstack for other objects. A side benefit is that the dedicated XRL obstack can be specially aligned.

The method obstack is intended for temporaries that are used in external methods. Data in this obstack exists for the life of the method call. This obstack is cleared on exit from a method.

⟨ Widely aligned grammar elements 59 ⟩ +≡
```
struct marpa_obstack *t_obs;
struct marpa_obstack *t_xrl_obs;
```

125. ⟨ Initialize grammar elements 54 ⟩ +≡
```
g→t_obs ⟸ marpa_obs_init;
g→t_xrl_obs ⟸ marpa_obs_init;
```

126. ⟨ Destroy grammar elements 61 ⟩ +≡
```
marpa_obs_free(g→t_obs);
marpa_obs_free(g→t_xrl_obs);
```

127. The grammar constant integer list arena. Keeps constant integer lists with the same lifetime as the grammar. This arena is one of the grammar objects shared by all objects based on this grammar, something to be noted if grammars are ever to be shared by multiple threads.

⟨ Widely aligned grammar elements 59 ⟩ +≡
```
CILAR_Objectt_cilar;
```

128. ⟨ Initialize grammar elements 54 ⟩ +≡
```
cilar_init(&(g)→t_cilar);
```

129. ⟨ Destroy grammar elements 61 ⟩ +≡
```
cilar_destroy(&(g)→t_cilar);
```

130. The "is OK" word.

131. To Do: I probably should delete this. I don't use it in the SLIF.

132. The grammar needs a flag for a fatal error. This is an *int* for defensive coding reasons. Since I am paying the code of an *int*, I also use this word as a sanity test — testing that arguments that are passed as Marpa grammars actually do point to properly initialized Marpa grammars. It is also possible this will catch certain memory overwrites.

133. The word is placed first, because references to the first word of a bogus pointer are the most likely to be handled without a memory access error. Also, there it is somewhat more likely to catch memory overwrite errors. #69734f4b is the ASCII for 'isOK'.

```
#define I_AM_OK #69734f4b
#define IS_G_OK(g) ((g)→t_is_ok ≡ I_AM_OK)
```

⟨ First grammar element 133 ⟩ ≡
```
int t_is_ok;
```

This code is used in section 48.

134. The grammar's error ID. This is an error flag for the grammar. Error status is not necessarily cleared on successful return, so that it is only valid when an external function has indicated there is an error, and becomes invalid again when another external method is called on the grammar. Checking it at other times may reveal "stale" error messages.

⟨ Public typedefs 91 ⟩ +≡
```
typedef int Marpa_Error_Code;
```

135. ⟨ Widely aligned grammar elements 59 ⟩ +≡
```
const char *t_error_string;
```

136. ⟨ Int aligned grammar elements 53 ⟩ +≡
```
Marpa_Error_Code t_error;
```

137. ⟨ Initialize grammar elements 54 ⟩ +≡
```
g→t_error ⟸ MARPA_ERR_NONE;
g→t_error_string ⟸ Λ;
```

138. There is no destructor. The error strings are assummed to be **not** error messages, but "cookies". These cookies are constants residing in static memory (which may be read-only depending on implementation). They cannot and should not be de-allocated.

139. As a side effect, the current error is cleared if it is non=fatal.

⟨ Function definitions 41 ⟩ +≡
```
Marpa_Error_Code marpa_g_error(Marpa_Grammar g, const char **p_error_string)
{
  const Marpa_Error_Code error_code ⟸ g→t_error;
  const char *error_string ⟸ g→t_error_string;
  if (p_error_string) {
    *p_error_string ⟸ error_string;
  }
  return error_code;
}
```

140. ⟨ Function definitions 41 ⟩ +≡
```
Marpa_Error_Code marpa_g_error_clear(Marpa_Grammar g)
{
  clear_error(g);
  return g→t_error;
}
```

141. Symbol (XSY) code.

⟨ Public typedefs 91 ⟩ +≡
typedef int Marpa_Symbol_ID;

142. ⟨ Private typedefs 49 ⟩ +≡
typedef Marpa_Symbol_ID XSYID;

143. ⟨ Private incomplete structures 107 ⟩ +≡
struct s_xsy;
typedef struct s_xsy ∗*XSY*;
typedef const struct s_xsy ∗*XSY_Const*;

144. ⟨ Private structures 48 ⟩ +≡
struct s_xsy {
  ⟨ Widely aligned XSY elements 202 ⟩
  ⟨ Int aligned XSY elements 145 ⟩
  ⟨ Bit aligned XSY elements 154 ⟩
};

145. ID.

#define `ID_of_XSY(xsy)` `((xsy)→t_symbol_id)`

⟨ Int aligned XSY elements 145 ⟩ ≡
XSYID `t_symbol_id`;

See also section 150.

This code is used in section 144.

146. ⟨ Function definitions 41 ⟩ +≡
PRIVATE XSY `symbol_new`(*GRAMMAR g*)
{
  XSY `xsy` ⟸ `marpa_obs_new`(*g*→`t_obs`, *struct s_xsy*, 1);
  ⟨ Initialize XSY elements 151 ⟩
  `symbol_add`(*g*, `xsy`);
  return `xsy`;
}

147. ⟨ Function definitions 41 ⟩ +≡
Marpa_Symbol_ID `marpa_g_symbol_new`(*Marpa_Grammar g*)
{
  const XSY `symbol` ⟸ `symbol_new`(*g*);
  return `ID_of_XSY`(`symbol`);
}

148. Symbol is start?.

149. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_is_start(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  if (g→t_start_xsy_id < 0) return 0;
  return xsy_id ≡ g→t_start_xsy_id ? 1 : 0;
}
```

150. Symbol rank.

⟨ Int aligned XSY elements 145 ⟩ +≡

```
Marpa_Rank t_rank;
```

151. ⟨ Initialize XSY elements 151 ⟩ ≡

```
xsy→t_rank ⟸ Default_Rank_of_G(g);
```

See also sections 155, 157, 159, 167, 170, 173, 176, 179, 184, 187, 192, 197, 203, 206, and 210.

This code is used in section 146.

152. *#define* Rank_of_XSY(symbol) ((symbol)→t_rank)

⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_rank(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
{
  XSY xsy;
  ⟨ Return −2 on failure 1229 ⟩
  clear_error(g);
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Fail if xsy_id does not exist 1234 ⟩
  xsy ⟸ XSY_by_ID(xsy_id);
  return Rank_of_XSY(xsy);
}
```

153. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_rank_set(Marpa_Grammar g, Marpa_Symbol_ID
        xsy_id, Marpa_Rank rank)
{
  XSY xsy;
  ⟨ Return −2 on failure 1229 ⟩
  clear_error(g);
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if precomputed 1230 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Fail if xsy_id does not exist 1234 ⟩
```

```
    xsy ⟸ XSY_by_ID(xsy_id);
    if (_MARPA_UNLIKELY(rank < MINIMUM_RANK)) {
      MARPA_ERROR(MARPA_ERR_RANK_TOO_LOW);
      return failure_indicator;
    }
    if (_MARPA_UNLIKELY(rank > MAXIMUM_RANK)) {
      MARPA_ERROR(MARPA_ERR_RANK_TOO_HIGH);
      return failure_indicator;
    }
    return Rank_of_XSY(xsy) ⟸ rank;
  }
```

154. Symbol is LHS?. Is this (external) symbol on the LHS of any rule, whether sequence or BNF.

```
#define XSY_is_LHS(xsy) ((xsy)→t_is_lhs)
```

⟨ Bit aligned XSY elements 154 ⟩ ≡

```
  BITFIELD t_is_lhs:1;
```

See also sections 156, 158, 166, 169, 172, 175, 178, 183, 186, 191, and 196.

This code is used in section 144.

155. ⟨ Initialize XSY elements 151 ⟩ +≡

```
  XSY_is_LHS(xsy) ⟸ 0;
```

156. Symbol is sequence LHS?. Is this (external) symbol on the LHS of a sequence rule?

```
#define XSY_is_Sequence_LHS(xsy) ((xsy)→t_is_sequence_lhs)
```

⟨ Bit aligned XSY elements 154 ⟩ +≡

```
  BITFIELD t_is_sequence_lhs:1;
```

157. ⟨ Initialize XSY elements 151 ⟩ +≡

```
  XSY_is_Sequence_LHS(xsy) ⟸ 0;
```

158. Nulling symbol is valued?. This value describes the semantics for a symbol when it is nulling. Marpa optimizes for the case where the application does not care about the value of a symbol – that is, the semantics is arbitrary.

```
#define XSY_is_Valued(symbol) ((symbol)→t_is_valued)
#define XSY_is_Valued_Locked(symbol) ((symbol)→t_is_valued_locked)
```

⟨ Bit aligned XSY elements 154 ⟩ +≡

```
  BITFIELD t_is_valued:1;
  BITFIELD t_is_valued_locked:1;
```

159. ⟨ Initialize XSY elements 151 ⟩ +≡

```
  XSY_is_Valued(xsy) ⟸ g→t_force_valued ? 1 : 0;
  XSY_is_Valued_Locked(xsy) ⟸ g→t_force_valued ? 1 : 0;
```

160. Force all symbols to be valued. Unvalued symbols are deprecated, so that this will be the default, going forward.

161. ⟨ Int aligned grammar elements 53 ⟩ +≡

```
int t_force_valued;
```

162. ⟨ Initialize grammar elements 54 ⟩ +≡

```
g→t_force_valued ⇐ 0;
```

163. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_force_valued(Marpa_Grammar g)
{
  XSYID xsyid;
  ⟨ Return −2 on failure 1229 ⟩
  for (xsyid ⇐ 0; xsyid < XSY_Count_of_G(g); xsyid++) {
    const XSY xsy ⇐ XSY_by_ID(xsyid);
    if (¬XSY_is_Valued(xsy) ∧ XSY_is_Valued_Locked(xsy)) {
      MARPA_ERROR(MARPA_ERR_VALUED_IS_LOCKED);
      return failure_indicator;
    }
    XSY_is_Valued(xsy) ⇐ 1;
    XSY_is_Valued_Locked(xsy) ⇐ 1;
  }
  g→t_force_valued ⇐ 1;
  return 0;
}
```

164. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_is_valued(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  return XSY_is_Valued(XSY_by_ID(xsy_id));
}
```

165. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_is_valued_set(Marpa_Grammar g, Marpa_Symbol_ID xsy_id, int
        value)
{
  XSY symbol;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  symbol ⇐ XSY_by_ID(xsy_id);
```

```
    if (_MARPA_UNLIKELY(value < 0 ∨ value > 1)) {
      MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
      return failure_indicator;
    }
    if (_MARPA_UNLIKELY(XSY_is_Valued_Locked(symbol) ∧ value ≠
          XSY_is_Valued(symbol))) {
      MARPA_ERROR(MARPA_ERR_VALUED_IS_LOCKED);
      return failure_indicator;
    }
    XSY_is_Valued(symbol) ⟸ Boolean(value);
    return value;
  }
```

166. Symbol is accessible?.

```
#define XSY_is_Accessible(xsy) ((xsy)→t_is_accessible)
```

⟨ Bit aligned XSY elements 154 ⟩ +≡

```
  BITFIELD t_is_accessible:1;
```

167. ⟨ Initialize XSY elements 151 ⟩ +≡

```
  xsy→t_is_accessible ⟸ 0;
```

168. The trace accessor returns the boolean value. Right now this function uses a pointer to the symbol function. If that becomes private, the prototype of this function must be changed.

⟨ Function definitions 41 ⟩ +≡

```
  int marpa_g_symbol_is_accessible(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if not precomputed 1231 ⟩
    ⟨ Fail if xsy_id is malformed 1232 ⟩
    ⟨ Soft fail if xsy_id does not exist 1233 ⟩
    return XSY_is_Accessible(XSY_by_ID(xsy_id));
  }
```

169. Symbol is counted?.

⟨ Bit aligned XSY elements 154 ⟩ +≡

```
  BITFIELD t_is_counted:1;
```

170. ⟨ Initialize XSY elements 151 ⟩ +≡

```
  xsy→t_is_counted ⟸ 0;
```

171. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_is_counted(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  return XSY_by_ID(xsy_id)→t_is_counted;
}
```

172. Symbol is nulling?.

```
#define XSY_is_Nulling(sym) ((sym)→t_is_nulling)
```

⟨ Bit aligned XSY elements 154 ⟩ +≡

```
BITFIELD t_is_nulling:1;
```

173. ⟨ Initialize XSY elements 151 ⟩ +≡

```
xsy→t_is_nulling ⇐ 0;
```

174. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_is_nulling(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  return XSY_is_Nulling(XSY_by_ID(xsy_id));
}
```

175. Symbol is nullable?.

```
#define XSY_is_Nullable(xsy) ((xsy)→t_is_nullable)
#define XSYID_is_Nullable(xsyid) XSY_is_Nullable(XSY_by_ID(xsyid))
```

⟨ Bit aligned XSY elements 154 ⟩ +≡

```
BITFIELD t_is_nullable:1;
```

176. ⟨ Initialize XSY elements 151 ⟩ +≡

```
xsy→t_is_nullable ⇐ 0;
```

177. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_is_nullable(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
```

```
    ⟨ Soft fail if xsy_id does not exist 1233 ⟩
    return XSYID_is_Nullable(xsy_id);
  }
```

178. Symbol is terminal?. The "locked terminal" flag tracked whether the terminal flag was set by the user. It distinguishes those terminal settings that will be overwritten by the default from those should not be.

```
⟨ Bit aligned XSY elements 154 ⟩ +≡
  BITFIELD t_is_terminal:1;
  BITFIELD t_is_locked_terminal:1;
```

179. ⟨ Initialize XSY elements 151 ⟩ +≡

```
  xsy→t_is_terminal ⟸ 0;
  xsy→t_is_locked_terminal ⟸ 0;
```

180. *#define* `XSY_is_Terminal(xsy) ((xsy)→t_is_terminal)`

181. *#define* `XSY_is_Locked_Terminal(xsy) ((xsy)→t_is_locked_terminal)`
#define `XSYID_is_Terminal(id) (XSY_is_Terminal(XSY_by_ID(id)))`

```
⟨ Function definitions 41 ⟩ +≡
  int marpa_g_symbol_is_terminal(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if xsy_id is malformed 1232 ⟩
    ⟨ Soft fail if xsy_id does not exist 1233 ⟩
    return XSYID_is_Terminal(xsy_id);
  }
```

182. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_g_symbol_is_terminal_set(Marpa_Grammar g, Marpa_Symbol_ID
          xsy_id, int value)
  {
    XSY symbol;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if precomputed 1230 ⟩
    ⟨ Fail if xsy_id is malformed 1232 ⟩
    ⟨ Soft fail if xsy_id does not exist 1233 ⟩
    symbol ⟸ XSY_by_ID(xsy_id);
    if (_MARPA_UNLIKELY(value < 0 ∨ value > 1)) {
      MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
      return failure_indicator;
    }
```

```
    if (_MARPA_UNLIKELY(XSY_is_Locked_Terminal(symbol)) ∧
          XSY_is_Terminal(symbol) ≠ value) {
      MARPA_ERROR(MARPA_ERR_TERMINAL_IS_LOCKED);
      return failure_indicator;
    }
    XSY_is_Locked_Terminal(symbol) ⇐ 1;
    return XSY_is_Terminal(symbol) ⇐ Boolean(value);
  }
```

183. XSY is productive?.

```
#define XSY_is_Productive(xsy) ((xsy)→t_is_productive)
```

⟨ Bit aligned XSY elements 154 ⟩ +≡

```
  BITFIELD t_is_productive:1;
```

184. ⟨ Initialize XSY elements 151 ⟩ +≡

```
  xsy→t_is_productive ⇐ 0;
```

185. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_g_symbol_is_productive(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if not precomputed 1231 ⟩
    ⟨ Fail if xsy_id is malformed 1232 ⟩
    ⟨ Soft fail if xsy_id does not exist 1233 ⟩
    return XSY_is_Productive(XSY_by_ID(xsy_id));
  }
```

186. XSY is completion event?.

```
#define XSY_is_Completion_Event(xsy) ((xsy)→t_is_completion_event)
#define XSYID_is_Completion_Event(xsyid)
          XSY_is_Completion_Event(XSY_by_ID(xsyid))
#define XSY_Completion_Event_Starts_Active(xsy)
          ((xsy)→t_completion_event_starts_active)
#define XSYID_Completion_Event_Starts_Active(xsyid)
          XSY_Completion_Event_Starts_Active(XSY_by_ID(xsyid))
```

⟨ Bit aligned XSY elements 154 ⟩ +≡

```
  BITFIELD t_is_completion_event:1;
  BITFIELD t_completion_event_starts_active:1;
```

187. ⟨ Initialize XSY elements 151 ⟩ +≡

```
  xsy→t_is_completion_event ⇐ 0;
  xsy→t_completion_event_starts_active ⇐ 0;
```

188. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_is_completion_event(Marpa_Grammar g, Marpa_Symbol_ID
        xsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  return XSYID_is_Completion_Event(xsy_id);
}
```

189. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_is_completion_event_set(Marpa_Grammar g, Marpa_Symbol_ID
        xsy_id, int value)
{
  XSY xsy;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if precomputed 1230 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  xsy ⟸ XSY_by_ID(xsy_id);
  switch (value) {
  case 0: case 1: XSY_Completion_Event_Starts_Active(xsy) ⟸ Boolean(value);
    return XSY_is_Completion_Event(xsy) ⟸ Boolean(value);
  }
  MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
  return failure_indicator;
}
```

190. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_completion_symbol_activate(Marpa_Grammar g, Marpa_Symbol_ID
        xsy_id, int reactivate)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if precomputed 1230 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  switch (reactivate) {
  case 0:
    XSYID_Completion_Event_Starts_Active(xsy_id) ⟸ Boolean(reactivate);
    return 0;
  case 1:
```

```
      if (¬XSYID_is_Completion_Event(xsy_id)) {      /* An attempt to activate a
            completion event on a symbol which was not set up for them. */
        MARPA_ERROR(MARPA_ERR_SYMBOL_IS_NOT_COMPLETION_EVENT);
      }
      XSYID_Completion_Event_Starts_Active(xsy_id) ⟸ Boolean(reactivate);
      return 1;
    }
    MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
    return failure_indicator;
  }
```

191. XSY is nulled event?.

```
#define XSY_is_Nulled_Event(xsy) ((xsy)→t_is_nulled_event)
#define XSYID_is_Nulled_Event(xsyid) XSY_is_Nulled_Event(XSY_by_ID(xsyid))
#define XSY_Nulled_Event_Starts_Active(xsy)
        ((xsy)→t_nulled_event_starts_active)
#define XSYID_Nulled_Event_Starts_Active(xsyid)
        XSY_Nulled_Event_Starts_Active(XSY_by_ID(xsyid))
```

⟨ Bit aligned XSY elements 154 ⟩ +≡

```
  BITFIELD t_is_nulled_event:1;
  BITFIELD t_nulled_event_starts_active:1;
```

192. ⟨ Initialize XSY elements 151 ⟩ +≡

```
  xsy→t_is_nulled_event ⟸ 0;
  xsy→t_nulled_event_starts_active ⟸ 0;
```

193. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_g_symbol_is_nulled_event(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if xsy_id is malformed 1232 ⟩
    ⟨ Soft fail if xsy_id does not exist 1233 ⟩
    return XSYID_is_Nulled_Event(xsy_id);
  }
```

194. Does not check if the symbol is actually nullable – this is by design.

⟨ Function definitions 41 ⟩ +≡

```
  int marpa_g_symbol_is_nulled_event_set(Marpa_Grammar g, Marpa_Symbol_ID
          xsy_id, int value)
  {
    XSY xsy;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if precomputed 1230 ⟩
```

```
    ⟨ Fail if xsy_id is malformed 1232 ⟩
    ⟨ Soft fail if xsy_id does not exist 1233 ⟩
    xsy ⟸ XSY_by_ID(xsy_id);
    switch (value) {
    case 0: case 1: XSY_Nulled_Event_Starts_Active(xsy) ⟸ Boolean(value);
      return XSY_is_Nulled_Event(xsy) ⟸ Boolean(value);
    }
    MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
    return failure_indicator;
  }
```

195. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_g_nulled_symbol_activate(Marpa_Grammar g, Marpa_Symbol_ID
          xsy_id, int reactivate)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if precomputed 1230 ⟩
    ⟨ Fail if xsy_id is malformed 1232 ⟩
    ⟨ Soft fail if xsy_id does not exist 1233 ⟩
    switch (reactivate) {
    case 0: XSYID_Nulled_Event_Starts_Active(xsy_id) ⟸ Boolean(reactivate);
      return 0;
    case 1:
      if (¬XSYID_is_Nulled_Event(xsy_id)) {    /* An attempt to activate a nulled
            event on a symbol which was not set up for them. */
        MARPA_ERROR(MARPA_ERR_SYMBOL_IS_NOT_COMPLETION_EVENT);
      }
      XSYID_Nulled_Event_Starts_Active(xsy_id) ⟸ Boolean(reactivate);
      return 1;
    }
    MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
    return failure_indicator;
  }
```

196. XSY is prediction event?.

```
#define XSY_is_Prediction_Event(xsy) ((xsy)→t_is_prediction_event)
#define XSYID_is_Prediction_Event(xsyid)
        XSY_is_Prediction_Event(XSY_by_ID(xsyid))
#define XSY_Prediction_Event_Starts_Active(xsy)
        ((xsy)→t_prediction_event_starts_active)
#define XSYID_Prediction_Event_Starts_Active(xsyid)
        XSY_Prediction_Event_Starts_Active(XSY_by_ID(xsyid))
```

⟨ Bit aligned XSY elements 154 ⟩ +≡

```
  BITFIELD t_is_prediction_event:1;
```

BITFIELD `t_prediction_event_starts_active`:1;

197. ⟨ Initialize XSY elements 151 ⟩ +≡

```
xsy→t_is_prediction_event ⇐ 0;
xsy→t_prediction_event_starts_active ⇐ 0;
```

198. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_is_prediction_event(Marpa_Grammar g, Marpa_Symbol_ID
        xsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  return XSYID_is_Prediction_Event(xsy_id);
}
```

199. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_symbol_is_prediction_event_set(Marpa_Grammar g, Marpa_Symbol_ID
        xsy_id, int value)
{
  XSY xsy;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if precomputed 1230 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  xsy ⇐ XSY_by_ID(xsy_id);
  switch (value) {
  case 0: case 1: XSY_Prediction_Event_Starts_Active(xsy) ⇐ Boolean(value);
    return XSY_is_Prediction_Event(xsy) ⇐ Boolean(value);
  }
  MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
  return failure_indicator;
}
```

200. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_prediction_symbol_activate(Marpa_Grammar g, Marpa_Symbol_ID
        xsy_id, int reactivate)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if precomputed 1230 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
```

```
    switch (reactivate) {
    case 0:
      XSYID_Prediction_Event_Starts_Active(xsy_id) ⇐ Boolean(reactivate);
      return 0;
    case 1:
      if (¬XSYID_is_Prediction_Event(xsy_id)) {     /* An attempt to activate a
             prediction event on a symbol which was not set up for them. */
        MARPA_ERROR(MARPA_ERR_SYMBOL_IS_NOT_COMPLETION_EVENT);
      }
      XSYID_Prediction_Event_Starts_Active(xsy_id) ⇐ Boolean(reactivate);
      return 1;
    }
    MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
    return failure_indicator;
  }
```

201. ⟨ Function definitions 41 ⟩ +≡

202. Nulled XSYIDs.

```
#define Nulled_XSYIDs_of_XSY(xsy) ((xsy)→t_nulled_event_xsyids)
#define Nulled_XSYIDs_of_XSYID(xsyid) Nulled_XSYIDs_of_XSY(XSY_by_ID(xsyid))
```

⟨ Widely aligned XSY elements 202 ⟩ ≡

```
  CIL t_nulled_event_xsyids;
```

See also sections 205 and 209.

This code is used in section 144.

203. The nulled XSYIDs include all the symbols nullified by an XSY. A nullable symbol always nullifies itself. It may nullify additional XSY's through derivations of nulled rules. The issue of ambiguous derivations is dealt with by including all nulled derivations. If XSY `xsy1` can nullify XSY `xsy2`, then it does. For non-nullable XSY's, this will be the empty CIL. If there are no nulled events, the nulled event CIL's will be populated with the empty CIL.

⟨ Initialize XSY elements 151 ⟩ +≡

```
  Nulled_XSYIDs_of_XSY(xsy) ⇐ Λ;
```

204. Primary internal equivalent. This is the internal equivalent of the external symbol. If the external symbol is nullable it is the non-nullable NSY.

```
#define NSY_of_XSY(xsy) ((xsy)→t_nsy_equivalent)
#define NSYID_of_XSY(xsy) ID_of_NSY(NSY_of_XSY(xsy))
#define NSY_by_XSYID(xsy_id) (XSY_by_ID(xsy_id)→t_nsy_equivalent)
```

205. Note that it is up to the calling environment for `NSYID_by_XSYID(xsy_id)` to ensure that `NSY_of_XSY(xsy)` exists.

```
#define NSYID_by_XSYID(xsy_id) ID_of_NSY(NSY_of_XSY(XSY_by_ID(xsy_id)))
```

⟨ Widely aligned XSY elements 202 ⟩ +≡

```
  NSY t_nsy_equivalent;
```

206. ⟨ Initialize XSY elements 151 ⟩ +≡
```
NSY_of_XSY(xsy) ⟸ Λ;
```

207. ⟨ Function definitions 41 ⟩ +≡
```
Marpa_NSY_ID_marpa_g_xsy_nsy(Marpa_Grammar g, Marpa_Symbol_ID xsy_id)
{
  XSY xsy;
  NSY nsy;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  xsy ⟸ XSY_by_ID(xsy_id);
  nsy ⟸ NSY_of_XSY(xsy);
  return nsy ? ID_of_NSY(nsy) : −1;
}
```

208. Nulling internal equivalent. This is the nulling internal equivalent of the external symbol. If the external symbol is nullable it is the nulling NSY. If the external symbol is nulling it is the same as the primary internal equivalent. If the external symbol is non-nulling, there is no nulling internal equivalent.

```
#define Nulling_NSY_of_XSY(xsy) ((xsy)→t_nulling_nsy)
#define Nulling_NSY_by_XSYID(xsy) (XSY_by_ID(xsy)→t_nulling_nsy)
```

209. Note that it is up to the calling environment for `Nulling_NSYID_by_XSYID(xsy_id)` to ensure that `Nulling_NSY_of_XSY(xsy)` exists.

```
#define Nulling_NSYID_by_XSYID(xsy) ID_of_NSY(XSY_by_ID(xsy)→t_nulling_nsy)
```

⟨ Widely aligned XSY elements 202 ⟩ +≡
```
NSY t_nulling_nsy;
```

210. ⟨ Initialize XSY elements 151 ⟩ +≡
```
Nulling_NSY_of_XSY(xsy) ⟸ Λ;
```

211. ⟨ Function definitions 41 ⟩ +≡
```
Marpa_NSY_ID_marpa_g_xsy_nulling_nsy(Marpa_Grammar g, Marpa_Symbol_ID
        xsy_id)
{
  XSY xsy;
  NSY nsy;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  xsy ⟸ XSY_by_ID(xsy_id);
  nsy ⟸ Nulling_NSY_of_XSY(xsy);
  return nsy ? ID_of_NSY(nsy) : −1;
}
```

212. Given a proper nullable symbol as its argument, converts the argument into two "aliases". The proper (non-nullable) alias will have the same symbol ID as the arugment. The nulling alias will have a new symbol ID. The return value is a pointer to the nulling alias.

213. ⟨Function definitions 41⟩ +≡

```
PRIVATE NSY symbol_alias_create(GRAMMAR g, XSY symbol)
{
  NSY alias_nsy ⟸ semantic_nsy_new(g, symbol);
  XSY_is_Nulling(symbol) ⟸ 0;
  XSY_is_Nullable(symbol) ⟸ 1;
  NSY_is_Nulling(alias_nsy) ⟸ 1;
  return alias_nsy;
}
```

214. Internal symbols (NSY). This is the logic for keeping track of symbols created internally by libmarpa.

215. ⟨ Public typedefs 91 ⟩ +≡

```
typedef int Marpa_NSY_ID;
```

216. ⟨ Private typedefs 49 ⟩ +≡

```
struct s_nsy;
typedef struct s_nsy *NSY;
typedef Marpa_NSY_ID NSYID;
```

217. Internal symbols are also used as the or-nodes for nulling tokens. The initial element is a type *int*, and the next element is the symbol ID, (the unique identifier for the symbol), so that the symbol structure may be used where token or-nodes are expected.

```
#define Nulling_OR_by_NSYID(nsyid) ((OR) &NSY_by_ID(nsyid)→t_nulling_or_node)
#define Unvalued_OR_by_NSYID(nsyid)
        ((OR) &NSY_by_ID(nsyid)→t_unvalued_or_node)
```

⟨ Private structures 48 ⟩ +≡

```
struct s_unvalued_token_or_node {
  int t_or_node_type;
  NSYID t_nsyid;
};
struct s_nsy {
  ⟨ Widely aligned NSY elements 236 ⟩
  ⟨ Int aligned NSY elements 250 ⟩
  ⟨ Bit aligned NSY elements 227 ⟩
  struct s_unvalued_token_or_node t_nulling_or_node;
  struct s_unvalued_token_or_node t_unvalued_or_node;
};
```

218. `t_nsyid` is initialized when the symbol is added to the list of symbols. Symbols are used a nulling tokens, and `t_or_node_type` is set accordingly.

⟨ Initialize NSY elements 218 ⟩ ≡

```
nsy→t_nulling_or_node.t_or_node_type ⟸ NULLING_TOKEN_OR_NODE;
  /* ID of nulling or-node is already set */
nsy→t_unvalued_or_node.t_or_node_type ⟸ UNVALUED_TOKEN_OR_NODE;
nsy→t_unvalued_or_node.t_nsyid ⟸ ID_of_NSY(nsy);
```

See also sections 228, 231, 234, 237, 239, 242, 246, and 251.
This code is used in section 220.

219. Constructors.

220. Common logic for creating an NSY.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE NSY nsy_start(GRAMMAR g)
{
  const NSY nsy ⟸ marpa_obs_new(g→t_obs, struct s_nsy, 1);
  ID_of_NSY(nsy) ⟸ MARPA_DSTACK_LENGTH((g)→t_nsy_stack);
  *MARPA_DSTACK_PUSH((g)→t_nsy_stack, NSY) ⟸ nsy;
  ⟨ Initialize NSY elements 218 ⟩
  return nsy;
}
```

221. Create a virtual NSY from scratch. A source symbol must be specified.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE NSY nsy_new(GRAMMAR g, XSY source)
{
  const NSY new_nsy ⟸ nsy_start(g);
  Source_XSY_of_NSY(new_nsy) ⟸ source;
  Rank_of_NSY(new_nsy) ⟸ NSY_Rank_by_XSY(source);
  return new_nsy;
}
```

222. Create an semantically-visible NSY from scratch. A source symbol must be specified.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE NSY semantic_nsy_new(GRAMMAR g, XSY source)
{
  const NSY new_nsy ⟸ nsy_new(g, source);
  NSY_is_Semantic(new_nsy) ⟸ 1;
  return new_nsy;
}
```

223. Clone an NSY from an XSY. An XSY must be specified.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE NSY nsy_clone(GRAMMAR g, XSY xsy)
{
  const NSY new_nsy ⟸ nsy_start(g);
  Source_XSY_of_NSY(new_nsy) ⟸ xsy;
  NSY_is_Semantic(new_nsy) ⟸ 1;
  Rank_of_NSY(new_nsy) ⟸ NSY_Rank_by_XSY(xsy);
  NSY_is_Nulling(new_nsy) ⟸ XSY_is_Nulling(xsy);
  return new_nsy;
}
```

224. ID. The **NSY ID** is a number which acts as the unique identifier for an NSY. The NSY ID is initialized when the NSY is added to the list of rules.

```
#define NSY_by_ID(id) (*MARPA_DSTACK_INDEX(g→t_nsy_stack, NSY, (id)))
#define ID_of_NSY(nsy) ((nsy)→t_nulling_or_node.t_nsyid)
```

225. Symbol count accesors.

```
#define NSY_Count_of_G(g) (MARPA_DSTACK_LENGTH((g)→t_nsy_stack))
```

226. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_nsy_count(Marpa_Grammar g)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  return NSY_Count_of_G(g);
}
```

227. Is Start?.

```
#define NSY_is_Start(nsy) ((nsy)→t_is_start)
```

⟨ Bit aligned NSY elements 227 ⟩ ≡

```
  BITFIELD t_is_start:1;
```

See also sections 230, 233, and 238.

This code is used in section 217.

228. ⟨ Initialize NSY elements 218 ⟩ +≡

```
  NSY_is_Start(nsy) ⟸ 0;
```

229. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_nsy_is_start(Marpa_Grammar g, Marpa_NSY_ID nsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if nsy_id is invalid 1235 ⟩
  return NSY_is_Start(NSY_by_ID(nsy_id));
}
```

230. Is LHS?.

```
#define NSY_is_LHS(nsy) ((nsy)→t_is_lhs)
```

⟨ Bit aligned NSY elements 227 ⟩ +≡

```
  BITFIELD t_is_lhs:1;
```

231. ⟨ Initialize NSY elements 218 ⟩ +≡

```
  NSY_is_LHS(nsy) ⟸ 0;
```

232. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_nsy_is_lhs(Marpa_Grammar g, Marpa_NSY_ID nsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if nsy_id is invalid 1235 ⟩
  return NSY_is_LHS(NSY_by_ID(nsy_id));
}
```

233. NSY is nulling?.

```
#define NSY_is_Nulling(nsy) ((nsy)→t_nsy_is_nulling)
```

⟨ Bit aligned NSY elements 227 ⟩ +≡

```
BITFIELD t_nsy_is_nulling:1;
```

234. ⟨ Initialize NSY elements 218 ⟩ +≡

```
NSY_is_Nulling(nsy) ⟸ 0;
```

235. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_nsy_is_nulling(Marpa_Grammar g, Marpa_NSY_ID nsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if nsy_id is invalid 1235 ⟩
  return NSY_is_Nulling(NSY_by_ID(nsy_id));
}
```

236. LHS CIL. A CIL which records the IRL's of which this NSY is the LHS.

```
#define LHS_CIL_of_NSY(nsy) ((nsy)→t_lhs_cil)
#define LHS_CIL_of_NSYID(nsyid) LHS_CIL_of_NSY(NSY_by_ID(nsyid))
```

⟨ Widely aligned NSY elements 236 ⟩ ≡

```
CIL t_lhs_cil;
```

See also sections 241 and 245.

This code is used in section 217.

237. ⟨ Initialize NSY elements 218 ⟩ +≡

```
LHS_CIL_of_NSY(nsy) ⟸ Λ;
```

238. Semantic XSY. Set if the internal symbol is semantically visible externally.

```
#define NSY_is_Semantic(nsy) ((nsy)→t_is_semantic)
#define NSYID_is_Semantic(nsyid) (NSY_is_Semantic(NSY_by_ID(nsyid)))
```

⟨ Bit aligned NSY elements 227 ⟩ +≡

```
BITFIELD t_is_semantic:1;
```

239. ⟨ Initialize NSY elements 218 ⟩ +≡
```
NSY_is_Semantic(nsy) ⟸ 0;
```

240. ⟨ Function definitions 41 ⟩ +≡
```
int _marpa_g_nsy_is_semantic(Marpa_Grammar g, Marpa_IRL_ID nsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if nsy_id is invalid 1235 ⟩
  return NSYID_is_Semantic(nsy_id);
}
```

241. Source XSY. This is the external "source" of the internal symbol – the external symbol that it is derived from. There is always a non-null source XSY. It is used in ranking, and is also convenient for tracing and debugging.

```
#define Source_XSY_of_NSY(nsy) ((nsy)→t_source_xsy)
#define Source_XSY_of_NSYID(nsyid) (Source_XSY_of_NSY(NSY_by_ID(nsyid)))
#define Source_XSYID_of_NSYID(nsyid) ID_of_XSY(Source_XSY_of_NSYID(nsyid))
```

⟨ Widely aligned NSY elements 236 ⟩ +≡
```
XSY t_source_xsy;
```

242. ⟨ Initialize NSY elements 218 ⟩ +≡
```
Source_XSY_of_NSY(nsy) ⟸ Λ;
```

243. ⟨ Function definitions 41 ⟩ +≡
```
Marpa_Rule_ID _marpa_g_source_xsy(Marpa_Grammar g, Marpa_IRL_ID nsy_id)
{
  XSY source_xsy;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if nsy_id is invalid 1235 ⟩
  source_xsy ⟸ Source_XSY_of_NSYID(nsy_id);
  return ID_of_XSY(source_xsy);
}
```

244. Source rule and offset. In the case of sequences and CHAF rules, internal symbols are created to act as the LHS of internal rules. These fields record the symbol's source information with the symbol. The semantics need this information so that they can simulate the external "source" rule.

245.
```
#define LHS_XRL_of_NSY(nsy) ((nsy)→t_lhs_xrl)
#define XRL_Offset_of_NSY(nsy) ((nsy)→t_xrl_offset)
```

⟨ Widely aligned NSY elements 236 ⟩ +≡
```
XRL t_lhs_xrl;
int t_xrl_offset;
```

246. ⟨ Initialize NSY elements 218 ⟩ +≡
```
LHS_XRL_of_NSY(nsy) ⟸ Λ;
XRL_Offset_of_NSY(nsy) ⟸ −1;
```

247. Virtual LHS trace accessor: If this symbol is an internal LHS used in the rewrite of an external rule, returns the XRLID. If there is no such external rule, returns −1. On other failures, returns −2.

248. ⟨ Function definitions 41 ⟩ +≡
```
Marpa_Rule_ID _marpa_g_nsy_lhs_xrl(Marpa_Grammar g, Marpa_NSY_ID nsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if nsy_id is invalid 1235 ⟩
  {
    const NSY nsy ⟸ NSY_by_ID(nsy_id);
    const XRL lhs_xrl ⟸ LHS_XRL_of_NSY(nsy);
    if (lhs_xrl) return ID_of_XRL(lhs_xrl);
  }
  return −1;
}
```

249. If the NSY was created as a LHS during the rewrite of an external rule, and there is an associated offset within that rule, this call returns the offset. This value is especially relevant for the symbols used in the CHAF rewrite. Otherwise, -1 is returned. On other failures, returns −2.

⟨ Function definitions 41 ⟩ +≡
```
int _marpa_g_nsy_xrl_offset(Marpa_Grammar g, Marpa_NSY_ID nsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  NSY nsy;
  ⟨ Fail if nsy_id is invalid 1235 ⟩
  nsy ⟸ NSY_by_ID(nsy_id);
  return XRL_Offset_of_NSY(nsy);
}
```

250. Rank. The rank of the internal symbol.
```
#define NSY_Rank_by_XSY(xsy)
        ((xsy)→t_rank ∗ EXTERNAL_RANK_FACTOR + MAXIMUM_CHAF_RANK)
#define Rank_of_NSY(nsy) ((nsy)→t_rank)
```

⟨ Int aligned NSY elements 250 ⟩ ≡
```
Marpa_Rank t_rank;
```
This code is used in section 217.

251. ⟨ Initialize NSY elements 218 ⟩ +≡

```
Rank_of_NSY(nsy) ⟸ Default_Rank_of_G(g) * EXTERNAL_RANK_FACTOR +
    MAXIMUM_CHAF_RANK;
```

252. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Rank _marpa_g_nsy_rank(Marpa_Grammar g, Marpa_NSY_ID nsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if nsy_id is invalid 1235 ⟩
  return Rank_of_NSY(NSY_by_ID(nsy_id));
}
```

253. External rule (XRL) code.

```
⟨ Public typedefs 91 ⟩ +≡
  typedef int Marpa_Rule_ID;
```

254.

```
⟨ Private structures 48 ⟩ +≡
  struct s_xrl {
    ⟨ Int aligned rule elements 267 ⟩
    ⟨ Bit aligned rule elements 280 ⟩
    ⟨ Final rule elements 268 ⟩
  };
```

255.

```
⟨ Private typedefs 49 ⟩ +≡
  struct s_xrl;
  typedef struct s_xrl *XRL;
  typedef XRL RULE;
  typedef Marpa_Rule_ID RULEID;
  typedef Marpa_Rule_ID XRLID;
```

256. Rule construction.

257. Set up the basic data. This logic is intended to be common to all individual rules. The name comes from the idea that this logic "starts" the initialization of a rule. It is assummed that the caller has checked that all symbol ID's are valid.

258. Not inline because GCC complains, and not unreasonably. It is big, and it is used in a lot of places.

```
⟨ Function definitions 41 ⟩ +≡
  PRIVATE XRL xrl_start(GRAMMAR g, const XSYID lhs, const XSYID *rhs, int
          length)
  {
    XRL xrl;
    const size_t sizeof_xrl ⟸ offsetof(struct s_xrl, t_symbols) + ((size_t)
        length + 1) * sizeof (xrl→t_symbols[0]);
    xrl ⟸ marpa_obs_start(g→t_xrl_obs, sizeof_xrl, ALIGNOF(XRL));
    Length_of_XRL(xrl) ⟸ length;
    xrl→t_symbols[0] ⟸ lhs;
    XSY_is_LHS(XSY_by_ID(lhs)) ⟸ 1;
    {
      int i;
      for (i ⟸ 0; i < length; i++) {
        xrl→t_symbols[i + 1] ⟸ rhs[i];
      }
    }
    return xrl;
```

```
}
PRIVATE XRL xrl_finish(GRAMMAR g, XRL rule)
{
  ⟨ Initialize rule elements 277 ⟩
  rule_add(g, rule);
  return rule;
}
PRIVATE_NOT_INLINE RULE rule_new(GRAMMAR g, const XSYID lhs, const
        XSYID *rhs, int length)
{
  RULE rule ⟸ xrl_start(g, lhs, rhs, length);
  xrl_finish(g, rule);
  rule ⟸ marpa_obs_finish(g→t_xrl_obs);
  return rule;
}
```

259. This is the logic common to every IRL construction.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE IRLirl_start(GRAMMAR g, int length)
{
  IRLirl;
  const size_t sizeof_irl ⟸ offsetof(struct s_irl, t_nsyid_array) + ((size_t)
      length + 1) * sizeof (irl→t_nsyid_array[0]);
    /* Needs to be aligned as an IRL */
  irl ⟸ marpa__obs_alloc(g→t_obs, sizeof_irl, ALIGNOF(IRL_Object));
  ID_of_IRL(irl) ⟸ MARPA_DSTACK_LENGTH((g)→t_irl_stack);
  Length_of_IRL(irl) ⟸ length;
  ⟨ Initialize IRL elements 342 ⟩
  *MARPA_DSTACK_PUSH((g)→t_irl_stack, IRL) ⟸ irl;
  return irl;
}
PRIVATE void irl_finish(GRAMMAR g, IRLirl)
{
  const NSY lhs_nsy ⟸ LHS_of_IRL(irl);
  NSY_is_LHS(lhs_nsy) ⟸ 1;
}
```

260. ⟨ Clone a new IRL from rule 260 ⟩ ≡

```
{
  int symbol_ix;
  const IRLnew_irl ⟸ irl_start(g, rewrite_xrl_length);
  Source_XRL_of_IRL(new_irl) ⟸ rule;
  Rank_of_IRL(new_irl) ⟸ IRL_Rank_by_XRL(rule);
```

```
    for (symbol_ix ⟸ 0; symbol_ix ≤ rewrite_xrl_length; symbol_ix++) {
      new_irl→t_nsyid_array[symbol_ix] ⟸
          NSYID_by_XSYID(rule→t_symbols[symbol_ix]);
    }
    irl_finish(g, new_irl);
  }
```

This code is used in section 413.

261. ⟨ Function definitions 41 ⟩ +≡

```
  Marpa_Rule_ID marpa_g_rule_new(Marpa_Grammar g, Marpa_Symbol_ID
          lhs_id, Marpa_Symbol_ID *rhs_ids, int length)
  {
    ⟨ Return −2 on failure 1229 ⟩
    Marpa_Rule_ID rule_id;
    RULE rule;
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if precomputed 1230 ⟩
    if (_MARPA_UNLIKELY(length > MAX_RHS_LENGTH)) {
      MARPA_ERROR(MARPA_ERR_RHS_TOO_LONG);
      return failure_indicator;
    }
    if (_MARPA_UNLIKELY(¬xsy_id_is_valid(g, lhs_id))) {
      MARPA_ERROR(MARPA_ERR_INVALID_SYMBOL_ID);
      return failure_indicator;
    }
    {
      int rh_index;
      for (rh_index ⟸ 0; rh_index < length; rh_index++) {
        const XSYID rhs_id ⟸ rhs_ids[rh_index];
        if (_MARPA_UNLIKELY(¬xsy_id_is_valid(g, rhs_id))) {
          MARPA_ERROR(MARPA_ERR_INVALID_SYMBOL_ID);
          return failure_indicator;
        }
      }
    }
    {
      const XSY lhs ⟸ XSY_by_ID(lhs_id);
      if (_MARPA_UNLIKELY(XSY_is_Sequence_LHS(lhs))) {
        MARPA_ERROR(MARPA_ERR_SEQUENCE_LHS_NOT_UNIQUE);
        return failure_indicator;
      }
    }
    rule ⟸ xrl_start(g, lhs_id, rhs_ids, length);
    if (_MARPA_UNLIKELY(_marpa_avl_insert(g→t_xrl_tree, rule) ≠ Λ)) {
```

```
    MARPA_ERROR(MARPA_ERR_DUPLICATE_RULE);
    marpa_obs_reject(g→t_xrl_obs);
    return failure_indicator;
  }
  rule ⟸ xrl_finish(g, rule);
  rule ⟸ marpa_obs_finish(g→t_xrl_obs);
  XRL_is_BNF(rule) ⟸ 1;
  rule_id ⟸ rule→t_id;
  return rule_id;
}
```

262. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Rule_ID marpa_g_sequence_new(Marpa_Grammar g, Marpa_Symbol_ID
        lhs_id, Marpa_Symbol_ID rhs_id, Marpa_Symbol_ID separator_id, int
        min, int flags)
{
  RULE original_rule;
  RULEID original_rule_id ⟸ −2;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if precomputed 1230 ⟩
  ⟨ Check that the sequence symbols are valid 264 ⟩
  ⟨ Add the original rule for a sequence 263 ⟩
  return original_rule_id;
FAILURE: return failure_indicator;
}
```

263. As a side effect, this checks the LHS and RHS symbols for validity.

⟨ Add the original rule for a sequence 263 ⟩ ≡

```
{
  original_rule ⟸ rule_new(g, lhs_id, &rhs_id, 1);
  original_rule_id ⟸ original_rule→t_id;
  if (separator_id ≥ 0) Separator_of_XRL(original_rule) ⟸ separator_id;
  Minimum_of_XRL(original_rule) ⟸ min;
  XRL_is_Sequence(original_rule) ⟸ 1;
  original_rule→t_is_discard ⟸ ¬(flags & MARPA_KEEP_SEPARATION) ∧
      separator_id ≥ 0;
  if (flags & MARPA_PROPER_SEPARATION) {
    XRL_is_Proper_Separation(original_rule) ⟸ 1;
  }
  XSY_is_Sequence_LHS(XSY_by_ID(lhs_id)) ⟸ 1;
  XSY_by_ID(rhs_id)→t_is_counted ⟸ 1;
  if (separator_id ≥ 0) {
    XSY_by_ID(separator_id)→t_is_counted ⟸ 1;
  }
```

```
}
```

This code is used in section 262.

264. ⟨ Check that the sequence symbols are valid 264 ⟩ ≡

```
{
  if (separator_id ≠ −1) {
    if (_MARPA_UNLIKELY(¬xsy_id_is_valid(g, separator_id))) {
      MARPA_ERROR(MARPA_ERR_BAD_SEPARATOR);
      goto FAILURE;
    }
  }
  if (_MARPA_UNLIKELY(¬xsy_id_is_valid(g, lhs_id))) {
    MARPA_ERROR(MARPA_ERR_INVALID_SYMBOL_ID);
    goto FAILURE;
  }
  {
    const XSY lhs ⇐ XSY_by_ID(lhs_id);
    if (_MARPA_UNLIKELY(XSY_is_LHS(lhs))) {
      MARPA_ERROR(MARPA_ERR_SEQUENCE_LHS_NOT_UNIQUE);
      goto FAILURE;
    }
  }
  if (_MARPA_UNLIKELY(¬xsy_id_is_valid(g, rhs_id))) {
    MARPA_ERROR(MARPA_ERR_INVALID_SYMBOL_ID);
    goto FAILURE;
  }
}
```

This code is used in section 262.

265. Does this rule duplicate an already existing rule? A duplicate is a rule with the same lhs symbol, the same rhs length, and the same symbol in each position on the rhs. BNF rules are prevented from duplicating sequence rules because sequence LHS's are required to be unique.

The order of the sort function is for convenience in computation. All that matters is that identical rules sort the same and otherwise the order does not need to make sense.

I do not think the restrictions on sequence rules represent real limitations. Multiple sequences with the same lhs and rhs would be very confusing. And users who really, really want such them are free to write the sequences out as BNF rules. After all, sequence rules are only a shorthand. And shorthand is counter-productive when it makes you lose track of what you are trying to say.

266. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE_NOT_INLINE int duplicate_rule_cmp(const void *ap, const void
        *bp, void *param UNUSED)
{
  XRL xrl1 ⟸ (XRL) ap;
  XRL xrl2 ⟸ (XRL) bp;
  int diff ⟸ LHS_ID_of_XRL(xrl2) − LHS_ID_of_XRL(xrl1);
  if (diff) return diff;
  {
    /* Length is a key in-between LHS. That way we only need to compare the RHS
        of rules of the same length */
    int ix;
    const int length ⟸ Length_of_XRL(xrl1);
    diff ⟸ Length_of_XRL(xrl2) − length;
    if (diff) return diff;
    for (ix ⟸ 0; ix < length; ix++) {
      diff ⟸ RHS_ID_of_XRL(xrl2, ix) − RHS_ID_of_XRL(xrl1, ix);
      if (diff) return diff;
    }
  }
  return 0;
}
```

267. Rule symbols. A rule takes the traditiona form of a left hand side (LHS), and a right hand side (RHS). The **length** of a rule is the length of the RHS — there is always exactly one LHS symbol. Maximum length of the RHS is restricted. I take off two more bits than necessary, as a fudge factor. This is only checked for new rules. The rules generated internally by libmarpa are either shorter than a small constant in length, or else shorter than the XRL which is their source. On a 32-bit machine, this still allows a RHS of over a billion symbols. I believe by the time 64-bit machines become universal, nobody will have noticed this restriction.

```
#define MAX_RHS_LENGTH (INT_MAX ≫ (2))
#define Length_of_XRL(xrl) ((xrl)→t_rhs_length)
```

⟨ Int aligned rule elements 267 ⟩ ≡

```
int t_rhs_length;
```

See also sections 275 and 276.

This code is used in section 254.

268. The symbols come at the end of the marpa_rule structure, so that they can be variable length.

⟨ Final rule elements 268 ⟩ ≡

```
Marpa_Symbol_ID t_symbols[1];
```

This code is used in section 254.

269. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE Marpa_Symbol_ID rule_lhs_get(RULE rule)
{
  return rule→t_symbols[0];
}
```

270. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Symbol_ID marpa_g_rule_lhs(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  return rule_lhs_get(XRL_by_ID(xrl_id));
}
```

271. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE Marpa_Symbol_ID *rule_rhs_get(RULE rule)
{
  return rule→t_symbols + 1;
}
```

272. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Symbol_ID marpa_g_rule_rhs(Marpa_Grammar g, Marpa_Rule_ID xrl_id, int
        ix)
{
  RULE rule;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  rule ⟸ XRL_by_ID(xrl_id);
  if (ix < 0) {
    MARPA_ERROR(MARPA_ERR_RHS_IX_NEGATIVE);
    return failure_indicator;
  }
  if (Length_of_XRL(rule) ≤ ix) {
    MARPA_ERROR(MARPA_ERR_RHS_IX_OOB);
    return failure_indicator;
  }
  return RHS_ID_of_RULE(rule, ix);
}
```

273. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_rule_length(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  return Length_of_XRL(XRL_by_ID(xrl_id));
}
```

274. Symbols of the rule.

```
#define LHS_ID_of_RULE(rule) ((rule)→t_symbols[0])
#define LHS_ID_of_XRL(xrl) ((xrl)→t_symbols[0])
#define RHS_ID_of_RULE(rule, position) ((rule)→t_symbols[(position) + 1])
#define RHS_ID_of_XRL(xrl, position) ((xrl)→t_symbols[(position) + 1])
```

275. Rule ID. The **rule ID** is a number which acts as the unique identifier for a rule. The rule ID is initialized when the rule is added to the list of rules.

```
#define ID_of_XRL(xrl) ((xrl)→t_id)
#define ID_of_RULE(rule) ID_of_XRL(rule)
```

⟨ Int aligned rule elements 267 ⟩ +≡

```
  Marpa_Rule_ID t_id;
```

276. Rule rank.

⟨ Int aligned rule elements 267 ⟩ +≡

```
  Marpa_Rank t_rank;
```

277. ⟨ Initialize rule elements 277 ⟩ ≡

```
  rule→t_rank ⇐ Default_Rank_of_G(g);
```

See also sections 281, 285, 287, 289, 292, 297, 301, 305, 308, 311, 315, 318, and 321.

This code is used in section 258.

278.

```
#define Rank_of_XRL(rule) ((rule)→t_rank)
```

⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_rule_rank(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  XRL xrl;
  ⟨ Return −2 on failure 1229 ⟩
  clear_error(g);
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Fail if xrl_id does not exist 1240 ⟩
  clear_error(g);
  xrl ⇐ XRL_by_ID(xrl_id);
  return Rank_of_XRL(xrl);
}
```

279. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_rule_rank_set(Marpa_Grammar g, Marpa_Rule_ID xrl_id, Marpa_Rank
        rank)
{
  XRL xrl;
  ⟨ Return −2 on failure 1229 ⟩
  clear_error(g);
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if precomputed 1230 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Fail if xrl_id does not exist 1240 ⟩
  xrl ⟸ XRL_by_ID(xrl_id);
  if (_MARPA_UNLIKELY(rank < MINIMUM_RANK)) {
    MARPA_ERROR(MARPA_ERR_RANK_TOO_LOW);
    return failure_indicator;
  }
  if (_MARPA_UNLIKELY(rank > MAXIMUM_RANK)) {
    MARPA_ERROR(MARPA_ERR_RANK_TOO_HIGH);
    return failure_indicator;
  }
  return Rank_of_XRL(xrl) ⟸ rank;
}
```

280. Rule ranks high?. The "rule ranks high" setting affects the ranking of the null variants, for rules with properly nullable symbols on their RHS.

⟨ Bit aligned rule elements 280 ⟩ ≡

```
  BITFIELD t_null_ranks_high:1;
```

See also sections 284, 286, 288, 291, 296, 300, 304, 307, 310, 314, 317, and 320.

This code is used in section 254.

281. ⟨ Initialize rule elements 277 ⟩ +≡

```
  rule→t_null_ranks_high ⟸ 0;
```

282.

```
#define Null_Ranks_High_of_RULE(rule) ((rule)→t_null_ranks_high)
```

⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_rule_null_high(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  XRL xrl;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  xrl ⟸ XRL_by_ID(xrl_id);
```

```
    return Null_Ranks_High_of_RULE(xrl);
  }
```

283. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_g_rule_null_high_set(Marpa_Grammar g, Marpa_Rule_ID xrl_id, int
          flag)
  {
    XRL xrl;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if precomputed 1230 ⟩
    ⟨ Fail if xrl_id is malformed 1241 ⟩
    ⟨ Soft fail if xrl_id does not exist 1239 ⟩
    xrl ⟸ XRL_by_ID(xrl_id);
    if (_MARPA_UNLIKELY(flag < 0 ∨ flag > 1)) {
      MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
      return failure_indicator;
    }
    return Null_Ranks_High_of_RULE(xrl) ⟸ Boolean(flag);
  }
```

284. Rule is user-created BNF?. True for if the rule is a user-created BNF rule, false otherwise.

```
#define XRL_is_BNF(rule) ((rule)→t_is_bnf)
⟨ Bit aligned rule elements 280 ⟩ +≡
  BITFIELD t_is_bnf:1;
```

285. ⟨ Initialize rule elements 277 ⟩ +≡

```
  rule→t_is_bnf ⟸ 0;
```

286. Rule is sequence?.

```
#define XRL_is_Sequence(rule) ((rule)→t_is_sequence)
⟨ Bit aligned rule elements 280 ⟩ +≡
  BITFIELD t_is_sequence:1;
```

287. ⟨ Initialize rule elements 277 ⟩ +≡

```
  rule→t_is_sequence ⟸ 0;
```

288. Sequence minimum length. The minimum length for a sequence rule. This accessor can also be used as a test of whether or not a rule is a sequence rule. −1 is returned if and only if the rule is valid but not a sequence rule. Rule IDs which do not exist and other failures are hard failures.

```
#define Minimum_of_XRL(rule) ((rule)→t_minimum)
⟨ Bit aligned rule elements 280 ⟩ +≡
  int t_minimum;
```

289. ⟨ Initialize rule elements 277 ⟩ +≡

```
rule→t_minimum ⇐ −1;
```

290. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_sequence_min(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  XRL xrl;
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Fail if xrl_id does not exist 1240 ⟩
  xrl ⇐ XRL_by_ID(xrl_id);
  if (¬XRL_is_Sequence(xrl)) {
    MARPA_ERROR(MARPA_ERR_NOT_A_SEQUENCE);
    return −1;
  }
  return Minimum_of_XRL(xrl);
}
```

291. Sequence separator. Rule IDs which do not exist and other failures are hard failures.

```
#define Separator_of_XRL(rule) ((rule)→t_separator_id)
```

⟨ Bit aligned rule elements 280 ⟩ +≡

```
XSYID t_separator_id;
```

292. ⟨ Initialize rule elements 277 ⟩ +≡

```
Separator_of_XRL(rule) ⇐ −1;
```

293. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Symbol_ID marpa_g_sequence_separator(Marpa_Grammar g, Marpa_Rule_ID
        xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  XRL xrl;
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Fail if xrl_id does not exist 1240 ⟩
  xrl ⇐ XRL_by_ID(xrl_id);
  if (¬XRL_is_Sequence(xrl)) {
    MARPA_ERROR(MARPA_ERR_NOT_A_SEQUENCE);
    return failure_indicator;
  }
  return Separator_of_XRL(xrl);
}
```

294. Rule keeps separator?. When this rule is evaluated by the semantics, do they want to see the separators? Default is that they are thrown away. Usually the role of the separators is only syntactic, and that is what is wanted. For non-sequence rules, this flag should be false.

295. To Do: At present this call does nothing except return the value of an undocumented and unused flag. In the future, this flag may be used to optimize the evaluation in cases where separators are discarded. Alternatively, it may be deleted.

⟨ Public defines 109 ⟩ +≡
#define `MARPA_KEEP_SEPARATION` #1

296. ⟨ Bit aligned rule elements 280 ⟩ +≡
BITFIELD `t_is_discard`:1;

297. ⟨ Initialize rule elements 277 ⟩ +≡
`rule→t_is_discard` ⟸ 0;

298. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_rule_is_keep_separation(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  return ¬XRL_by_ID(xrl_id)→t_is_discard;
}
```

299. Rule has proper separation?. In Marpa's terminology, proper separation means that a sequence cannot legally end with a separator. In "proper" separation, the term separator is interpreted strictly, as something which separates two list items. A separator coming after the final list item does not separate two items, and therefore traditionally was considered a syntax error.

Proper separation is often inconvenient, or even counter-productive. Increasingly, the practice is to be "liberal" and to allow a separator to come after the last list item. Liberal separation is the default in Marpa.

There is not bitfield for this, because proper separation is a completely syntactic matter, taken care of in the rewrite itself.

#define `XRL_is_Proper_Separation(rule)` `((rule)→t_is_proper_separation)`

⟨ Public defines 109 ⟩ +≡
#define `MARPA_PROPER_SEPARATION` #2

300. ⟨ Bit aligned rule elements 280 ⟩ +≡
BITFIELD `t_is_proper_separation`:1;

301. ⟨ Initialize rule elements 277 ⟩ +≡

```
rule→t_is_proper_separation ⇐ 0;
```

302. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_rule_is_proper_separation(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  return XRL_is_Proper_Separation(XRL_by_ID(xrl_id));
}
```

303. Loop rule.

304. A rule is a loop rule if it non-trivially produces the string of length one which consists only of its LHS symbol. "Non-trivially" means the zero-step derivation does not count – the derivation must have at least one step.

⟨ Bit aligned rule elements 280 ⟩ +≡

```
BITFIELD t_is_loop:1;
```

305. ⟨ Initialize rule elements 277 ⟩ +≡

```
rule→t_is_loop ⇐ 0;
```

306. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_rule_is_loop(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  return XRL_by_ID(xrl_id)→t_is_loop;
}
```

307. Is rule nulling?. Is the rule nulling?

```
#define XRL_is_Nulling(rule) ((rule)→t_is_nulling)
```

⟨ Bit aligned rule elements 280 ⟩ +≡

```
BITFIELD t_is_nulling:1;
```

308. ⟨ Initialize rule elements 277 ⟩ +≡

```
XRL_is_Nulling(rule) ⇐ 0;
```

309. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_rule_is_nulling(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  XRL xrl;
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  xrl ⟸ XRL_by_ID(xrl_id);
  return XRL_is_Nulling(xrl);
}
```

310. Is rule nullable?. Is the rule nullable?

```
#define XRL_is_Nullable(rule) ((rule)→t_is_nullable)
```

⟨ Bit aligned rule elements 280 ⟩ +≡

```
BITFIELD t_is_nullable:1;
```

311. ⟨ Initialize rule elements 277 ⟩ +≡

```
XRL_is_Nullable(rule) ⟸ 0;
```

312. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_rule_is_nullable(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  XRL xrl;
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  xrl ⟸ XRL_by_ID(xrl_id);
  return XRL_is_Nullable(xrl);
}
```

313. Is rule accessible?.

314. A rule is accessible if its LHS is accessible.

```
#define XRL_is_Accessible(rule) ((rule)→t_is_accessible)
```

⟨ Bit aligned rule elements 280 ⟩ +≡

```
BITFIELD t_is_accessible:1;
```

315. ⟨ Initialize rule elements 277 ⟩ +≡

```
XRL_is_Accessible(rule) ⟸ 1;
```

316. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_rule_is_accessible(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  XRL xrl;
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  xrl ⟸ XRL_by_ID(xrl_id);
  return XRL_is_Accessible(xrl);
}
```

317. Is rule productive?. Is the rule productive?

```
#define XRL_is_Productive(rule) ((rule)→t_is_productive)
```

⟨ Bit aligned rule elements 280 ⟩ +≡

```
  BITFIELD t_is_productive:1;
```

318. ⟨ Initialize rule elements 277 ⟩ +≡

```
  XRL_is_Productive(rule) ⟸ 1;
```

319. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_g_rule_is_productive(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  XRL xrl;
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  xrl ⟸ XRL_by_ID(xrl_id);
  return XRL_is_Productive(xrl);
}
```

320. Is XRL used?.

```
#define XRL_is_Used(rule) ((rule)→t_is_used)
```

⟨ Bit aligned rule elements 280 ⟩ +≡

```
  BITFIELD t_is_used:1;
```

321. Initialize to not used, because that's easier to debug.

⟨ Initialize rule elements 277 ⟩ +≡

```
  XRL_is_Used(rule) ⟸ 0;
```

322. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_rule_is_used(Marpa_Grammar g, Marpa_Rule_ID xrl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if xrl_id is malformed 1241 ⟩
  ⟨ Soft fail if xrl_id does not exist 1239 ⟩
  return XRL_is_Used(XRL_by_ID(xrl_id));
}
```

323. If this rule is the semantic equivalent of another rule, this external accessor returns the "original rule". Otherwise it returns -1.

324. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Rule_ID _marpa_g_irl_semantic_equivalent(Marpa_Grammar
      g, Marpa_IRL_ID irl_id)
{
  IRLirl;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if irl_id is invalid 1238 ⟩
  irl ⟸ IRL_by_ID(irl_id);
  if (IRL_has_Virtual_LHS(irl)) return −1;
  return ID_of_XRL(Source_XRL_of_IRL(irl));
}
```

325. Internal rule (IRL) code.

326. ⟨ Private structures 48 ⟩ +≡

```
struct s_irl {
  ⟨ Widely aligned IRL elements 359 ⟩
  ⟨ Int aligned IRL elements 329 ⟩
  ⟨ Bit aligned IRL elements 341 ⟩
  ⟨ Final IRL elements 331 ⟩
};
typedef struct s_irl IRL_Object;
```

327. ⟨ Public typedefs 91 ⟩ +≡

```
typedef int Marpa_IRL_ID;
```

328. ⟨ Private typedefs 49 ⟩ +≡

```
struct s_irl;
typedef struct s_irl *IRL;
typedef Marpa_IRL_ID IRLID;
```

329. ID. The **IRL ID** is a number which acts as the unique identifier for an IRL. The rule ID is initialized when the IRL is added to the list of rules.

```
#define ID_of_IRL(irl) ((irl)→t_irl_id)
```

⟨ Int aligned IRL elements 329 ⟩ ≡

```
IRLID t_irl_id;
```

See also sections 336, 338, 350, 353, 356, 362, and 472.

This code is used in section 326.

330. Symbols.

331. The symbols come at the end of the structure, so that they can be variable length.

⟨ Final IRL elements 331 ⟩ ≡

```
NSYID t_nsyid_array[1];
```

This code is used in section 326.

332.

```
#define LHSID_of_IRL(irlid) ((irlid)→t_nsyid_array[0])
```

333.

```
#define LHS_of_IRL(irl) (NSY_by_ID(LHSID_of_IRL(irl)))
```

⟨ Function definitions 41 ⟩ +≡

```
Marpa_NSY_ID _marpa_g_irl_lhs(Marpa_Grammar g, Marpa_IRL_ID irl_id)
{
  IRL irl;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if irl_id is invalid 1238 ⟩
```

```
    irl ⟸ IRL_by_ID(irl_id);
    return LHSID_of_IRL(irl);
  }
```

334. *#define* `RHSID_of_IRL(irl, position)` `((irl)→t_nsyid_array[(position) + 1])`

335. *#define* `RHS_of_IRL(irl, position)`
`NSY_by_ID(RHSID_of_IRL((irl), (position)))`

⟨ Function definitions 41 ⟩ +≡

```
  Marpa_NSY_ID _marpa_g_irl_rhs(Marpa_Grammar g, Marpa_IRL_ID irl_id, int ix)
  {
    IRL irl;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if not precomputed 1231 ⟩
    ⟨ Fail if irl_id is invalid 1238 ⟩
    irl ⟸ IRL_by_ID(irl_id);
    if (Length_of_IRL(irl) ≤ ix) return −1;
    return RHSID_of_IRL(irl, ix);
  }
```

336. *#define* `Length_of_IRL(irl)` `((irl)→t_length)`

⟨ Int aligned IRL elements 329 ⟩ +≡

```
  int t_length;
```

337. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_g_irl_length(Marpa_Grammar g, Marpa_IRL_ID irl_id)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if not precomputed 1231 ⟩
    ⟨ Fail if irl_id is invalid 1238 ⟩
    return Length_of_IRL(IRL_by_ID(irl_id));
  }
```

338. An IRL is a unit rule (that is, a rule of length one, not counting nullable symbols) if and only if its AHM count is 2 – the predicted AHM and the final AHM.

#define `IRL_is_Unit_Rule(irl)` `((irl)→t_ahm_count ≡ 2)`
#define `AHM_Count_of_IRL(irl)` `((irl)→t_ahm_count)`

⟨ Int aligned IRL elements 329 ⟩ +≡

```
  int t_ahm_count;
```

339. IRL has virtual LHS?. This is for Marpa's "internal semantics". When Marpa rewrites rules, it does so in a way invisible to the user's semantics. It does this by marking rules so that it can reassemble the results of rewritten rules to appear "as if" they were the result of evaluating the original, un-rewritten rule.

All Marpa's rewrites allow the rewritten rules to be "dummied up" to look like the originals. That this must be possible for any rewrite was one of Marpa's design criteria. It was an especially non-negotiable criteria, because almost the only reason for parsing a grammar is to apply the semantics specified for the original grammar.

340. The rewriting of rules into internal rules must be such that every one of their parses corresponds to a "factoring" – a way of dividing up the input. If the rewriting is unambiguous, this is trivially true. For an ambiguous rewrite, each parse will be visible external as a unique "factoring" of the external rule's RHS symbols by location, and the rewrite must make sense when interpreted that way.

341. An IRL has an external semantics if and only if it does have a non-virtual LHS. And if a rule does not have a virtual LHS, then its LHS side NSY must have a semantic XRL.

```
#define IRL_has_Virtual_LHS(irl) ((irl)→t_is_virtual_lhs)
```

⟨ Bit aligned IRL elements 341 ⟩ ≡

```
  BITFIELD t_is_virtual_lhs:1;
```

See also sections 344, 347, and 409.

This code is used in section 326.

342. ⟨ Initialize IRL elements 342 ⟩ ≡

```
  IRL_has_Virtual_LHS(irl) ⇐ 0;
```

See also sections 345, 348, 351, 354, 357, 360, 363, 366, 410, and 473.

This code is used in section 259.

343. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_g_irl_is_virtual_lhs(Marpa_Grammar g, Marpa_IRL_ID irl_id)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if not precomputed 1231 ⟩
    ⟨ Fail if irl_id is invalid 1238 ⟩
    return IRL_has_Virtual_LHS(IRL_by_ID(irl_id));
  }
```

344. IRL has virtual RHS?.

```
#define IRL_has_Virtual_RHS(irl) ((irl)→t_is_virtual_rhs)
```

⟨ Bit aligned IRL elements 341 ⟩ +≡

```
  BITFIELD t_is_virtual_rhs:1;
```

345. ⟨ Initialize IRL elements 342 ⟩ +≡

```
  IRL_has_Virtual_RHS(irl) ⇐ 0;
```

346. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_irl_is_virtual_rhs(Marpa_Grammar g, Marpa_IRL_ID irl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if irl_id is invalid 1238 ⟩
  return IRL_has_Virtual_RHS(IRL_by_ID(irl_id));
}
```

347. IRL right recursion status. Being right recursive, for an IRL, means it will be used in the Leo logic.

```
#define IRL_is_Right_Recursive(irl) ((irl)→t_is_right_recursive)
#define IRL_is_Leo(irl) IRL_is_Right_Recursive(irl)
```

⟨ Bit aligned IRL elements 341 ⟩ +≡

```
BITFIELD t_is_right_recursive:1;
```

348. ⟨ Initialize IRL elements 342 ⟩ +≡

```
IRL_is_Right_Recursive(irl) ⟸ 0;
```

349. Rule real symbol count. This is another data element used for the "internal semantics" – the logic to reassemble results of rewritten rules so that they look as if they came from the original, un-rewritten rules. The value of this field is meaningful if and only if the rule has a virtual rhs or a virtual lhs.

```
#define Real_SYM_Count_of_IRL(irl) ((irl)→t_real_symbol_count)
```

350. ⟨ Int aligned IRL elements 329 ⟩ +≡

```
int t_real_symbol_count;
```

351. ⟨ Initialize IRL elements 342 ⟩ +≡

```
Real_SYM_Count_of_IRL(irl) ⟸ 0;
```

352. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_real_symbol_count(Marpa_Grammar g, Marpa_IRL_ID irl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if irl_id is invalid 1238 ⟩
  return Real_SYM_Count_of_IRL(IRL_by_ID(irl_id));
}
```

353. Virtual start position. For an IRL, this is the RHS position in the XRL where the IRL starts.

```
#define Virtual_Start_of_IRL(irl) ((irl)→t_virtual_start)
```

⟨ Int aligned IRL elements 329 ⟩ +≡

```
int t_virtual_start;
```

354. ⟨ Initialize IRL elements 342 ⟩ +≡
```
irl→t_virtual_start ⇐ −1;
```

355. ⟨ Function definitions 41 ⟩ +≡
```
int _marpa_g_virtual_start(Marpa_Grammar g, Marpa_IRL_ID irl_id)
{
  IRL irl;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if irl_id is invalid 1238 ⟩
  irl ⇐ IRL_by_ID(irl_id);
  return Virtual_Start_of_IRL(irl);
}
```

356. Virtual end position. For an IRL, this is the RHS position in the XRL where the IRL ends.

```
#define Virtual_End_of_IRL(irl) ((irl)→t_virtual_end)
```

⟨ Int aligned IRL elements 329 ⟩ +≡
```
int t_virtual_end;
```

357. ⟨ Initialize IRL elements 342 ⟩ +≡
```
irl→t_virtual_end ⇐ −1;
```

358. ⟨ Function definitions 41 ⟩ +≡
```
int _marpa_g_virtual_end(Marpa_Grammar g, Marpa_IRL_ID irl_id)
{
  IRL irl;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if irl_id is invalid 1238 ⟩
  irl ⇐ IRL_by_ID(irl_id);
  return Virtual_End_of_IRL(irl);
}
```

359. Source XRL. This is the "source" of the IRL – the XRL that it is derived from. Currently, there is no dedicated flag for determining whether this rule also provides the semantics, because the "virtual LHS" flag serves that purpose.

```
#define Source_XRL_of_IRL(irl) ((irl)→t_source_xrl)
```

⟨ Widely aligned IRL elements 359 ⟩ ≡
```
XRL t_source_xrl;
```

See also section 365.

This code is used in section 326.

360. ⟨ Initialize IRL elements 342 ⟩ +≡
```
Source_XRL_of_IRL(irl) ⇐ Λ;
```

361. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Rule_ID _marpa_g_source_xrl(Marpa_Grammar g, Marpa_IRL_ID irl_id)
{
  XRL source_xrl;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if irl_id is invalid 1238 ⟩
  source_xrl ⟸ Source_XRL_of_IRL(IRL_by_ID(irl_id));
  return source_xrl ? ID_of_XRL(source_xrl) : −1;
}
```

362. Rank. The rank of the internal rule. IRL_Rank_by_XRL and IRL_CHAF_Rank_by_XRL assume that t_source_xrl is not Λ.

```
#define EXTERNAL_RANK_FACTOR 4
#define MAXIMUM_CHAF_RANK 3
#define IRL_CHAF_Rank_by_XRL(xrl, chaf_rank)
        (((xrl)→t_rank ∗ EXTERNAL_RANK_FACTOR) + (((xrl)→t_null_ranks_high) ?
            (MAXIMUM_CHAF_RANK − (chaf_rank)) : (chaf_rank)))
#define IRL_Rank_by_XRL(xrl) IRL_CHAF_Rank_by_XRL((xrl), MAXIMUM_CHAF_RANK)
#define Rank_of_IRL(irl) ((irl)→t_rank)
```

⟨ Int aligned IRL elements 329 ⟩ +≡

```
  Marpa_Rank t_rank;
```

363. ⟨ Initialize IRL elements 342 ⟩ +≡

```
  Rank_of_IRL(irl) ⟸ Default_Rank_of_G(g) ∗ EXTERNAL_RANK_FACTOR +
      MAXIMUM_CHAF_RANK;
```

364. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Rank _marpa_g_irl_rank(Marpa_Grammar g, Marpa_IRL_ID irl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if irl_id is invalid 1238 ⟩
  return Rank_of_IRL(IRL_by_ID(irl_id));
}
```

365. First AHM. This is the first AHM for a rule. There may not be one, in which case it is Λ. Currently, this is not used after grammar precomputation, and there may be an optimization here. Perhaps later Marpa objects **should** be using it.

```
#define First_AHM_of_IRL(irl) ((irl)→t_first_ahm)
#define First_AHM_of_IRLID(irlid) (IRL_by_ID(irlid)→t_first_ahm)
```

⟨ Widely aligned IRL elements 359 ⟩ +≡

```
  AHMt_first_ahm;
```

366. ⟨ Initialize IRL elements 342 ⟩ +≡

```
  First_AHM_of_IRL(irl) ⟸ Λ;
```

367. Precomputing the grammar. Marpa's logic divides roughly into three pieces – grammar precomputation, the actual parsing of input tokens, and semantic evaluation. Precomputing the grammar is complex enough to divide into several stages of its own, which are covered in the next few sections. This section describes the top-level method for precomputation, which is external.

368. If `marpa_g_precompute` is called on a precomputed grammar, the upper layers have a lot of latitude. There's no harm done, so the upper layers can simply ignore this one. On the other hand, the upper layer may see this as a sign of a major logic error, and treat it as a fatal error. Anything in between these two extremes is also possible.

```
⟨Function definitions 41⟩ +≡
  int marpa_g_precompute(Marpa_Grammar g){ ⟨Return −2 on failure 1229⟩
      int return_value ⇐ failure_indicator;
      struct marpa_obstack *obs_precompute ⇐ marpa_obs_init;
      ⟨Declare precompute variables 373⟩
      ⟨Fail if fatal error 1249⟩
      G_EVENTS_CLEAR(g);
      ⟨Fail if no rules 374⟩
      ⟨Fail if precomputed 1230⟩
      ⟨Fail if bad start symbol 376⟩
      /* After this point, errors are not recoverable */
      ⟨Clear rule duplication tree 122⟩
      /* Phase 1: census the external grammar */
      {     /* Scope with only external grammar */
        ⟨Declare census variables 382⟩
        ⟨Perform census of grammar g 372⟩
        ⟨Detect cycles 448⟩
      }
      /* Phase 2: rewrite the grammar into internal form */
      ⟨Initialize IRL stack 512⟩
      ⟨Initialize NSY stack 513⟩
      ⟨Rewrite grammar g into CHAF form 413⟩
      ⟨Augment grammar g 442⟩
      post_census_xsy_count ⇐ XSY_Count_of_G(g);
      ⟨Populate the event boolean vectors 524⟩
      /* Phase 3: memoize the internal grammar */
      if (¬G_is_Trivial(g)) { ⟨Declare variables for the internal grammar
          memoizations 511⟩
      ⟨Calculate Rule by LHS lists 514⟩
      ⟨Create AHMs 485⟩
      ⟨Construct prediction matrix 517⟩
      ⟨Construct right derivation matrix 507⟩
```

```
⟨ Populate the predicted IRL CIL's in the AHM's 522 ⟩⟨ Populate the terminal
      boolean vector 523 ⟩
⟨ Populate the prediction and nulled symbol CILs 525 ⟩
⟨ Mark the event AHMs 526 ⟩
⟨ Calculate AHM Event Group Sizes 527 ⟩
⟨ Find the direct ZWA's for each AHM 546 ⟩
⟨ Find the indirect ZWA's for each AHM's 547 ⟩
} g→t_is_precomputed ⇐ 1;
if (g→t_has_cycle) {
  MARPA_ERROR(MARPA_ERR_GRAMMAR_HAS_CYCLE);
  goto FAILURE;
}
⟨ Reinitialize the CILAR 369 ⟩
return_value ⇐ 0;
goto CLEANUP;
FAILURE: ;
  goto CLEANUP;
CLEANUP: ;
  marpa_obs_free(obs_precompute);
  return return_value; }
```

369. Reinitialize the CILAR, because its size requirement may vary wildly bewteen a base grammar and its recognizers. A large allocation may be required in the grammar, which thereafter would be wasted space.

⟨ Reinitialize the CILAR 369 ⟩ ≡

```
{
  cilar_buffer_reinit(&g→t_cilar);
}
```

This code is used in section 368.

370. **To Do**: Perhaps someday there should be a CILAR for each recognizer. This probably is an issue to be dealt with, when adding the ability to clone grammars.

371. The grammar census.

372. Implementation: inacessible and unproductive Rules. The textbooks say that, in order to automatically **eliminate** inaccessible and unproductive productions from a grammar, you have to first eliminate the unproductive productions, **then** the inaccessible ones.

In practice, this advice does not seem very helpful. Imagine the (quite possible) case of an unproductive start symbol. Following the correct procedure for automatically cleaning the grammar, I would have to regard the start symbol and its productions as eliminated and therefore go on to report every other production and symbol as inaccessible. Almost certainly all these inaccessiblity reports, while theoretically correct, would be irrelevant. What the user probably wants to is to make the start symbol productive.

In `libmarpa`, inaccessibility is determined based on the assumption that unproductive symbols will be made productive somehow, and not eliminated. The downside of this choice is that, in a few uncommon cases, a user relying entirely on Marpa warnings to clean up his grammar will have to go through more than a single pass of the diagnostics. (As of this writing, I personally have yet to encounter such a case.) The upside is that in the more frequent cases, the user is spared a lot of useless diagnostics.

⟨ Perform census of grammar g 372 ⟩ ≡
{
⟨ Census symbols 380 ⟩
⟨ Census terminals 381 ⟩
⟨ Calculate reach matrix 389 ⟩
⟨ Census nullable symbols 385 ⟩
⟨ Census productive symbols 386 ⟩
⟨ Check that start symbol is productive 387 ⟩
⟨ Census accessible symbols 391 ⟩
⟨ Census nulling symbols 392 ⟩
⟨ Classify rules 393 ⟩
⟨ Mark valued symbols 396 ⟩
⟨ Populate nullification CILs 397 ⟩
}

This code is used in section 368.

373. ⟨ Declare precompute variables 373 ⟩ ≡
XRLID `xrl_count` ⟸ `XRL_Count_of_G`(g);
XSYID `pre_census_xsy_count` ⟸ `XSY_Count_of_G`(g);
XSYID `post_census_xsy_count` ⟸ −1;

See also sections 377 and 390.

This code is used in section 368.

374. ⟨ Fail if no rules 374 ⟩ ≡

```
if (_MARPA_UNLIKELY(xrl_count ≤ 0)) {
  MARPA_ERROR(MARPA_ERR_NO_RULES);
  goto FAILURE;
}
```

This code is used in section 368.

375. Loop over the rules, producing boolean vector of LHS symbols, and of symbols which are the LHS of empty rules. While at it, set a flag to indicate if there are empty rules.

376. ⟨ Fail if bad start symbol 376 ⟩ ≡

```
{
  if (_MARPA_UNLIKELY(start_xsy_id < 0)) {
    MARPA_ERROR(MARPA_ERR_NO_START_SYMBOL);
    goto FAILURE;
  }
  if (_MARPA_UNLIKELY(¬xsy_id_is_valid(g, start_xsy_id))) {
    MARPA_ERROR(MARPA_ERR_INVALID_START_SYMBOL);
    goto FAILURE;
  }
  if (_MARPA_UNLIKELY(¬XSY_is_LHS(XSY_by_ID(start_xsy_id)))) {
    MARPA_ERROR(MARPA_ERR_START_NOT_LHS);
    goto FAILURE;
  }
}
```

This code is used in section 368.

377. ⟨ Declare precompute variables 373 ⟩ +≡

```
XSYID start_xsy_id ⟸ g→t_start_xsy_id;
```

378. Used for sorting RHS symbols for memoization.

⟨ Private structures 48 ⟩ +≡

```
struct sym_rule_pair {
  XSYID t_symid;
  RULEID t_ruleid;
};
```

379. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE_NOT_INLINE int sym_rule_cmp(const void *ap, const void *bp, void
        *param UNUSED)
{
  const struct sym_rule_pair *pair_a ⟸ (struct sym_rule_pair *) ap;
  const struct sym_rule_pair *pair_b ⟸ (struct sym_rule_pair *) bp;
  int result ⟸ pair_a→t_symid − pair_b→t_symid;
```

```
    if (result) return result;
    return pair_a→t_ruleid − pair_b→t_ruleid;
  }
```

380. ⟨ Census symbols 380 ⟩ ≡

```
  {
    Marpa_Rule_ID rule_id;
      /* AVL tree for RHS symbols */
    const MARPA_AVL_TREE rhs_avl_tree ⟸ _marpa_avl_create(sym_rule_cmp, Λ);
        /* Size of G is sum of RHS lengths, plus 1 for each rule, which here is necessary
           for separator of sequences */
    struct sym_rule_pair *const p_rh_sym_rule_pair_base ⟸
        marpa_obs_new(MARPA_AVL_OBSTACK(rhs_avl_tree), struct sym_rule_pair, (size_t)
        External_Size_of_G(g));
    struct sym_rule_pair *p_rh_sym_rule_pairs ⟸ p_rh_sym_rule_pair_base;
      /* AVL tree for LHS symbols */
    const MARPA_AVL_TREE lhs_avl_tree ⟸ _marpa_avl_create(sym_rule_cmp, Λ);
    struct sym_rule_pair *const p_lh_sym_rule_pair_base ⟸
        marpa_obs_new(MARPA_AVL_OBSTACK(lhs_avl_tree), struct sym_rule_pair, (size_t)
        xrl_count);
    struct sym_rule_pair *p_lh_sym_rule_pairs ⟸ p_lh_sym_rule_pair_base;
    lhs_v ⟸ bv_obs_create(obs_precompute, pre_census_xsy_count);
    empty_lhs_v ⟸ bv_obs_shadow(obs_precompute, lhs_v);
    for (rule_id ⟸ 0; rule_id < xrl_count; rule_id++) {
      const XRL rule ⟸ XRL_by_ID(rule_id);
      const Marpa_Symbol_ID lhs_id ⟸ LHS_ID_of_RULE(rule);
      const int rule_length ⟸ Length_of_XRL(rule);
      const int is_sequence ⟸ XRL_is_Sequence(rule);
      bv_bit_set(lhs_v, lhs_id);
      /* Insert the LH Sym / XRL pair into the LH AVL tree */
      p_lh_sym_rule_pairs→t_symid ⟸ lhs_id;
      p_lh_sym_rule_pairs→t_ruleid ⟸ rule_id;
      _marpa_avl_insert(lhs_avl_tree, p_lh_sym_rule_pairs);
      p_lh_sym_rule_pairs++;
      if (is_sequence) {
        const XSYID separator_id ⟸ Separator_of_XRL(rule);
        if (Minimum_of_XRL(rule) ≤ 0) {
          bv_bit_set(empty_lhs_v, lhs_id);
        }
        if (separator_id ≥ 0) {
          p_rh_sym_rule_pairs→t_symid ⟸ separator_id;
          p_rh_sym_rule_pairs→t_ruleid ⟸ rule_id;
          _marpa_avl_insert(rhs_avl_tree, p_rh_sym_rule_pairs);
```

```
      p_rh_sym_rule_pairs++;
    }
  }
  if (rule_length ≤ 0) {
    bv_bit_set(empty_lhs_v, lhs_id);
  }
  else {
    int rhs_ix;
    for (rhs_ix ⟸ 0; rhs_ix < rule_length; rhs_ix++) {
      p_rh_sym_rule_pairs→t_symid ⟸ RHS_ID_of_RULE(rule, rhs_ix);
      p_rh_sym_rule_pairs→t_ruleid ⟸ rule_id;
      _marpa_avl_insert(rhs_avl_tree, p_rh_sym_rule_pairs);
      p_rh_sym_rule_pairs++;
    }
  }
}
{
  MARPA_AVL_TRAV traverser;
  struct sym_rule_pair *pair;
  XSYID seen_symid ⟸ −1;
  RULEID *const rule_data_base ⟸ marpa_obs_new(obs_precompute, RULEID,
      (size_t) External_Size_of_G(g));
  RULEID *p_rule_data ⟸ rule_data_base;
  traverser ⟸ _marpa_avl_t_init(rhs_avl_tree);
  /* One extra "symbol" as an end marker */
  xrl_list_x_rh_sym ⟸ marpa_obs_new(obs_precompute, RULEID *, (size_t)
      pre_census_xsy_count + 1);
  for (pair ⟸ _marpa_avl_t_first(traverser); pair; pair ⟸ (struct
        sym_rule_pair *) _marpa_avl_t_next(traverser)) {
    const XSYID current_symid ⟸ pair→t_symid;
    while (seen_symid < current_symid)
      xrl_list_x_rh_sym[++seen_symid] ⟸ p_rule_data;
    *p_rule_data++ ⟸ pair→t_ruleid;
  }
  while (++seen_symid ≤ pre_census_xsy_count)
    xrl_list_x_rh_sym[seen_symid] ⟸ p_rule_data;
  _marpa_avl_destroy(rhs_avl_tree);
}
{
  MARPA_AVL_TRAV traverser;
  struct sym_rule_pair *pair;
  XSYID seen_symid ⟸ −1;
  RULEID *const rule_data_base ⟸ marpa_obs_new(obs_precompute, RULEID,
      (size_t) xrl_count);
```

```
        RULEID *p_rule_data ⇐ rule_data_base;
        traverser ⇐ _marpa_avl_t_init(lhs_avl_tree);
        /* One extra "symbol" as an end marker */
        xrl_list_x_lh_sym ⇐ marpa_obs_new(obs_precompute, RULEID *, (size_t)
            pre_census_xsy_count + 1);
        for (pair ⇐ _marpa_avl_t_first(traverser); pair; pair ⇐ (struct
              sym_rule_pair *) _marpa_avl_t_next(traverser)) {
          const XSYID current_symid ⇐ pair→t_symid;
          while (seen_symid < current_symid)
            xrl_list_x_lh_sym[++seen_symid] ⇐ p_rule_data;
          *p_rule_data++ ⇐ pair→t_ruleid;
        }
        while (++seen_symid ≤ pre_census_xsy_count)
          xrl_list_x_lh_sym[seen_symid] ⇐ p_rule_data;
        _marpa_avl_destroy(lhs_avl_tree);
      }
    }
```

This code is used in section 372.

381. Loop over the symbols, producing the boolean vector of symbols already marked as terminal, and a flag which indicates if there are any.

⟨ Census terminals 381 ⟩ ≡

```
    {
      XSYID symid;
      terminal_v ⇐ bv_obs_create(obs_precompute, pre_census_xsy_count);
      bv_not(terminal_v, lhs_v);
      for (symid ⇐ 0; symid < pre_census_xsy_count; symid++) {
        XSY symbol ⇐ XSY_by_ID(symid);
        /* If marked by the user, leave the symbol as set by the user, and update the
            boolean vector */
        if (XSY_is_Locked_Terminal(symbol)) {
          if (XSY_is_Terminal(symbol)) {
            bv_bit_set(terminal_v, symid);
            continue;
          }
          bv_bit_clear(terminal_v, symid);
          continue;
        }
        /* If not marked by the user, take the default from the boolean vector and mark
              the symbol, if necessary. */
        if (bv_bit_test(terminal_v, symid)) XSY_is_Terminal(symbol) ⇐ 1;
      }
    }
```

This code is used in section 372.

382. ⟨ Declare census variables 382 ⟩ ≡

```
Bit_Vector terminal_v ⟸ Λ;
```

See also sections 383, 384, and 388.
This code is used in section 368.

383. ⟨ Declare census variables 382 ⟩ +≡

```
Bit_Vector lhs_v ⟸ Λ;
Bit_Vector empty_lhs_v ⟸ Λ;
```

384. These might better be tracked as per-XSY CIL's.

⟨ Declare census variables 382 ⟩ +≡

```
RULEID **xrl_list_x_rh_sym ⟸ Λ;
RULEID **xrl_list_x_lh_sym ⟸ Λ;
```

385. ⟨ Census nullable symbols 385 ⟩ ≡

```
{
  int min, max, start;
  XSYID xsy_id;
  int counted_nullables ⟸ 0;
  nullable_v ⟸ bv_obs_clone(obs_precompute, empty_lhs_v);
  rhs_closure(g, nullable_v, xrl_list_x_rh_sym);
  for (start ⟸ 0; bv_scan(nullable_v, start, &min, &max); start ⟸ max + 2) {
    for (xsy_id ⟸ min; xsy_id ≤ max; xsy_id++) {
      XSY xsy ⟸ XSY_by_ID(xsy_id);
      XSY_is_Nullable(xsy) ⟸ 1;
      if (_MARPA_UNLIKELY(xsy→t_is_counted)) {
        counted_nullables++;
        int_event_new(g, MARPA_EVENT_COUNTED_NULLABLE, xsy_id);
      }
    }
  }
  if (_MARPA_UNLIKELY(counted_nullables)) {
    MARPA_ERROR(MARPA_ERR_COUNTED_NULLABLE);
    goto FAILURE;
  }
}
```

This code is used in section 372.

386. ⟨ Census productive symbols 386 ⟩ ≡

```
{
  productive_v ⟸ bv_obs_shadow(obs_precompute, nullable_v);
  bv_or(productive_v, nullable_v, terminal_v);
  rhs_closure(g, productive_v, xrl_list_x_rh_sym);
  {
    int min, max, start;
```

```
      XSYID symid;
      for (start ⇐ 0; bv_scan(productive_v, start, &min, &max); start ⇐ max + 2)
        {
        for (symid ⇐ min; symid ≤ max; symid++) {
          XSY symbol ⇐ XSY_by_ID(symid);
          symbol→t_is_productive ⇐ 1;
        }
      }
    }
  }
```

This code is used in section 372.

387. ⟨ Check that start symbol is productive 387 ⟩ ≡

```
  if (_MARPA_UNLIKELY(¬bv_bit_test(productive_v, start_xsy_id))) {
    MARPA_ERROR(MARPA_ERR_UNPRODUCTIVE_START);
    goto FAILURE;
  }
```

This code is used in section 372.

388. ⟨ Declare census variables 382 ⟩ +≡

```
  Bit_Vector productive_v ⇐ Λ;
  Bit_Vector nullable_v ⇐ Λ;
```

389. The reach matrix is the an $n \times n$ matrix, where n is the number of symbols. Bit (i, j) is set in the reach matrix if and only if symbol i can reach symbol j.

This logic could be put earlier, and a child array for each rule could be efficiently calculated during the initialization for the calculation of the reach matrix. A rule-child array is a list of the rule's RHS symbols, in sequence and without duplicates. There are places were traversing a rule-child array, instead of the rhs, would be more efficient. At this point, however, it is not clear whether use of a rule-child array is not a pointless or even counter-productive optimization. It would only make a difference in grammars where many of the right hand sides repeat symbols.

⟨ Calculate reach matrix 389 ⟩ ≡

```
  {
    XRLID rule_id;
    reach_matrix ⇐ matrix_obs_create(obs_precompute, pre_census_xsy_count,
        pre_census_xsy_count);
    for (rule_id ⇐ 0; rule_id < xrl_count; rule_id++) {
      XRL rule ⇐ XRL_by_ID(rule_id);
      XSYID lhs_id ⇐ LHS_ID_of_RULE(rule);
      int rhs_ix;
      int rule_length ⇐ Length_of_XRL(rule);
      for (rhs_ix ⇐ 0; rhs_ix < rule_length; rhs_ix++) {
        matrix_bit_set(reach_matrix, lhs_id, RHS_ID_of_RULE(rule, rhs_ix));
```

```
      }
      if (XRL_is_Sequence(rule)) {
        const XSYID separator_id ⟸ Separator_of_XRL(rule);
        if (separator_id ≥ 0) {
          matrix_bit_set(reach_matrix, lhs_id, separator_id);
        }
      }
    }
    transitive_closure(reach_matrix);
  }
```

This code is used in section 372.

390. ⟨ Declare precompute variables 373 ⟩ +≡

```
  Bit_Matrix reach_matrix ⟸ Λ;
```

391. accessible_v is a pointer into the reach_matrix. Therefore there is no code to free it.

⟨ Census accessible symbols 391 ⟩ ≡

```
  {
    Bit_Vector accessible_v ⟸ matrix_row(reach_matrix, start_xsy_id);
    int min, max, start;
    XSYID symid;
    for (start ⟸ 0; bv_scan(accessible_v, start, &min, &max); start ⟸ max + 2)
      {
      for (symid ⟸ min; symid ≤ max; symid++) {
        XSY symbol ⟸ XSY_by_ID(symid);
        symbol→t_is_accessible ⟸ 1;
      }
    }
    XSY_by_ID(start_xsy_id)→t_is_accessible ⟸ 1;
  }
```

This code is used in section 372.

392. A symbol is nulling if and only if it is an LHS symbol which does not reach a terminal symbol.

⟨ Census nulling symbols 392 ⟩ ≡

```
  {
    Bit_Vector reaches_terminal_v ⟸ bv_shadow(terminal_v);
    int nulling_terminal_found ⟸ 0;
    int min, max, start;
    for (start ⟸ 0; bv_scan(lhs_v, start, &min, &max); start ⟸ max + 2) {
      XSYID productive_id;
      for (productive_id ⟸ min; productive_id ≤ max; productive_id++) {
```

```
        bv_and(reaches_terminal_v, terminal_v, matrix_row(reach_matrix,
            productive_id));
        if (bv_is_empty(reaches_terminal_v)) {
          const XSY symbol ⟸ XSY_by_ID(productive_id);
          XSY_is_Nulling(symbol) ⟸ 1;
          if (_MARPA_UNLIKELY(XSY_is_Terminal(symbol))) {
            nulling_terminal_found ⟸ 1;
            int_event_new(g, MARPA_EVENT_NULLING_TERMINAL, productive_id);
          }
        }
      }
    }
    bv_free(reaches_terminal_v);
    if (_MARPA_UNLIKELY(nulling_terminal_found)) {
      MARPA_ERROR(MARPA_ERR_NULLING_TERMINAL);
      goto FAILURE;
    }
  }
```

This code is used in section 372.

393. A rule is accessible if its LHS is accessible. A rule is nulling if every symbol on its RHS is nulling. A rule is productive if every symbol on its RHS is productive. Note that these can be vacuously true — an empty rule is nulling and productive.

⟨ Classify rules 393 ⟩ ≡

```
  {
    XRLID xrl_id;
    for (xrl_id ⟸ 0; xrl_id < xrl_count; xrl_id++) {
      const XRL xrl ⟸ XRL_by_ID(xrl_id);
      const XSYID lhs_id ⟸ LHS_ID_of_XRL(xrl);
      const XSY lhs ⟸ XSY_by_ID(lhs_id);
      XRL_is_Accessible(xrl) ⟸ XSY_is_Accessible(lhs);
      if (XRL_is_Sequence(xrl)) {
        ⟨ Classify sequence rule 395 ⟩
        continue;
      }
      ⟨ Classify BNF rule 394 ⟩
    }
  }
```

This code is used in section 372.

394. Accessibility was determined in outer loop. Classify as nulling, nullable or productive.

⟨ Classify BNF rule 394 ⟩ ≡

```
{
  int rh_ix;
  int is_nulling ⇐ 1;
  int is_nullable ⇐ 1;
  int is_productive ⇐ 1;
  for (rh_ix ⇐ 0; rh_ix < Length_of_XRL(xrl); rh_ix++) {
    const XSYID rhs_id ⇐ RHS_ID_of_XRL(xrl, rh_ix);
    const XSY rh_xsy ⇐ XSY_by_ID(rhs_id);
    if (_MARPA_LIKELY(¬XSY_is_Nulling(rh_xsy))) is_nulling ⇐ 0;
    if (_MARPA_LIKELY(¬XSY_is_Nullable(rh_xsy))) is_nullable ⇐ 0;
    if (_MARPA_UNLIKELY(¬XSY_is_Productive(rh_xsy))) is_productive ⇐ 0;
  }
  XRL_is_Nulling(xrl) ⇐ Boolean(is_nulling);
  XRL_is_Nullable(xrl) ⇐ Boolean(is_nullable);
  XRL_is_Productive(xrl) ⇐ Boolean(is_productive);
  XRL_is_Used(xrl) ⇐ XRL_is_Accessible(xrl) ∧ XRL_is_Productive(xrl) ∧
      ¬XRL_is_Nulling(xrl);
}
```

This code is used in section 393.

395. Accessibility was determined in outer loop. Classify as nulling, nullable or productive. In the case of an unproductive separator, we could create a "degenerate" sequence, allowing only those sequence which don't require separators. (These are sequences of length 0 and 1.) But currently we don't both – we just mark the rule unproductive.

⟨ Classify sequence rule 395 ⟩ ≡

```
{
  const XSYID rhs_id ⇐ RHS_ID_of_XRL(xrl, 0);
  const XSY rh_xsy ⇐ XSY_by_ID(rhs_id);
  const XSYID separator_id ⇐ Separator_of_XRL(xrl);
    /* A sequence rule is nullable if it can be zero length or if its RHS is nullable */
  XRL_is_Nullable(xrl) ⇐ Minimum_of_XRL(xrl) ≤ 0 ∨ XSY_is_Nullable(rh_xsy);

    /* A sequence rule is nulling if its RHS is nulling */
  XRL_is_Nulling(xrl) ⇐ XSY_is_Nulling(rh_xsy);
    /* A sequence rule is productive if it is nulling or if its RHS is productive */
  XRL_is_Productive(xrl) ⇐ XRL_is_Nullable(xrl) ∨ XSY_is_Productive(rh_xsy);
    /* Initialize to used if accessible and RHS is productive */
  XRL_is_Used(xrl) ⇐ XRL_is_Accessible(xrl) ∧ XSY_is_Productive(rh_xsy);
```

```
    /* Touch-ups to account for the separator */
  if (separator_id ≥ 0) {
    const XSY separator_xsy ⇐ XSY_by_ID(separator_id);
    /* A non-nulling separator means a non-nulling rule */
    if (¬XSY_is_Nulling(separator_xsy)) {
      XRL_is_Nulling(xrl) ⇐ 0;
    }
    /* A unproductive separator means a unproductive rule, unless it is nullable. */
    if (_MARPA_UNLIKELY(¬XSY_is_Productive(separator_xsy))) {
      XRL_is_Productive(xrl) ⇐ XRL_is_Nullable(xrl);
    /* Do not use a sequence rule with an unproductive separator */
      XRL_is_Used(xrl) ⇐ 0;
    }
  }
    /* Do not use if nulling */
  if (XRL_is_Nulling(xrl)) XRL_is_Used(xrl) ⇐ 0;
}
```

This code is used in section 393.

396. Those LHS terminals that have not been explicitly marked (as indicated by their "valued locked" bit), should be marked valued and locked. This is to follow the principle of least surprise. A recognizer might mark these symbols as unvalued, prior to valuator trying to assign semantics to rules with them on the LHS. Better to mark them valued now, and cause an error in the recognizer.

⟨ Mark valued symbols 396 ⟩ ≡

```
if (0) {
    /* Commented out. The LHS terminal user is a sophisticated user so it is probably
       the better course to allow her the choice. */
  XSYID xsy_id;
  for (xsy_id ⇐ 0; xsy_id < pre_census_xsy_count; xsy_id++) {
    if (bv_bit_test(terminal_v, xsy_id) ∧ bv_bit_test(lhs_v, xsy_id)) {
      const XSY xsy ⇐ XSY_by_ID(xsy_id);
      if (XSY_is_Valued_Locked(xsy)) continue;
      XSY_is_Valued(xsy) ⇐ 1;
      XSY_is_Valued_Locked(xsy) ⇐ 1;
    }
  }
}
```

This code is used in section 372.

397. An XSY A nullifies XSY B if the fact that A is nulled implies that B is nulled as well. This may happen trivially – a nullable symbol nullifies itself. And it may happen through a nullable derivation. The derivation may be ambiguous – in other words, A nullifies B if a nulled B can be derived from a nulled A. Change so that this runs only if there are prediction events.

⟨ Populate nullification CILs 397 ⟩ ≡

```
{
  XSYID xsyid;
  XRLID xrlid;
    /* Use this to make sure we have enough CILAR buffer space */
  int nullable_xsy_count ⇐ 0;
    /* This matrix is large and very temporary, so it does not go on the obstack */
  void *matrix_buffer ⇐ my_malloc(matrix_sizeof(pre_census_xsy_count,
      pre_census_xsy_count));
  Bit_Matrix nullification_matrix ⇐ matrix_buffer_create(matrix_buffer,
      pre_census_xsy_count, pre_census_xsy_count);
  for (xsyid ⇐ 0; xsyid < pre_census_xsy_count; xsyid++) {
      /* Every nullable symbol symbol nullifies itself */
    if (¬XSYID_is_Nullable(xsyid)) continue;
    nullable_xsy_count++;
    matrix_bit_set(nullification_matrix, xsyid, xsyid);
  }
  for (xrlid ⇐ 0; xrlid < xrl_count; xrlid++) {
    int rh_ix;
    XRL xrl ⇐ XRL_by_ID(xrlid);
    const XSYID lhs_id ⇐ LHS_ID_of_XRL(xrl);
    if (XRL_is_Nullable(xrl)) {
      for (rh_ix ⇐ 0; rh_ix < Length_of_XRL(xrl); rh_ix++) {
        const XSYID rhs_id ⇐ RHS_ID_of_XRL(xrl, rh_ix);
        matrix_bit_set(nullification_matrix, lhs_id, rhs_id);
      }
    }
  }
  transitive_closure(nullification_matrix);
  for (xsyid ⇐ 0; xsyid < pre_census_xsy_count; xsyid++) {
    Bit_Vector bv_nullifications_by_to_xsy ⇐
        matrix_row(nullification_matrix, xsyid);
    Nulled_XSYIDs_of_XSYID(xsyid) ⇐ cil_bv_add(&g→t_cilar,
        bv_nullifications_by_to_xsy);
  }
  my_free(matrix_buffer);
}
```

This code is used in section 372.

398. The sequence rewrite.

⟨ Rewrite sequence rule into BNF 398 ⟩ ≡

```
{
  const XSYID lhs_id ⟸ LHS_ID_of_RULE(rule);
  const NSY lhs_nsy ⟸ NSY_by_XSYID(lhs_id);
  const NSYID lhs_nsyid ⟸ ID_of_NSY(lhs_nsy);
  const NSY internal_lhs_nsy ⟸ nsy_new(g, XSY_by_ID(lhs_id));
  const NSYID internal_lhs_nsyid ⟸ ID_of_NSY(internal_lhs_nsy);
  const XSYID rhs_id ⟸ RHS_ID_of_RULE(rule, 0);
  const NSY rhs_nsy ⟸ NSY_by_XSYID(rhs_id);
  const NSYID rhs_nsyid ⟸ ID_of_NSY(rhs_nsy);
  const XSYID separator_id ⟸ Separator_of_XRL(rule);
  NSYID separator_nsyid ⟸ −1;
  if (separator_id ≥ 0) {
    const NSY separator_nsy ⟸ NSY_by_XSYID(separator_id);
    separator_nsyid ⟸ ID_of_NSY(separator_nsy);
  }
  LHS_XRL_of_NSY(internal_lhs_nsy) ⟸ rule;
  ⟨ Add the top rule for the sequence 399 ⟩
  if (separator_nsyid ≥ 0 ∧ ¬XRL_is_Proper_Separation(rule)) {
    ⟨ Add the alternate top rule for the sequence 400 ⟩
  }
  ⟨ Add the minimum rule for the sequence 401 ⟩
  ⟨ Add the iterating rule for the sequence 402 ⟩
}
```

This code is used in section 413.

399. ⟨ Add the top rule for the sequence 399 ⟩ ≡

```
{
  IRL rewrite_irl ⟸ irl_start(g, 1);
  LHSID_of_IRL(rewrite_irl) ⟸ lhs_nsyid;
  RHSID_of_IRL(rewrite_irl, 0) ⟸ internal_lhs_nsyid;
  irl_finish(g, rewrite_irl);
  Source_XRL_of_IRL(rewrite_irl) ⟸ rule;
  Rank_of_IRL(rewrite_irl) ⟸ IRL_Rank_by_XRL(rule);
    /* Real symbol count remains at default of 0 */
  IRL_has_Virtual_RHS(rewrite_irl) ⟸ 1;
}
```

This code is used in section 398.

400. This "alternate" top rule is needed if a final separator is allowed.

⟨ Add the alternate top rule for the sequence 400 ⟩ ≡

```
{
  IRL rewrite_irl;
  rewrite_irl ⟸ irl_start(g, 2);
  LHSID_of_IRL(rewrite_irl) ⟸ lhs_nsyid;
  RHSID_of_IRL(rewrite_irl, 0) ⟸ internal_lhs_nsyid;
  RHSID_of_IRL(rewrite_irl, 1) ⟸ separator_nsyid;
  irl_finish(g, rewrite_irl);
  Source_XRL_of_IRL(rewrite_irl) ⟸ rule;
  Rank_of_IRL(rewrite_irl) ⟸ IRL_Rank_by_XRL(rule);
  IRL_has_Virtual_RHS(rewrite_irl) ⟸ 1;
  Real_SYM_Count_of_IRL(rewrite_irl) ⟸ 1;
}
```

This code is used in section 398.

401. The traditional way to write a sequence in BNF is with one rule to represent the minimum, and another to deal with iteration. That's the core of Marpa's rewrite.

⟨ Add the minimum rule for the sequence 401 ⟩ ≡

```
{
  const IRL rewrite_irl ⟸ irl_start(g, 1);
  LHSID_of_IRL(rewrite_irl) ⟸ internal_lhs_nsyid;
  RHSID_of_IRL(rewrite_irl, 0) ⟸ rhs_nsyid;
  irl_finish(g, rewrite_irl);
  Source_XRL_of_IRL(rewrite_irl) ⟸ rule;
  Rank_of_IRL(rewrite_irl) ⟸ IRL_Rank_by_XRL(rule);
  IRL_has_Virtual_LHS(rewrite_irl) ⟸ 1;
  Real_SYM_Count_of_IRL(rewrite_irl) ⟸ 1;
}
```

This code is used in section 398.

402. ⟨ Add the iterating rule for the sequence 402 ⟩ ≡

```
{
  IRL rewrite_irl;
  int rhs_ix ⟸ 0;
  const int length ⟸ separator_nsyid ≥ 0 ? 3 : 2;
  rewrite_irl ⟸ irl_start(g, length);
  LHSID_of_IRL(rewrite_irl) ⟸ internal_lhs_nsyid;
  RHSID_of_IRL(rewrite_irl, rhs_ix++) ⟸ internal_lhs_nsyid;
  if (separator_nsyid ≥ 0)
    RHSID_of_IRL(rewrite_irl, rhs_ix++) ⟸ separator_nsyid;
  RHSID_of_IRL(rewrite_irl, rhs_ix) ⟸ rhs_nsyid;
  irl_finish(g, rewrite_irl);
  Source_XRL_of_IRL(rewrite_irl) ⟸ rule;
```

```
    Rank_of_IRL(rewrite_irl) ⟸ IRL_Rank_by_XRL(rule);
    IRL_has_Virtual_LHS(rewrite_irl) ⟸ 1;
    IRL_has_Virtual_RHS(rewrite_irl) ⟸ 1;
    Real_SYM_Count_of_IRL(rewrite_irl) ⟸ length − 1;
  }
```

This code is used in section 398.

403. The CHAF rewrite.
Nullable symbols have been a difficulty for Earley implementations since day zero. Aycock and Horspool came up with a solution to this problem, part of which involved rewriting the grammar to eliminate all proper nullables. Marpa's CHAF rewrite is built on the work of Aycock and Horspool.

Marpa's CHAF rewrite is one of its two rewrites of the BNF. The other adds a new start symbol to the grammar.

404. The rewrite strategy for Marpa is new to it. It is an elaboration on the one developed by Aycock and Horspool. The basic idea behind Aycock and Horspool's NNF was to elimnate proper nullables by replacing the rules with variants which used only nulling and non-nulling symbols. These had to be created for every possible combination of nulling and non-nulling symbols. This meant that the number of NNF rules was potentially exponential in the length of rule of the original grammar.

405. Marpa's CHAF (Chomsky-Horspool-Aycock Form) eliminates the problem of exponential explosion by first breaking rules up into pieces, each piece containing no more than two proper nullables. The number of rewritten rules in CHAF in linear in the length of the original rule.

406. The CHAF rewrite affects only rules with proper nullables. In this context, the proper nullables are called "factors". Each piece of the original rule is rewritten into up to four "factored pieces". When there are two proper nullables, the potential CHAF rules are

- The PP rule: Both factors are replaced with non-nulling symbols.
- The PN rule: The first factor is replaced with a non-nulling symbol, and the second factor is replaced with a nulling symbol.
- The NP rule: The first factor is replaced with a nulling symbol, and the second factor is replaced with a non-nulling symbol.
- The NN rule: Both factors are replaced with nulling symbols.

407. Sometimes the CHAF piece will have only one factor. A one-factor piece is rewritten into at most two factored pieces:

- The P rule: The factor is replaced with a non-nulling symbol.
- The N rule: The factor is replaced with a nulling symbol.

408. In `CHAF_rewrite`, a `rule_count` is taken before the loop over the grammar's rules, even though rules are added in the loop. This is not an error. The CHAF rewrite is not recursive – the new rules it creates are not themselves subject to CHAF rewrite. And rule ID's increase by one each time, so that all the new rules will have ID's equal to or greater than the pre-CHAF rule count.

409. Is this a CHAF IRL?. Is this IRL a product of the CHAF rewrite?

#define `IRL_is_CHAF(irl)` `((irl)→t_is_chaf)`

⟨ Bit aligned IRL elements 341 ⟩ +≡
  BITFIELD `t_is_chaf`:1;

410. ⟨ Initialize IRL elements 342 ⟩ +≡

```
IRL_is_CHAF(irl) ⟸ 0;
```

411. ⟨ Public function prototypes 411 ⟩ ≡

```
int _marpa_g_irl_is_chaf(Marpa_Grammar g, Marpa_IRL_ID irl_id);
```

See also sections 1352, 1354, 1360, and 1362.

This code is used in section 1387.

412. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_irl_is_chaf(Marpa_Grammar g, Marpa_IRL_ID irl_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if irl_id is invalid 1238 ⟩
  return IRL_is_CHAF(IRL_by_ID(irl_id));
}
```

413. ⟨ Rewrite grammar g into CHAF form 413 ⟩ ≡

```
{
  ⟨ CHAF rewrite declarations 414 ⟩
  ⟨ CHAF rewrite allocations 418 ⟩
  ⟨ Clone external symbols 415 ⟩
  pre_chaf_rule_count ⟸ XRL_Count_of_G(g);
  for (rule_id ⟸ 0; rule_id < pre_chaf_rule_count; rule_id++) {
    XRL rule ⟸ XRL_by_ID(rule_id);
    XRL rewrite_xrl ⟸ rule;
    const int rewrite_xrl_length ⟸ Length_of_XRL(rewrite_xrl);
    int nullable_suffix_ix ⟸ 0;

    if (¬XRL_is_Used(rule)) continue;
    if (XRL_is_Sequence(rule)) {
      ⟨ Rewrite sequence rule into BNF 398 ⟩
      continue;
    }
    ⟨ Calculate CHAF rule statistics 416 ⟩
      /* Do not factor if there is no proper nullable in the rule */
    if (factor_count > 0) {
      ⟨ Factor the rule into CHAF rules 419 ⟩
      continue;
    }
    ⟨ Clone a new IRL from rule 260 ⟩
  }
}
```

This code is used in section 368.

414. ⟨ CHAF rewrite declarations 414 ⟩ ≡

```
Marpa_Rule_ID rule_id;
int pre_chaf_rule_count;
```

See also section 417.
This code is used in section 413.

415. For every accessible and productive proper nullable which is not already aliased, alias it.

⟨ Clone external symbols 415 ⟩ ≡

```
{
  XSYID xsy_id;
  for (xsy_id ⇐ 0; xsy_id < pre_census_xsy_count; xsy_id++) {
    const XSY xsy_to_clone ⇐ XSY_by_ID(xsy_id);
    if (_MARPA_UNLIKELY(¬xsy_to_clone→t_is_accessible)) continue;
    if (_MARPA_UNLIKELY(¬xsy_to_clone→t_is_productive)) continue;
    NSY_of_XSY(xsy_to_clone) ⇐ nsy_clone(g, xsy_to_clone);
    if (XSY_is_Nulling(xsy_to_clone)) {
      Nulling_NSY_of_XSY(xsy_to_clone) ⇐ NSY_of_XSY(xsy_to_clone);
      continue;
    }
    if (XSY_is_Nullable(xsy_to_clone)) {
      Nulling_NSY_of_XSY(xsy_to_clone) ⇐ symbol_alias_create(g,
          xsy_to_clone);
    }
  }
}
```

This code is used in section 413.

416. Compute statistics needed to rewrite the nule. The term "factor" is used to mean an instance of a proper nullable symbol on the RHS of a rule. This comes from the idea that replacing the proper nullables with proper symbols and nulling symbols "factors" pieces of the rule being rewritten (the original rule) into multiple CHAF rules.

⟨ Calculate CHAF rule statistics 416 ⟩ ≡

```
{
  int rhs_ix;
  factor_count ⇐ 0;
  for (rhs_ix ⇐ 0; rhs_ix < rewrite_xrl_length; rhs_ix++) {
    Marpa_Symbol_ID symid ⇐ RHS_ID_of_RULE(rule, rhs_ix);
    XSY symbol ⇐ XSY_by_ID(symid);
    if (XSY_is_Nulling(symbol)) continue;     /* Do nothing for nulling symbols */
    if (XSY_is_Nullable(symbol)) {     /* If a proper nullable, record its position */
      factor_positions[factor_count++] ⇐ rhs_ix;
      continue;
    }
```

```
      nullable_suffix_ix ⇐ rhs_ix + 1;    /* If not a nullable symbol, move
          forward the index of the nullable suffix location */
    }
  }
```

This code is used in section 413.

417. ⟨ CHAF rewrite declarations 414 ⟩ +≡

```
  int factor_count;
  int *factor_positions;
```

418. ⟨ CHAF rewrite allocations 418 ⟩ ≡

```
  factor_positions ⇐ marpa_obs_new(obs_precompute, int, g→t_max_rule_length);
```

This code is used in section 413.

419. Divide the rule into pieces.

⟨ Factor the rule into CHAF rules 419 ⟩ ≡

```
  {
    const XRL chaf_xrl ⇐ rule;    /* The number of proper nullables for which
        CHAF rules have yet to be written */
    int unprocessed_factor_count;    /* Current index into the list of factors */
    int factor_position_ix ⇐ 0;
    NSY current_lhs_nsy ⇐ NSY_by_XSYID(LHS_ID_of_RULE(rule));
    NSYID current_lhs_nsyid ⇐ ID_of_NSY(current_lhs_nsy);    /* The positions,
        in the original rule, where the new (virtual) rule starts and ends */
    int piece_end, piece_start ⇐ 0;
    for (unprocessed_factor_count ⇐ factor_count − factor_position_ix;
          unprocessed_factor_count ≥ 3; unprocessed_factor_count ⇐
          factor_count − factor_position_ix) {
      ⟨ Add non-final CHAF rules 422 ⟩
    }
    if (unprocessed_factor_count ≡ 2) {
      ⟨ Add final CHAF rules for two factors 432 ⟩
    }
    else {
      ⟨ Add final CHAF rules for one factor 437 ⟩
    }
  }
```

This code is used in section 413.

420. ⟨ Create a CHAF virtual symbol 420 ⟩ ≡

```
  {
    const XSYID chaf_xrl_lhs_id ⇐ LHS_ID_of_XRL(chaf_xrl);
    chaf_virtual_nsy ⇐ nsy_new(g, XSY_by_ID(chaf_xrl_lhs_id));
    chaf_virtual_nsyid ⇐ ID_of_NSY(chaf_virtual_nsy);
  }
```

This code is used in section 422.

421. Factor a non-final piece.

422. As long as I have more than 3 unprocessed factors, I am working on a non-final rule.

⟨ Add non-final CHAF rules 422 ⟩ ≡
```
NSY chaf_virtual_nsy;
NSYID chaf_virtual_nsyid;
int first_factor_position ⇐ factor_positions[factor_position_ix];
int second_factor_position ⇐ factor_positions[factor_position_ix + 1];
if (second_factor_position ≥ nullable_suffix_ix) {
  piece_end ⇐ second_factor_position − 1;
    /* The last factor is in the nullable suffix, so the virtual RHS must be nullable */
  ⟨ Create a CHAF virtual symbol 420 ⟩
  ⟨ Add CHAF rules for nullable continuation 423 ⟩
  factor_position_ix++;
}
else {
  piece_end ⇐ second_factor_position;
  ⟨ Create a CHAF virtual symbol 420 ⟩
  ⟨ Add CHAF rules for proper continuation 427 ⟩
  factor_position_ix += 2;
}
current_lhs_nsy ⇐ chaf_virtual_nsy;
current_lhs_nsyid ⇐ chaf_virtual_nsyid;
piece_start ⇐ piece_end + 1;
```
This code is used in section 419.

423. Add CHAF rules for nullable continuations. For a piece that has a nullable continuation, the virtual RHS counts as one of the two allowed proper nullables. That means the piece must end before the second proper nullable (or factor).

⟨ Add CHAF rules for nullable continuation 423 ⟩ ≡
```
{
  {
    const int real_symbol_count ⇐ piece_end − piece_start + 1;
    ⟨ Add PP CHAF rule for proper continuation 428 ⟩;
  }
  ⟨ Add PN CHAF rule for nullable continuation 424 ⟩;
  {
    const int real_symbol_count ⇐ piece_end − piece_start + 1;
    ⟨ Add NP CHAF rule for proper continuation 430 ⟩;
  }
  ⟨ Add NN CHAF rule for nullable continuation 425 ⟩;
}
```
This code is used in section 422.

424. ⟨ Add PN CHAF rule for nullable continuation 424 ⟩ ≡

```
{
  int piece_ix;
  const int second_nulling_piece_ix ⇐ second_factor_position − piece_start;
  const int chaf_irl_length ⇐ rewrite_xrl_length − piece_start;
  const int real_symbol_count ⇐ chaf_irl_length;
  IRL chaf_irl ⇐ irl_start(g, chaf_irl_length);
  LHSID_of_IRL(chaf_irl) ⇐ current_lhs_nsyid;
  for (piece_ix ⇐ 0; piece_ix < second_nulling_piece_ix; piece_ix++) {
    RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + piece_ix));
  }
  for (piece_ix ⇐ second_nulling_piece_ix; piece_ix < chaf_irl_length;
        piece_ix++) {
    RHSID_of_IRL(chaf_irl,
        piece_ix) ⇐ Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + piece_ix));
  }
  irl_finish(g, chaf_irl);
  Rank_of_IRL(chaf_irl) ⇐ IRL_CHAF_Rank_by_XRL(rule, 2);
  ⟨ Add CHAF IRL 440 ⟩
}
```

This code is used in section 423.

425. If this piece is nullable (piece_start at or after nullable_suffix_ix), I don't add an NN choice, because nulling both factors makes the entire piece nulling, and nulling rules cannot be fed directly to the Marpa parse engine.

⟨ Add NN CHAF rule for nullable continuation 425 ⟩ ≡

```
{
  if (piece_start < nullable_suffix_ix) {
    int piece_ix;
    const int first_nulling_piece_ix ⇐ first_factor_position − piece_start;
    const int second_nulling_piece_ix ⇐ second_factor_position − piece_start;
    const int chaf_irl_length ⇐ rewrite_xrl_length − piece_start;
    const int real_symbol_count ⇐ chaf_irl_length;
    IRL chaf_irl ⇐ irl_start(g, chaf_irl_length);
    LHSID_of_IRL(chaf_irl) ⇐ current_lhs_nsyid;
    for (piece_ix ⇐ 0; piece_ix < first_nulling_piece_ix; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl, first_nulling_piece_ix) ⇐
        Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + first_nulling_piece_ix));
```

```
      for (piece_ix ⟸ first_nulling_piece_ix + 1;
            piece_ix < second_nulling_piece_ix; piece_ix++) {
        RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
            piece_start + piece_ix));
      }
      for (piece_ix ⟸ second_nulling_piece_ix; piece_ix < chaf_irl_length;
            piece_ix++) {
        RHSID_of_IRL(chaf_irl,
            piece_ix) ⟸ Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
            piece_start + piece_ix));
      }
      irl_finish(g, chaf_irl);
      Rank_of_IRL(chaf_irl) ⟸ IRL_CHAF_Rank_by_XRL(rule, 0);
      ⟨ Add CHAF IRL 440 ⟩
    }
  }
```

This code is used in section 423.

426. Add CHAF rules for proper continuations.

427. Open block and declarations.

```
⟨ Add CHAF rules for proper continuation 427 ⟩ ≡
  {
    const int real_symbol_count ⟸ piece_end − piece_start + 1;
    ⟨ Add PP CHAF rule for proper continuation 428 ⟩
    ⟨ Add PN CHAF rule for proper continuation 429 ⟩
    ⟨ Add NP CHAF rule for proper continuation 430 ⟩
    ⟨ Add NN CHAF rule for proper continuation 431 ⟩
  }
```

This code is used in section 422.

428. The PP Rule.

```
⟨ Add PP CHAF rule for proper continuation 428 ⟩ ≡
  {
    int piece_ix;
    const int chaf_irl_length ⟸ (piece_end − piece_start) + 2;
    IRL chaf_irl ⟸ irl_start(g, chaf_irl_length);
    LHSID_of_IRL(chaf_irl) ⟸ current_lhs_nsyid;
    for (piece_ix ⟸ 0; piece_ix < chaf_irl_length − 1; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl, chaf_irl_length − 1) ⟸ chaf_virtual_nsyid;
    irl_finish(g, chaf_irl);
    Rank_of_IRL(chaf_irl) ⟸ IRL_CHAF_Rank_by_XRL(rule, 3);
```

```
    ⟨ Add CHAF IRL 440 ⟩
  }
```

This code is used in sections 423 and 427.

429. The PN Rule.

⟨ Add PN CHAF rule for proper continuation 429 ⟩ ≡

```
  {
    int piece_ix;
    const int second_nulling_piece_ix ⇐ second_factor_position − piece_start;
    const int chaf_irl_length ⇐ (piece_end − piece_start) + 2;
    IRL chaf_irl ⇐ irl_start(g, chaf_irl_length);
    LHSID_of_IRL(chaf_irl) ⇐ current_lhs_nsyid;
    for (piece_ix ⇐ 0; piece_ix < second_nulling_piece_ix; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl,
        second_nulling_piece_ix) ⇐ Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + second_nulling_piece_ix));
    for (piece_ix ⇐ second_nulling_piece_ix + 1; piece_ix < chaf_irl_length − 1;
          piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl, chaf_irl_length − 1) ⇐ chaf_virtual_nsyid;
    irl_finish(g, chaf_irl);
    Rank_of_IRL(chaf_irl) ⇐ IRL_CHAF_Rank_by_XRL(rule, 2);
    ⟨ Add CHAF IRL 440 ⟩
  }
```

This code is used in section 427.

430. The NP Rule.

⟨ Add NP CHAF rule for proper continuation 430 ⟩ ≡

```
  {
    int piece_ix;
    const int first_nulling_piece_ix ⇐ first_factor_position − piece_start;
    const int chaf_irl_length ⇐ (piece_end − piece_start) + 2;
    IRL chaf_irl ⇐ irl_start(g, chaf_irl_length);
    LHSID_of_IRL(chaf_irl) ⇐ current_lhs_nsyid;
    for (piece_ix ⇐ 0; piece_ix < first_nulling_piece_ix; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
```

```
    RHSID_of_IRL(chaf_irl,
        first_nulling_piece_ix) ⟸ Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + first_nulling_piece_ix));
    for (piece_ix ⟸ first_nulling_piece_ix + 1; piece_ix < chaf_irl_length − 1;
          piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl, chaf_irl_length − 1) ⟸ chaf_virtual_nsyid;
    irl_finish(g, chaf_irl);
    Rank_of_IRL(chaf_irl) ⟸ IRL_CHAF_Rank_by_XRL(rule, 1);
    ⟨ Add CHAF IRL 440 ⟩
  }
```

This code is used in sections 423 and 427.

431. The NN Rule.

⟨ Add NN CHAF rule for proper continuation 431 ⟩ ≡

```
  {
    int piece_ix;
    const int first_nulling_piece_ix ⟸ first_factor_position − piece_start;
    const int second_nulling_piece_ix ⟸ second_factor_position − piece_start;
    const int chaf_irl_length ⟸ (piece_end − piece_start) + 2;
    IRL chaf_irl ⟸ irl_start(g, chaf_irl_length);
    LHSID_of_IRL(chaf_irl) ⟸ current_lhs_nsyid;
    for (piece_ix ⟸ 0; piece_ix < first_nulling_piece_ix; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl,
        first_nulling_piece_ix) ⟸ Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + first_nulling_piece_ix));
    for (piece_ix ⟸ first_nulling_piece_ix + 1;
          piece_ix < second_nulling_piece_ix; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl,
        second_nulling_piece_ix) ⟸ Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + second_nulling_piece_ix));
    for (piece_ix ⟸ second_nulling_piece_ix + 1; piece_ix < chaf_irl_length − 1;
          piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl, chaf_irl_length − 1) ⟸ chaf_virtual_nsyid;
```

```
    irl_finish(g, chaf_irl);
    Rank_of_IRL(chaf_irl) ⟸ IRL_CHAF_Rank_by_XRL(rule, 0);
    ⟨ Add CHAF IRL 440 ⟩
  }
```

This code is used in section 427.

432. Add final CHAF rules for two factors. Open block, declarations and setup.

```
⟨ Add final CHAF rules for two factors 432 ⟩ ≡
  {
    const int first_factor_position ⟸ factor_positions[factor_position_ix];
    const int second_factor_position ⟸ factor_positions[factor_position_ix + 1];
    const int real_symbol_count ⟸ Length_of_XRL(rule) − piece_start;
    piece_end ⟸ Length_of_XRL(rule) − 1;
    ⟨ Add final CHAF PP rule for two factors 433 ⟩
    ⟨ Add final CHAF PN rule for two factors 434 ⟩
    ⟨ Add final CHAF NP rule for two factors 435 ⟩
    ⟨ Add final CHAF NN rule for two factors 436 ⟩
  }
```

This code is used in section 419.

433. The PP Rule.

```
⟨ Add final CHAF PP rule for two factors 433 ⟩ ≡
  {
    int piece_ix;
    const int chaf_irl_length ⟸ (piece_end − piece_start) + 1;
    IRL chaf_irl ⟸ irl_start(g, chaf_irl_length);
    LHSID_of_IRL(chaf_irl) ⟸ current_lhs_nsyid;
    for (piece_ix ⟸ 0; piece_ix < chaf_irl_length; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    irl_finish(g, chaf_irl);
    Rank_of_IRL(chaf_irl) ⟸ IRL_CHAF_Rank_by_XRL(rule, 3);
    ⟨ Add CHAF IRL 440 ⟩
  }
```

This code is used in section 432.

434. The PN Rule.

```
⟨ Add final CHAF PN rule for two factors 434 ⟩ ≡
  {
    int piece_ix;
    const int second_nulling_piece_ix ⟸ second_factor_position − piece_start;
    const int chaf_irl_length ⟸ (piece_end − piece_start) + 1;
    IRL chaf_irl ⟸ irl_start(g, chaf_irl_length);
```

```
    LHSID_of_IRL(chaf_irl) ⟸ current_lhs_nsyid;
    for (piece_ix ⟸ 0; piece_ix < second_nulling_piece_ix; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl,
        second_nulling_piece_ix) ⟸ Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + second_nulling_piece_ix));
    for (piece_ix ⟸ second_nulling_piece_ix + 1; piece_ix < chaf_irl_length;
          piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    irl_finish(g, chaf_irl);
    Rank_of_IRL(chaf_irl) ⟸ IRL_CHAF_Rank_by_XRL(rule, 2);
    ⟨ Add CHAF IRL 440 ⟩
  }
```

This code is used in section 432.

435. The NP Rule.

```
⟨ Add final CHAF NP rule for two factors 435 ⟩ ≡
  {
    int piece_ix;
    const int first_nulling_piece_ix ⟸ first_factor_position − piece_start;
    const int chaf_irl_length ⟸ (piece_end − piece_start) + 1;
    IRL chaf_irl ⟸ irl_start(g, chaf_irl_length);
    LHSID_of_IRL(chaf_irl) ⟸ current_lhs_nsyid;
    for (piece_ix ⟸ 0; piece_ix < first_nulling_piece_ix; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl,
        first_nulling_piece_ix) ⟸ Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + first_nulling_piece_ix));
    for (piece_ix ⟸ first_nulling_piece_ix + 1; piece_ix < chaf_irl_length;
          piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⟸ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    irl_finish(g, chaf_irl);
    Rank_of_IRL(chaf_irl) ⟸ IRL_CHAF_Rank_by_XRL(rule, 1);
    ⟨ Add CHAF IRL 440 ⟩
  }
```

This code is used in section 432.

436. The NN Rule. This is added only if it would not turn this into a nulling rule.

⟨ Add final CHAF NN rule for two factors 436 ⟩ ≡

```
{
  if (piece_start < nullable_suffix_ix) {
    int piece_ix;
    const int first_nulling_piece_ix ⇐ first_factor_position − piece_start;
    const int second_nulling_piece_ix ⇐ second_factor_position − piece_start;
    const int chaf_irl_length ⇐ (piece_end − piece_start) + 1;
    IRL chaf_irl ⇐ irl_start(g, chaf_irl_length);
    LHSID_of_IRL(chaf_irl) ⇐ current_lhs_nsyid;
    for (piece_ix ⇐ 0; piece_ix < first_nulling_piece_ix; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl, first_nulling_piece_ix) ⇐
        Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + first_nulling_piece_ix));
    for (piece_ix ⇐ first_nulling_piece_ix + 1;
          piece_ix < second_nulling_piece_ix; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    RHSID_of_IRL(chaf_irl, second_nulling_piece_ix) ⇐
        Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
        piece_start + second_nulling_piece_ix));
    for (piece_ix ⇐ second_nulling_piece_ix + 1; piece_ix < chaf_irl_length;
          piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    irl_finish(g, chaf_irl);
    Rank_of_IRL(chaf_irl) ⇐ IRL_CHAF_Rank_by_XRL(rule, 0);
    ⟨ Add CHAF IRL 440 ⟩
  }
}
```

This code is used in section 432.

437. Add final CHAF rules for one factor.

⟨ Add final CHAF rules for one factor 437 ⟩ ≡

```
{
  int real_symbol_count;
  const int first_factor_position ⇐ factor_positions[factor_position_ix];
  piece_end ⇐ Length_of_XRL(rule) − 1;
  real_symbol_count ⇐ piece_end − piece_start + 1;
```

```
    ⟨ Add final CHAF P rule for one factor 438 ⟩
    ⟨ Add final CHAF N rule for one factor 439 ⟩
  }
```

This code is used in section 419.

438. The P Rule.

⟨ Add final CHAF P rule for one factor 438 ⟩ ≡

```
  {
    int piece_ix;
    const int chaf_irl_length ⇐ (piece_end − piece_start) + 1;
    IRL chaf_irl ⇐ irl_start(g, chaf_irl_length);
    LHSID_of_IRL(chaf_irl) ⇐ current_lhs_nsyid;
    for (piece_ix ⇐ 0; piece_ix < chaf_irl_length; piece_ix++) {
      RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + piece_ix));
    }
    irl_finish(g, chaf_irl);
    Rank_of_IRL(chaf_irl) ⇐ IRL_CHAF_Rank_by_XRL(rule, 3);
    ⟨ Add CHAF IRL 440 ⟩
  }
```

This code is used in section 437.

439. The N Rule. This is added only if it would not turn this into a nulling rule.

⟨ Add final CHAF N rule for one factor 439 ⟩ ≡

```
  {
    if (piece_start < nullable_suffix_ix) {
      int piece_ix;
      const int nulling_piece_ix ⇐ first_factor_position − piece_start;
      const int chaf_irl_length ⇐ (piece_end − piece_start) + 1;
      IRL chaf_irl ⇐ irl_start(g, chaf_irl_length);
      LHSID_of_IRL(chaf_irl) ⇐ current_lhs_nsyid;
      for (piece_ix ⇐ 0; piece_ix < nulling_piece_ix; piece_ix++) {
        RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
            piece_start + piece_ix));
      }
      RHSID_of_IRL(chaf_irl,
          nulling_piece_ix) ⇐ Nulling_NSYID_by_XSYID(RHS_ID_of_RULE(rule,
          piece_start + nulling_piece_ix));
      for (piece_ix ⇐ nulling_piece_ix + 1; piece_ix < chaf_irl_length;
            piece_ix++) {
        RHSID_of_IRL(chaf_irl, piece_ix) ⇐ NSYID_by_XSYID(RHS_ID_of_RULE(rule,
            piece_start + piece_ix));
      }
      irl_finish(g, chaf_irl);
```

```
      Rank_of_IRL(chaf_irl) ⟸ IRL_CHAF_Rank_by_XRL(rule, 0);
      ⟨ Add CHAF IRL 440 ⟩
    }
  }
```

This code is used in section 437.

440. Some of the code for adding CHAF rules is common to them all. This include the setting of many of the elements of the rule structure, and performing the call back.

⟨ Add CHAF IRL 440 ⟩ ≡

```
  {
    const int is_virtual_lhs ⟸ (piece_start > 0);
    IRL_is_CHAF(chaf_irl) ⟸ 1;
    Source_XRL_of_IRL(chaf_irl) ⟸ rule;
    IRL_has_Virtual_LHS(chaf_irl) ⟸ Boolean(is_virtual_lhs);
    IRL_has_Virtual_RHS(chaf_irl) ⟸ Length_of_IRL(chaf_irl) >
        real_symbol_count;
    Virtual_Start_of_IRL(chaf_irl) ⟸ piece_start;
    Virtual_End_of_IRL(chaf_irl) ⟸ piece_start + real_symbol_count − 1;
    Real_SYM_Count_of_IRL(chaf_irl) ⟸ real_symbol_count;
    LHS_XRL_of_NSY(current_lhs_nsy) ⟸ chaf_xrl;
    XRL_Offset_of_NSY(current_lhs_nsy) ⟸ piece_start;
  }
```

This code is used in sections 424, 425, 428, 429, 430, 431, 433, 434, 435, 436, 438, and 439.

441. Adding a new start symbol. This is such a common rewrite that it has a special name in the literature — it is called "augmenting the grammar".

442. ⟨ Augment grammar g 442 ⟩ ≡

```
{
  const XSY start_xsy ⟸ XSY_by_ID(start_xsy_id);
  if (_MARPA_LIKELY(¬XSY_is_Nulling(start_xsy))) {
    ⟨ Set up a new proper start rule 443 ⟩
  }
}
```

This code is used in section 368.

443. ⟨ Set up a new proper start rule 443 ⟩ ≡

```
{
  IRL new_start_irl;
  const NSY new_start_nsy ⟸ nsy_new(g, start_xsy);
  NSY_is_Start(new_start_nsy) ⟸ 1;
  new_start_irl ⟸ irl_start(g, 1);
  LHSID_of_IRL(new_start_irl) ⟸ ID_of_NSY(new_start_nsy);
  RHSID_of_IRL(new_start_irl, 0) ⟸ NSYID_of_XSY(start_xsy);
  irl_finish(g, new_start_irl);
  IRL_has_Virtual_LHS(new_start_irl) ⟸ 1;
  Real_SYM_Count_of_IRL(new_start_irl) ⟸ 1;
  g→t_start_irl ⟸ new_start_irl;
}
```

This code is used in section 442.

444. Loops. Loops are rules which non-trivially derive their own LHS. More precisely, a rule is a loop if and only if it non-trivially derives a string which contains its LHS symbol and is of length 1. In my experience, and according to Grune and Jacobs 2008 (pp. 48-49), loops are never of practical use.

445. Marpa allows loops, for two reasons. First, I want to be able to claim that Marpa handles **all** context-free grammars. This is of real value to the user, because it makes it very easy for her to know beforehand whether Marpa can handle a particular grammar. If she can write the grammar in BNF, then Marpa can handle it — it's that simple. For Marpa to make this claim, it must be able to handle grammars with loops.

Second, a user's drafts of a grammar might contain cycles. A parser generator which did not handle them would force the user's first order of business to be removing them. That might be inconvenient.

446. The grammar precomputations and the recognition phase have been set up so that loops are a complete non-issue — they are dealt with like any other situation, without additional overhead. However, loops do impose overhead and require special handling in the evaluation phase. It is unlikely that a user will want to leave one in a production grammar.

447. Marpa detects all loops during its grammar precomputation. `libmarpa` assumes that parsing will go through as usual, with the loops. But it enables the upper layers to make other choices.

448. The higher layers can differ greatly in their treatment of loop rules. It is perfectly reasonable for a higher layer to treat a loop rule as a fatal error. It is also reasonable for a higher layer to always silently allow them. There are lots of possibilities in between these two extremes. To assist the upper layers, an event is reported for a non-zero loop rule count, with the final tally.

```
⟨ Detect cycles 448 ⟩ ≡
  {
    int loop_rule_count ⇐ 0;
    Bit_Matrix unit_transition_matrix ⇐ matrix_obs_create(obs_precompute,
        xrl_count, xrl_count);
    ⟨ Mark direct unit transitions in unit_transition_matrix 449 ⟩
    transitive_closure(unit_transition_matrix);
    ⟨ Mark loop rules 451 ⟩
    if (loop_rule_count) {
      g→t_has_cycle ⇐ 1;
      int_event_new(g, MARPA_EVENT_LOOP_RULES, loop_rule_count);
    }
  }
```

This code is used in section 368.

449. Note that direct transitions are marked in advance, but not trivial ones. That is, bit (x, x) is not set true in advance. In other words, for this purpose, unit transitions are not in general reflexive.

```
⟨ Mark direct unit transitions in unit_transition_matrix 449 ⟩ ≡
{
  Marpa_Rule_ID rule_id;
  for (rule_id ⟸ 0; rule_id < xrl_count; rule_id++) {
    XRL rule ⟸ XRL_by_ID(rule_id);
    XSYID nonnullable_id ⟸ −1;
    int nonnullable_count ⟸ 0;
    int rhs_ix, rule_length;
    rule_length ⟸ Length_of_XRL(rule);
    /* Count the non-nullable rules */
    for (rhs_ix ⟸ 0; rhs_ix < rule_length; rhs_ix++) {
      XSYID xsy_id ⟸ RHS_ID_of_RULE(rule, rhs_ix);
      if (bv_bit_test(nullable_v, xsy_id)) continue;
      nonnullable_id ⟸ xsy_id;
      nonnullable_count++;
    }
    if (nonnullable_count ≡ 1) {
    /* If exactly one RHS symbol is non-nullable, it is a unit transition, and the only
           one for this rule */
      ⟨ For nonnullable_id, set to-, from-rule bit in unit_transition_matrix 450 ⟩
    }
    else if (nonnullable_count ≡ 0) {
      for (rhs_ix ⟸ 0; rhs_ix < rule_length; rhs_ix++) {
    /* If exactly zero RHS symbols are non-nullable, all the proper nullables (that is,
           nullables which are not nulling) are are potential unit transitions */
        nonnullable_id ⟸ RHS_ID_of_RULE(rule, rhs_ix);
        if (XSY_is_Nulling(XSY_by_ID(nonnullable_id))) continue;
    /* If here, nonnullable_id is a proper nullable */
        ⟨ For nonnullable_id, set to-, from-rule bit in unit_transition_matrix 450 ⟩
      }
    }
  }
}
```

This code is used in section 448.

450. We have a lone `nonnullable_id` in `rule_id`, so there is a unit transition from `rule_id` to every rule with `nonnullable_id` on the LHS.

⟨ For `nonnullable_id`, set to-, from-rule bit in `unit_transition_matrix` 450 ⟩ ≡

```
{
  RULEID *p_xrl ⟸ xrl_list_x_lh_sym[nonnullable_id];
  const RULEID *p_one_past_rules ⟸ xrl_list_x_lh_sym[nonnullable_id + 1];
  for ( ; p_xrl < p_one_past_rules; p_xrl++) {
      /* Direct loops (A → A) only need the (rule_id, rule_id) bit set, but it is not clear
         that it is a win to special case them. */
    const RULEID to_rule_id ⟸ *p_xrl;
    matrix_bit_set(unit_transition_matrix, rule_id, to_rule_id);
  }
}
```

This code is used in section 449.

451. ⟨ Mark loop rules 451 ⟩ ≡

```
{
  XRLID rule_id;
  for (rule_id ⟸ 0; rule_id < xrl_count; rule_id++) {
    XRL rule;
    if (¬matrix_bit_test(unit_transition_matrix, rule_id, rule_id)) continue;
    loop_rule_count++;
    rule ⟸ XRL_by_ID(rule_id);
    rule→t_is_loop ⟸ 1;
  }
}
```

This code is used in section 448.

452. Aycock-Horspool item (AHM) code. These were formerly called AHFA items, where AHFA stood for "Aycock-Horspool finite automaton". The finite automaton is not longer in use, but its special items (dotted rules which ignore nullables) remain very much a part of Marpa's parsing strategy.

```
⟨ Public typedefs 91 ⟩ +≡
  typedef int Marpa_AHM_ID;
```

453.

```
⟨ Private structures 48 ⟩ +≡
  struct s_ahm {
    ⟨ Widely aligned AHM elements 462 ⟩
    ⟨ Int aligned AHM elements 463 ⟩
    ⟨ Bit aligned AHM elements 477 ⟩
  };
```

454.

```
⟨ Private incomplete structures 107 ⟩ +≡
  struct s_ahm;
  typedef struct s_ahm *AHM;
  typedef Marpa_AHM_ID AHMID;
```

455. Because AHM's are in an array, the predecessor can be found by incrementing the AHM pointer, the successor can be found by decrementing it, and AHM pointers can be portably compared. A lot of code relies on these facts.

```
#define AHM_by_ID(id) (g→t_ahms + (id))
#define ID_of_AHM(ahm) (AHMID)((ahm) − g→t_ahms)
```

456. These require the caller to make sure all the *AHM*'s involved exist.

```
#define Next_AHM_of_AHM(ahm) ((ahm) + 1)
#define Prev_AHM_of_AHM(ahm) ((ahm) − 1)
⟨ Widely aligned grammar elements 59 ⟩ +≡
  AHM t_ahms;
```

457.

```
#define AHM_Count_of_G(g) ((g)→t_ahm_count)
⟨ Int aligned grammar elements 53 ⟩ +≡
  int t_ahm_count;
```

458. The space is allocated during precomputation. Because the grammar may be destroyed before precomputation, I test that `g→t_ahms` is non-zero.

459.

```
⟨ Initialize grammar elements 54 ⟩ +≡
  g→t_ahms ⟸ Λ;
```

460.

```
⟨ Destroy grammar elements 61 ⟩ +≡
  my_free(g→t_ahms);
```

461. Check that AHM ID is in valid range.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int ahm_is_valid(GRAMMAR g, AHMID item_id)
{
  return item_id < (AHMID) AHM_Count_of_G(g) ∧ item_id ≥ 0;
}
```

462. Rule.

```
#define IRL_of_AHM(ahm) ((ahm)→t_irl)
#define IRLID_of_AHM(item) ID_of_IRL(IRL_of_AHM(item))
#define LHS_NSYID_of_AHM(item) LHSID_of_IRL(IRL_of_AHM(item))
#define LHSID_of_AHM(item) LHS_NSYID_of_AHM(item)
```

⟨ Widely aligned AHM elements 462 ⟩ ≡

```
IRL t_irl;
```

See also sections 475, 476, 496, 500, and 503.

This code is used in section 453.

463. Postdot symbol. −1 if the item is a completion.

```
#define Postdot_NSYID_of_AHM(item) ((item)→t_postdot_nsyid)
#define AHM_is_Completion(ahm) (Postdot_NSYID_of_AHM(ahm) < 0)
#define AHM_is_Leo(ahm) (IRL_is_Leo(IRL_of_AHM(ahm)))
#define AHM_is_Leo_Completion(ahm) (AHM_is_Completion(ahm) ∧ AHM_is_Leo(ahm))
```

⟨ Int aligned AHM elements 463 ⟩ ≡

```
NSYID t_postdot_nsyid;
```

See also sections 464, 465, 467, 469, 501, and 504.

This code is used in section 453.

464. Leading nulls. In libmarpa's AHM's, the dot position is never in front of a nulling symbol. (Due to rewriting, every nullable symbol is also a nulling symbol.) This element contains the count of nulling symbols preceding this AHM's dot position.

```
#define Null_Count_of_AHM(ahm) ((ahm)→t_leading_nulls)
```

⟨ Int aligned AHM elements 463 ⟩ +≡

```
int t_leading_nulls;
```

465. RHS Position. RHS position, including nulling symbols. Position in the RHS, -1 for a completion. Raw position is the same as position except for completions, in which case it is the length of the IRL.

```
#define Position_of_AHM(ahm) ((ahm)→t_position)
#define Raw_Position_of_AHM(ahm)
        (Position_of_AHM(ahm) < 0 ? ((Length_of_IRL(IRL_of_AHM(ahm))) +
            Position_of_AHM(ahm) + 1) : Position_of_AHM(ahm))
```

⟨ Int aligned AHM elements 463 ⟩ +≡

```
int t_position;
```

466. Note the difference between `AHM_was_Predicted` and `AHM_is_Prediction`. `AHM_is_Prediction` indicates whether the dotted rule is a prediction. `AHM_was_Predicted` indicates whether the AHM is the result of a prediction. In the case of the start AHM, it is result of Initialization.

#define `AHM_is_Prediction(ahm)` `(Quasi_Position_of_AHM(ahm)` $\equiv 0$`)`

467. Quasi-position. Quasi-positions are positions calculated without counting nulling symbols.

#define `Quasi_Position_of_AHM(ahm)` `((ahm)→t_quasi_position)`

⟨ Int aligned AHM elements 463 ⟩ +≡
int `t_quasi_position;`

468. Symbol Instance. The symbol instance identifies the instance of a symbol in the internal grammar, That is, it identifies not just the symbol, but the specific use of a symbol in a rule. The SYMI count differs from the AHM count, in that predictions are not included, but nulling symbols are. Predictions are not included, because the count is of predot symbols. The symbol instance of a prediction is set to -1.

469. Symbol instances are for the **predot** symbol because symbol instances are used in evaluation. In parsing the emphasis is on what is to come — on what follows the dot. In evaluation we are looking at what we have, so the emphasis is on what precedes the dot position.

#define `SYMI_of_AHM(ahm)` `((ahm)→t_symbol_instance)`

⟨ Int aligned AHM elements 463 ⟩ +≡
int `t_symbol_instance;`

470. ⟨ Private typedefs 49 ⟩ +≡
typedef int SYMI;

471. *#define* `SYMI_Count_of_G(`g`)` `((`g`)→t_symbol_instance_count)`

⟨ Int aligned grammar elements 53 ⟩ +≡
int `t_symbol_instance_count;`

472. *#define* `SYMI_of_IRL(irl)` `((irl)→t_symbol_instance_base)`
#define `Last_Proper_SYMI_of_IRL(irl)` `((irl)→t_last_proper_symi)`
#define `SYMI_of_Completed_IRL(irl)` `(SYMI_of_IRL(irl) + Length_of_IRL(irl) − 1)`

⟨ Int aligned IRL elements 329 ⟩ +≡
int `t_symbol_instance_base;`
int `t_last_proper_symi;`

473. ⟨ Initialize IRL elements 342 ⟩ +≡
`Last_Proper_SYMI_of_IRL(irl)` ⟸ -1;

474. Predicted IRL's. One CIL representing the predicted IRL's, and another representing the directly predicted IRL's. Both are empty CIL if there are no predictions.

475. To Do: It is not clear whether both of these will be needed, or if not, which one will be needed.

```
#define Predicted_IRL_CIL_of_AHM(ahm) ((ahm)→t_predicted_irl_cil)
#define LHS_CIL_of_AHM(ahm) ((ahm)→t_lhs_cil)
```

⟨ Widely aligned AHM elements 462 ⟩ +≡

```
CIL t_predicted_irl_cil;
CIL t_lhs_cil;
```

476. Zero-width assertions at this AHM. A CIL representing the zero-width assertions at this AHM. The empty CIL if there are none.

```
#define ZWA_CIL_of_AHM(ahm) ((ahm)→t_zwa_cil)
```

⟨ Widely aligned AHM elements 462 ⟩ +≡

```
CIL t_zwa_cil;
```

477. Does this AHM predict any zero-width assertions?. A flag indicating that some of the predictions from this AHM may have zero-width assertions. Note this boolean is independent of whether the AHM itself has zero-width assertions.

```
#define AHM_predicts_ZWA(ahm) ((ahm)→t_predicts_zwa)
```

⟨ Bit aligned AHM elements 477 ⟩ ≡

```
BITFIELD t_predicts_zwa:1;
```

See also section 499.

This code is used in section 453.

478. AHM external accessors.

⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_ahm_count(Marpa_Grammar g)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  return AHM_Count_of_G(g);
}
```

479. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_IRL_ID _marpa_g_ahm_irl(Marpa_Grammar g, Marpa_AHM_ID item_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if item_id is invalid 1244 ⟩
  return IRLID_of_AHM(AHM_by_ID(item_id));
}
```

480. −1 is the value for completions, so −2 is the failure indicator.

481. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_g_ahm_position(Marpa_Grammar g, Marpa_AHM_ID item_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if item_id is invalid 1244 ⟩
  return Position_of_AHM(AHM_by_ID(item_id));
}
```

482. −1 is the value for completions, so −2 is the failure indicator.

483. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Symbol_ID _marpa_g_ahm_postdot(Marpa_Grammar g, Marpa_AHM_ID
        item_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  ⟨ Fail if item_id is invalid 1244 ⟩
  return Postdot_NSYID_of_AHM(AHM_by_ID(item_id));
}
```

484. Creating the AHMs.

485. ⟨ Create AHMs 485 ⟩ ≡

```
{
  IRLID irl_id;
  int ahm_count ⇐ 0;
  AHM base_item;
  AHM current_item;
  int symbol_instance_of_next_rule ⇐ 0;
  for (irl_id ⇐ 0; irl_id < irl_count; irl_id++) {
    const IRL irl ⇐ IRL_by_ID(irl_id);
    ⟨ Count the AHMs in a rule 487 ⟩
  }
  current_item ⇐ base_item ⇐ marpa_new(struct s_ahm, ahm_count);
  for (irl_id ⇐ 0; irl_id < irl_count; irl_id++) {
    const IRL irl ⇐ IRL_by_ID(irl_id);
    SYMI_of_IRL(irl) ⇐ symbol_instance_of_next_rule;
    ⟨ Create the AHMs for irl 486 ⟩
    {
      symbol_instance_of_next_rule += Length_of_IRL(irl);
    }
  }
  SYMI_Count_of_G(g) ⇐ symbol_instance_of_next_rule;
  MARPA_ASSERT(ahm_count ≡ current_item − base_item);
  AHM_Count_of_G(g) ⇐ ahm_count;
  g→t_ahms ⇐ marpa_renew(struct s_ahm, base_item, ahm_count);
  ⟨ Populate the first AHM's of the RULE's 493 ⟩
}
```

This code is used in section 368.

486. ⟨ Create the AHMs for irl 486 ⟩ ≡

```
{
  int leading_nulls ⇐ 0;
  int rhs_ix;
  const AHM first_ahm_of_irl ⇐ current_item;
  for (rhs_ix ⇐ 0; rhs_ix < Length_of_IRL(irl); rhs_ix++) {
    NSYID rh_nsyid ⇐ RHSID_of_IRL(irl, rhs_ix);
    if (¬NSY_is_Nulling(NSY_by_ID(rh_nsyid))) {
      Last_Proper_SYMI_of_IRL(irl) ⇐ symbol_instance_of_next_rule + rhs_ix;
      ⟨ Create an AHM for a precompletion 488 ⟩
      current_item++;
      leading_nulls ⇐ 0;
    }
    else {
```

```
        leading_nulls++;
      }
    }
    ⟨ Create an AHM for a completion 489 ⟩
    current_item++;
    AHM_Count_of_IRL(irl) ⟸ (int)(current_item − first_ahm_of_irl);
  }
```

This code is used in section 485.

487. ⟨ Count the AHMs in a rule 487 ⟩ ≡

```
  {
    int rhs_ix;
    for (rhs_ix ⟸ 0; rhs_ix < Length_of_IRL(irl); rhs_ix++) {
      const NSYID rh_nsyid ⟸ RHSID_of_IRL(irl, rhs_ix);
      const NSY nsy ⟸ NSY_by_ID(rh_nsyid);
      if (¬NSY_is_Nulling(nsy)) ahm_count++;
    }
    ahm_count++;
  }
```

This code is used in section 485.

488. ⟨ Create an AHM for a precompletion 488 ⟩ ≡

```
  {
    ⟨ Initializations common to all AHMs 490 ⟩
    AHM_predicts_ZWA(current_item) ⟸ 0;
      /* Initially unset, this bit will be populated later. */
    Postdot_NSYID_of_AHM(current_item) ⟸ rh_nsyid;
    Position_of_AHM(current_item) ⟸ rhs_ix;
    SYMI_of_AHM(current_item) ⟸ AHM_is_Prediction(current_item) ? −1 :
        SYMI_of_IRL(irl) + Position_of_AHM(current_item − 1);
    memoize_xrl_data_for_AHM(current_item, irl);
  }
```

This code is used in section 486.

489. ⟨ Create an AHM for a completion 489 ⟩ ≡

```
  {
    ⟨ Initializations common to all AHMs 490 ⟩
    Postdot_NSYID_of_AHM(current_item) ⟸ −1;
    Position_of_AHM(current_item) ⟸ −1;
    SYMI_of_AHM(current_item) ⟸ SYMI_of_IRL(irl) +
        Position_of_AHM(current_item − 1);
    memoize_xrl_data_for_AHM(current_item, irl);
  }
```

This code is used in section 486.

490. ⟨ Initializations common to all AHMs 490 ⟩ ≡

```
{
  IRL_of_AHM(current_item) ⟸ irl;
  Null_Count_of_AHM(current_item) ⟸ leading_nulls;
  Quasi_Position_of_AHM(current_item) ⟸ (int)(current_item −
      first_ahm_of_irl);
  if (Quasi_Position_of_AHM(current_item) ≡ 0) {
    if (ID_of_IRL(irl) ≡ ID_of_IRL(g→t_start_irl)) {
      AHM_was_Predicted(current_item) ⟸ 0;
      AHM_is_Initial(current_item) ⟸ 1;
    }
    else {
      AHM_was_Predicted(current_item) ⟸ 1;
      AHM_is_Initial(current_item) ⟸ 0;
    }
  }
  else {
    AHM_was_Predicted(current_item) ⟸ 0;
    AHM_is_Initial(current_item) ⟸ 0;
  }
  ⟨ Initialize event data for current_item 505 ⟩
}
```

This code is used in sections 488 and 489.

491. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void memoize_xrl_data_for_AHM(AHM current_item, IRL irl)
{
  XRL source_xrl ⟸ Source_XRL_of_IRL(irl);
  XRL_of_AHM(current_item) ⟸ source_xrl;
  if (¬source_xrl) {
    /* source_xrl ⟸ Λ, which is the case only for the start rule */
    XRL_Position_of_AHM(current_item) ⟸ −2;
    return;
  }
  {
    const int virtual_start ⟸ Virtual_Start_of_IRL(irl);
    const int irl_position ⟸ Position_of_AHM(current_item);
    if (XRL_is_Sequence(source_xrl)) {
      /* Note that a sequence XRL, because of the way it is rewritten, may have several
           IRL's, and therefore several AHM's at position 0. */
      XRL_Position_of_AHM(current_item) ⟸ irl_position ? −1 : 0;
      return;
    }
```

```
    /* Completed CHAF rules are a special case */
    if (IRL_is_CHAF(irl) ∧ (irl_position < 0 ∨ irl_position ≥ Length_of_IRL(irl)))
      {
      XRL_Position_of_AHM(current_item) ⟸ −1;
      return;
    }
    if (virtual_start ≥ 0) {
      XRL_Position_of_AHM(current_item) ⟸ irl_position + virtual_start;
      return;
    }
    XRL_Position_of_AHM(current_item) ⟸ irl_position;
  }
  return;
}
```

492. This is done after creating the AHMs, because in theory the `marpa_renew` might have moved them. This is not likely since the `marpa_renew` shortened the array, but if you are hoping for portability, you want to follow the rules.

493. Walks backwards through the *AHM*'s, setting each to the the first of its *IRL*. Last setting wins, which works since we are traversing backwards.

⟨Populate the first *AHM*'s of the *RULE*'s 493⟩ ≡

```
{
  AHM items ⟸ g→t_ahms;
  AHMID item_id ⟸ (AHMID) ahm_count;
  for (item_id−−; item_id ≥ 0; item_id−−) {
    AHM item ⟸ items + item_id;
    IRL irl ⟸ IRL_of_AHM(item);
    First_AHM_of_IRL(irl) ⟸ item;
  }
}
```

This code is used in section 485.

494. XSYID Events.

495.

```
#define Completion_XSYIDs_of_AHM(ahm) ((ahm)→t_completion_xsyids)
#define Nulled_XSYIDs_of_AHM(ahm) ((ahm)→t_nulled_xsyids)
#define Prediction_XSYIDs_of_AHM(ahm) ((ahm)→t_prediction_xsyids)
```

496. ⟨Widely aligned AHM elements 462⟩ +≡

```
  CIL t_completion_xsyids;
  CIL t_nulled_xsyids;
  CIL t_prediction_xsyids;
```

497. AHM container.

498. What is source of the AHM?.

499. These macros and booleans indicates source, not contents. In particular "was predicted" means was the result of a prediction, and does not always indicate whether the AHM or YIM contains a prediction. This is relevant in the case of the the initial AHM, which contains a prediction, but for which "was predicted" is false.

```
#define AHM_was_Predicted(ahm) ((ahm)→t_was_predicted)
#define YIM_was_Predicted(yim) AHM_was_Predicted(AHM_of_YIM(yim))
#define AHM_is_Initial(ahm) ((ahm)→t_is_initial)
#define YIM_is_Initial(yim) AHM_is_Initial(AHM_of_YIM(yim))
```

⟨ Bit aligned AHM elements 477 ⟩ +≡

```
  BITFIELD t_was_predicted:1;
  BITFIELD t_is_initial:1;
```

500. We memoize the XRL data for the AHM, XRL position is complicated to compute, and it depends on XRL – in particular if the XRL is Λ, XRL position is not defined.

```
#define XRL_of_AHM(ahm) ((ahm)→t_xrl)
```

⟨ Widely aligned AHM elements 462 ⟩ +≡

```
  XRL t_xrl;
```

501.

```
#define XRL_Position_of_AHM(ahm) ((ahm)→t_xrl_position)
#define Raw_XRL_Position_of_AHM(ahm)
        (XRL_Position_of_AHM(ahm) < 0 ? Length_of_XRL(XRL_of_AHM(ahm)) :
            XRL_Position_of_AHM(ahm))
```

⟨ Int aligned AHM elements 463 ⟩ +≡

```
  int t_xrl_position;
```

502. Event data. A boolean tracks whether this is an "event AHM", that is, whether there is an event for this AHM itself. Even an non-event AHM may be part of an "event group". In this context, the subset of event AHMs in an AHM's right recursion group is called an "event group". These data are used in various optimizations – the event processing can ignore AHM's without events.

```
#define Event_Group_Size_of_AHM(ahm) ((ahm)→t_event_group_size)
#define Event_AHMIDs_of_AHM(ahm) ((ahm)→t_event_ahmids)
#define AHM_has_Event(ahm) (Count_of_CIL(Event_AHMIDs_of_AHM(ahm)) ≠ 0)
```

503. This CIL is at most of size 1. It is either the singleton containing the AHM's own ID, or the empty CIL.

⟨ Widely aligned AHM elements 462 ⟩ +≡

```
  CIL t_event_ahmids;
```

504. A counter tracks the number of AHMs in this AHM's event group.

```
⟨ Int aligned AHM elements 463 ⟩ +≡
  int t_event_group_size;
```

505.
```
⟨ Initialize event data for current_item 505 ⟩ ≡
  Event_AHMIDs_of_AHM(current_item) ⇐ Λ;
  Event_Group_Size_of_AHM(current_item) ⇐ 0;
```
This code is used in section 490.

506. The NSY right derivation matrix. The NSY right derivation matrix is used in determining which states are Leo completions. The bit for the (nsy1, nsy2) duple is set if and only if nsy1 right derives a sentential form whose rightmost non-null symbol is nsy2. Trivial derivations are included – the bit is set if nsy1 = nsy2.

507.
```
⟨ Construct right derivation matrix 507 ⟩ ≡
  {
    nsy_by_right_nsy_matrix ⇐ matrix_obs_create(obs_precompute, nsy_count,
        nsy_count);
    ⟨ Initialize the nsy_by_right_nsy_matrix for right derivations 508 ⟩
    transitive_closure(nsy_by_right_nsy_matrix);
    ⟨ Mark the right recursive IRLs 509 ⟩
    matrix_clear(nsy_by_right_nsy_matrix);
    ⟨ Initialize the nsy_by_right_nsy_matrix for right recursions 510 ⟩
    transitive_closure(nsy_by_right_nsy_matrix);
  }
```
This code is used in section 368.

508.
```
⟨ Initialize the nsy_by_right_nsy_matrix for right derivations 508 ⟩ ≡
  {
    IRLID irl_id;
    for (irl_id ⇐ 0; irl_id < irl_count; irl_id++) {
      const IRL irl ⇐ IRL_by_ID(irl_id);
      int rhs_ix;
      for (rhs_ix ⇐ Length_of_IRL(irl) − 1; rhs_ix ≥ 0; rhs_ix−−) {
          /* LHS right dervies the last non-nulling symbol. There is at least one
            non-nulling symbol in each IRL. */
        const NSYID rh_nsyid ⇐ RHSID_of_IRL(irl, rhs_ix);
        if (¬NSY_is_Nulling(NSY_by_ID(rh_nsyid))) {
          matrix_bit_set(nsy_by_right_nsy_matrix, LHSID_of_IRL(irl), rh_nsyid);
          break;
        }
      }
    }
  }
```
This code is used in section 507.

509. ⟨ Mark the right recursive IRLs 509 ⟩ ≡

```
{
  IRLID irl_id;
  for (irl_id ⟸ 0; irl_id < irl_count; irl_id++) {
    const IRL irl ⟸ IRL_by_ID(irl_id);
    int rhs_ix;
    for (rhs_ix ⟸ Length_of_IRL(irl) − 1; rhs_ix ≥ 0; rhs_ix−−) {
      const NSYID rh_nsyid ⟸ RHSID_of_IRL(irl, rhs_ix);
      if (¬NSY_is_Nulling(NSY_by_ID(rh_nsyid))) {
          /* Does the last non-nulling symbol right derive the LHS? If so, the rule is
             right recursive. (There is at least one non-nulling symbol in each IRL.) */
        if (matrix_bit_test(nsy_by_right_nsy_matrix, rh_nsyid,
              LHSID_of_IRL(irl))) {
          IRL_is_Right_Recursive(irl) ⟸ 1;
        }
        break;
      }
    }
  }
}
```

This code is used in section 507.

510. ⟨ Initialize the nsy_by_right_nsy_matrix for right recursions 510 ⟩ ≡

```
{
  IRLID irl_id;
  for (irl_id ⟸ 0; irl_id < irl_count; irl_id++) {
    int rhs_ix;
    const IRL irl ⟸ IRL_by_ID(irl_id);
    if (¬IRL_is_Right_Recursive(irl)) {
      continue;
    }
    for (rhs_ix ⟸ Length_of_IRL(irl) − 1; rhs_ix ≥ 0; rhs_ix−−) {
        /* LHS right dervies the last non-nulling symbol. There is at least one
           non-nulling symbol in each IRL. */
      const NSYID rh_nsyid ⟸ RHSID_of_IRL(irl, rhs_ix);
      if (¬NSY_is_Nulling(NSY_by_ID(rh_nsyid))) {
        matrix_bit_set(nsy_by_right_nsy_matrix, LHSID_of_IRL(irl), rh_nsyid);
        break;
      }
    }
  }
}
```

This code is used in section 507.

511. ⟨ Declare variables for the internal grammar memoizations 511 ⟩ ≡

```
const RULEID irl_count ⇐ IRL_Count_of_G(g);
const NSYID nsy_count ⇐ NSY_Count_of_G(g);
Bit_Matrix nsy_by_right_nsy_matrix;
Bit_Matrix prediction_nsy_by_irl_matrix;
```

This code is used in section 368.

512. Initialized based on the capacity of the XRL stack, rather than its length, as a convenient way to deal with issues of minimum sizes.

⟨ Initialize IRL stack 512 ⟩ ≡

```
MARPA_DSTACK_INIT(g→t_irl_stack, IRL,
    2 * MARPA_DSTACK_CAPACITY(g→t_xrl_stack));
```

This code is used in section 368.

513. Clones all the used symbols, creating nulling versions as required. Initialized based on the capacity of the XSY stack, rather than its length, as a convenient way to deal with issues of minimum sizes.

⟨ Initialize NSY stack 513 ⟩ ≡

```
{
  MARPA_DSTACK_INIT(g→t_nsy_stack, NSY,
      2 * MARPA_DSTACK_CAPACITY(g→t_xsy_stack));
}
```

This code is used in section 368.

514. ⟨ Calculate Rule by LHS lists 514 ⟩ ≡

```
{
  NSYID lhsid;
    /* This matrix is large and very temporary, so it does not go on the obstack */
  void *matrix_buffer ⇐ my_malloc(matrix_sizeof(nsy_count, irl_count));
  Bit_Matrix irl_by_lhs_matrix ⇐ matrix_buffer_create(matrix_buffer,
      nsy_count, irl_count);
  IRLID irl_id;
  for (irl_id ⇐ 0; irl_id < irl_count; irl_id++) {
    const IRL irl ⇐ IRL_by_ID(irl_id);
    const NSYID lhs_nsyid ⇐ LHSID_of_IRL(irl);
    matrix_bit_set(irl_by_lhs_matrix, lhs_nsyid, irl_id);
  }
    /* for every LHS row of the IRL-by-LHS matrix, add all its IRL's to the LHS CIL
        */
  for (lhsid ⇐ 0; lhsid < nsy_count; lhsid++) {
    IRLID irlid;
    int min, max, start;
    cil_buffer_clear(&g→t_cilar);
```

```
    for (start ⇐ 0; bv_scan(matrix_row(irl_by_lhs_matrix, lhsid), start, &min,
          &max); start ⇐ max + 2) {
      for (irlid ⇐ min; irlid ≤ max; irlid++) {
        cil_buffer_push(&g→t_cilar, irlid);
      }
    }
    LHS_CIL_of_NSYID(lhsid) ⇐ cil_buffer_add(&g→t_cilar);
  }
  my_free(matrix_buffer);
}
```

This code is used in section 368.

515. Predictions.

516. For the predicted states, I construct a symbol-by-rule matrix of predictions. First, I determine which symbols directly predict others. Then I compute the transitive closure. Finally, I convert this to a symbol-by-rule matrix. The symbol-by-rule matrix will be used in constructing the prediction states.

517. ⟨Construct prediction matrix 517⟩ ≡

```
{
  Bit_Matrix prediction_nsy_by_nsy_matrix ⇐
      matrix_obs_create(obs_precompute, nsy_count, nsy_count);
  ⟨Initialize the prediction_nsy_by_nsy_matrix 518⟩
  transitive_closure(prediction_nsy_by_nsy_matrix);
  ⟨Create the prediction matrix from the symbol-by-symbol matrix 519⟩
}
```

This code is used in section 368.

518. ⟨Initialize the prediction_nsy_by_nsy_matrix 518⟩ ≡

```
{
  IRLID irl_id;
  NSYID nsyid;
  for (nsyid ⇐ 0; nsyid < nsy_count; nsyid++) {
      /* If a symbol appears on a LHS, it predicts itself. */
    NSY nsy ⇐ NSY_by_ID(nsyid);
    if (¬NSY_is_LHS(nsy)) continue;
    matrix_bit_set(prediction_nsy_by_nsy_matrix, nsyid, nsyid);
  }
  for (irl_id ⇐ 0; irl_id < irl_count; irl_id++) {
    NSYID from_nsyid, to_nsyid;
    const IRL irl ⇐ IRL_by_ID(irl_id);    /* Get the initial item for the rule */
    const AHM item ⇐ First_AHM_of_IRL(irl);
    to_nsyid ⇐ Postdot_NSYID_of_AHM(item);
      /* There is no symbol-to-symbol transition for a completion item */
```

```
      if (to_nsyid < 0) continue;      /* Set a bit in the matrix */
      from_nsyid ⟸ LHS_NSYID_of_AHM(item);
      matrix_bit_set(prediction_nsy_by_nsy_matrix, from_nsyid, to_nsyid);
    }
  }
```

This code is used in section 517.

519. At this point I have a full matrix showing which symbol implies a prediction of which others. To save repeated processing when creating the prediction Earley items, I now convert it into a matrix from symbols to the rules they predict. Specifically, if symbol S1 predicts symbol S2, then symbol S1 predicts every rule with S2 on its LHS.

```
⟨ Create the prediction matrix from the symbol-by-symbol matrix 519 ⟩ ≡
  { ⟨ Populate the prediction matrix 520 ⟩
  }
```

This code is used in section 517.

520. ⟨ Populate the prediction matrix 520 ⟩ ≡

```
  {
    NSYID from_nsyid;
    prediction_nsy_by_irl_matrix ⟸ matrix_obs_create(obs_precompute,
        nsy_count, irl_count);
    for (from_nsyid ⟸ 0; from_nsyid < nsy_count; from_nsyid++) {
      /* for every row of the symbol-by-symbol matrix */
      int min, max, start;
      for (start ⟸ 0; bv_scan(matrix_row(prediction_nsy_by_nsy_matrix,
            from_nsyid), start, &min, &max); start ⟸ max + 2) {
        NSYID to_nsyid;
      /* for every predicted symbol */
        for (to_nsyid ⟸ min; to_nsyid ≤ max; to_nsyid++) {
          int cil_ix;
          const CIL lhs_cil ⟸ LHS_CIL_of_NSYID(to_nsyid);
          const int cil_count ⟸ Count_of_CIL(lhs_cil);
          for (cil_ix ⟸ 0; cil_ix < cil_count; cil_ix++) {
            const IRLID irlid ⟸ Item_of_CIL(lhs_cil, cil_ix);
            matrix_bit_set(prediction_nsy_by_irl_matrix, from_nsyid, irlid);
          }
        }
      }
    }
  }
```

This code is used in section 519.

521. Populating the predicted IRL CIL's in the AHM's.

522. ⟨ Populate the predicted IRL CIL's in the AHM's 522 ⟩ ≡

```
{
  AHMID ahm_id;
  const int ahm_count ⇐ AHM_Count_of_G(g);
  for (ahm_id ⇐ 0; ahm_id < ahm_count; ahm_id++) {
    const AHM ahm ⇐ AHM_by_ID(ahm_id);
    const NSYID postdot_nsyid ⇐ Postdot_NSYID_of_AHM(ahm);
    if (postdot_nsyid < 0) {
      Predicted_IRL_CIL_of_AHM(ahm) ⇐ cil_empty(&g→t_cilar);
      LHS_CIL_of_AHM(ahm) ⇐ cil_empty(&g→t_cilar);
    }
    else {
      Predicted_IRL_CIL_of_AHM(ahm) ⇐ cil_bv_add(&g→t_cilar,
          matrix_row(prediction_nsy_by_irl_matrix, postdot_nsyid));
      LHS_CIL_of_AHM(ahm) ⇐ LHS_CIL_of_NSYID(postdot_nsyid);
    }
  }
}
```

This code is used in section 368.

523. Populating the terminal boolean vector.

⟨ Populate the terminal boolean vector 523 ⟩ ≡

```
{
  int xsy_id;
  g->t_bv_nsyid_is_terminal = bv_obs_create(g->t_obs, nsy_count);
  for (xsy_id = 0; xsy_id < post_census_xsy_count; xsy_id++) {
    if (XSYID_is_Terminal(xsy_id)) {
        /* A terminal might have no corresponding NSY. Currently that can happen
           if it is not accessible */
      const NSY nsy = NSY_of_XSY(XSY_by_ID(xsy_id));
      if (nsy) {
        bv_bit_set(g->t_bv_nsyid_is_terminal, ID_of_NSY(nsy));
      }
    }
  }
}
```

This code is used in section 368.

524. Populating the event boolean vectors.

⟨ Populate the event boolean vectors 524 ⟩ ≡

```
{
  int xsyid;
  g->t_lbv_xsyid_is_completion_event = bv_obs_create(g->t_obs,
      post_census_xsy_count);
  g->t_lbv_xsyid_completion_event_starts_active = bv_obs_create(g->t_obs,
      post_census_xsy_count);
  g->t_lbv_xsyid_is_nulled_event = bv_obs_create(g->t_obs,
      post_census_xsy_count);
  g->t_lbv_xsyid_nulled_event_starts_active = bv_obs_create(g->t_obs,
      post_census_xsy_count);
  g->t_lbv_xsyid_is_prediction_event = bv_obs_create(g->t_obs,
      post_census_xsy_count);
  g->t_lbv_xsyid_prediction_event_starts_active = bv_obs_create(g->t_obs,
      post_census_xsy_count);
  for (xsyid = 0; xsyid < post_census_xsy_count; xsyid++) {
    if (XSYID_is_Completion_Event(xsyid)) {
      lbv_bit_set(g->t_lbv_xsyid_is_completion_event, xsyid);
    }
    if (XSYID_Completion_Event_Starts_Active(xsyid)) {
      lbv_bit_set(g->t_lbv_xsyid_completion_event_starts_active, xsyid);
    }
    if (XSYID_is_Nulled_Event(xsyid)) {
      lbv_bit_set(g->t_lbv_xsyid_is_nulled_event, xsyid);
    }
    if (XSYID_Nulled_Event_Starts_Active(xsyid)) {
      lbv_bit_set(g->t_lbv_xsyid_nulled_event_starts_active, xsyid);
    }
    if (XSYID_is_Prediction_Event(xsyid)) {
      lbv_bit_set(g->t_lbv_xsyid_is_prediction_event, xsyid);
    }
    if (XSYID_Prediction_Event_Starts_Active(xsyid)) {
      lbv_bit_set(g->t_lbv_xsyid_prediction_event_starts_active, xsyid);
    }
  }
}
```

This code is used in section 368.

525. ⟨ Populate the prediction and nulled symbol CILs 525 ⟩ ≡

```
{
  AHMID ahm_id;
  const int ahm_count_of_g = AHM_Count_of_G(g);
  const LBV bv_completion_xsyid = bv_create(post_census_xsy_count);
  const LBV bv_prediction_xsyid = bv_create(post_census_xsy_count);
```

```
const LBV bv_nulled_xsyid ⟸ bv_create(post_census_xsy_count);
const CILAR cilar ⟸ &g→t_cilar;
for (ahm_id ⟸ 0; ahm_id < ahm_count_of_g; ahm_id++) {
  const AHM ahm ⟸ AHM_by_ID(ahm_id);
  const NSYID postdot_nsyid ⟸ Postdot_NSYID_of_AHM(ahm);
  const IRL irl ⟸ IRL_of_AHM(ahm);
  bv_clear(bv_completion_xsyid);
  bv_clear(bv_prediction_xsyid);
  bv_clear(bv_nulled_xsyid);
  {
    int rhs_ix;
    int raw_position ⟸ Position_of_AHM(ahm);
    if (raw_position < 0) {    /* Completion */
      raw_position ⟸ Length_of_IRL(irl);
      if (¬IRL_has_Virtual_LHS(irl)) {    /* Completion */
        const NSY lhs ⟸ LHS_of_IRL(irl);
        const XSY xsy ⟸ Source_XSY_of_NSY(lhs);
        if (XSY_is_Completion_Event(xsy)) {
          const XSYID xsyid ⟸ ID_of_XSY(xsy);
          bv_bit_set(bv_completion_xsyid, xsyid);
        }
      }
    }
    if (postdot_nsyid ≥ 0) {
      const XSY xsy ⟸ Source_XSY_of_NSYID(postdot_nsyid);
      const XSYID xsyid ⟸ ID_of_XSY(xsy);
      bv_bit_set(bv_prediction_xsyid, xsyid);
    }
    for (rhs_ix ⟸ raw_position − Null_Count_of_AHM(ahm);
          rhs_ix < raw_position; rhs_ix++) {
      int cil_ix;
      const NSYID rhs_nsyid ⟸ RHSID_of_IRL(irl, rhs_ix);
      const XSY xsy ⟸ Source_XSY_of_NSYID(rhs_nsyid);
      const CIL nulled_xsyids ⟸ Nulled_XSYIDs_of_XSY(xsy);
      const int cil_count ⟸ Count_of_CIL(nulled_xsyids);
      for (cil_ix ⟸ 0; cil_ix < cil_count; cil_ix++) {
        const XSYID nulled_xsyid ⟸ Item_of_CIL(nulled_xsyids, cil_ix);
        bv_bit_set(bv_nulled_xsyid, nulled_xsyid);
      }
    }
  }
  Completion_XSYIDs_of_AHM(ahm) ⟸ cil_bv_add(cilar, bv_completion_xsyid);
  Nulled_XSYIDs_of_AHM(ahm) ⟸ cil_bv_add(cilar, bv_nulled_xsyid);
```

```
        Prediction_XSYIDs_of_AHM(ahm) ⇐ cil_bv_add(cilar, bv_prediction_xsyid);
      }
      bv_free(bv_completion_xsyid);
      bv_free(bv_prediction_xsyid);
      bv_free(bv_nulled_xsyid);
    }
```

This code is used in section 368.

526. ⟨ Mark the event AHMs 526 ⟩ ≡

```
  {
    AHMID ahm_id;
    for (ahm_id ⇐ 0; ahm_id < AHM_Count_of_G(g); ahm_id++) {
      const CILAR cilar ⇐ &g→t_cilar;
      const AHM ahm ⇐ AHM_by_ID(ahm_id);
      const int ahm_is_event ⇐ Count_of_CIL(Completion_XSYIDs_of_AHM(ahm)) ∨
          Count_of_CIL(Nulled_XSYIDs_of_AHM(ahm)) ∨
          Count_of_CIL(Prediction_XSYIDs_of_AHM(ahm));
      Event_AHMIDs_of_AHM(ahm) ⇐ ahm_is_event ? cil_singleton(cilar,
          ahm_id) : cil_empty(cilar);
    }
  }
```

This code is used in section 368.

527. ⟨ Calculate AHM Event Group Sizes 527 ⟩ ≡

```
  {
    const int ahm_count_of_g ⇐ AHM_Count_of_G(g);
    AHMID outer_ahm_id;
    for (outer_ahm_id ⇐ 0; outer_ahm_id < ahm_count_of_g; outer_ahm_id++) {
      AHMID inner_ahm_id;
      const AHM outer_ahm ⇐ AHM_by_ID(outer_ahm_id);
        /* There is no test that outer_ahm is an event AHM. An AHM, even if it is not
          itself an event AHM, may be in a non-empty AHM event group. */
      NSYID outer_nsyid;
      if (¬AHM_is_Leo_Completion(outer_ahm)) {
        if (AHM_has_Event(outer_ahm)) {
          Event_Group_Size_of_AHM(outer_ahm) ⇐ 1;
        }
        continue;     /* This AHM is not a Leo completion, so we are done. */
      }
      outer_nsyid ⇐ LHSID_of_AHM(outer_ahm);
      for (inner_ahm_id ⇐ 0; inner_ahm_id < ahm_count_of_g; inner_ahm_id++) {
        NSYID inner_nsyid;
        const AHM inner_ahm ⇐ AHM_by_ID(inner_ahm_id);
```

```
        if (¬AHM_has_Event(inner_ahm)) continue;
            /* Not in the group, because it is not an event AHM. */
        if (¬AHM_is_Leo_Completion(inner_ahm)) continue;
            /* This AHM is not a Leo completion, so we are done. */
        inner_nsyid ⟸ LHSID_of_AHM(inner_ahm);
        if (matrix_bit_test(nsy_by_right_nsy_matrix, outer_nsyid, inner_nsyid)) {
              /* inner_ahm ≡ outer_ahm is not treated as special case */
          Event_Group_Size_of_AHM(outer_ahm)++;
        }
      }
    }
  }
```

This code is used in section 368.

528. Zero-width assertion (ZWA) code.

⟨ Private incomplete structures 107 ⟩ +≡
```
struct s_g_zwa;
struct s_r_zwa;
```

529.

```
#define ZWAID_is_Malformed(zwaid) ((zwaid) < 0)
#define ZWAID_of_G_Exists(zwaid) ((zwaid) < ZWA_Count_of_G(g))
```

⟨ Private typedefs 49 ⟩ +≡
```
typedef Marpa_Assertion_ID ZWAID;
typedef struct s_g_zwa *GZWA;
typedef struct s_r_zwa *ZWA;
```

530.
```
#define ZWA_Count_of_G(g) (MARPA_DSTACK_LENGTH((g)→t_gzwa_stack))
#define GZWA_by_ID(id) (*MARPA_DSTACK_INDEX((g)→t_gzwa_stack, GZWA, (id)))
```

⟨ Widely aligned grammar elements 59 ⟩ +≡
```
MARPA_DSTACK_DECLARE(t_gzwa_stack);
```

531. ⟨ Initialize grammar elements 54 ⟩ +≡
```
MARPA_DSTACK_INIT2(g→t_gzwa_stack, GZWA);
```

532. ⟨ Destroy grammar elements 61 ⟩ +≡
```
MARPA_DSTACK_DESTROY(g→t_gzwa_stack);
```

533. ⟨ Public typedefs 91 ⟩ +≡
```
typedef int Marpa_Assertion_ID;
```

534.

```
#define ID_of_GZWA(zwa) ((zwa)→t_id)
#define Default_Value_of_GZWA(zwa) ((zwa)→t_default_value)
```

⟨ Private structures 48 ⟩ +≡
```
struct s_g_zwa {
  ZWAID t_id;
  BITFIELD t_default_value:1;
};
typedef struct s_g_zwa GZWA_Object;
```

535. ⟨ Private incomplete structures 107 ⟩ +≡
```
struct s_zwp;
```

536. ⟨ Private typedefs 49 ⟩ +≡
```
typedef struct s_zwp *ZWP;
typedef const struct s_zwp *ZWP_Const;
```

537.

```
#define XRLID_of_ZWP(zwp) ((zwp)→t_xrl_id)
#define Dot_of_ZWP(zwp) ((zwp)→t_dot)
#define ZWAID_of_ZWP(zwp) ((zwp)→t_zwaid)
```

⟨ Private structures 48 ⟩ +≡

```
struct s_zwp {
  XRLID t_xrl_id;
  int t_dot;
  ZWAID t_zwaid;
};
typedef struct s_zwp ZWP_Object;
```

538. ⟨ Widely aligned grammar elements 59 ⟩ +≡

```
MARPA_AVL_TREE t_zwp_tree;
```

539. ⟨ Initialize grammar elements 54 ⟩ +≡

```
(g)→t_zwp_tree ⟸ _marpa_avl_create(zwp_cmp, Λ);
```

540. ⟨ Destroy grammar elements 61 ⟩ +≡

```
{
  _marpa_avl_destroy((g)→t_zwp_tree);
  (g)→t_zwp_tree ⟸ Λ;
}
```

541. ⟨ Destroy grammar elements 61 ⟩ +≡

⟨ Clear rule duplication tree 122 ⟩

542. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE_NOT_INLINE int zwp_cmp(const void *ap, const void *bp, void
      *param UNUSED)
{
  const ZWP_Const zwp_a ⟸ ap;
  const ZWP_Const zwp_b ⟸ bp;
  int subkey ⟸ XRLID_of_ZWP(zwp_a) − XRLID_of_ZWP(zwp_b);
  if (subkey) return subkey;
  subkey ⟸ Dot_of_ZWP(zwp_a) − Dot_of_ZWP(zwp_b);
  if (subkey) return subkey;
  return ZWAID_of_ZWP(zwp_a) − ZWAID_of_ZWP(zwp_b);
}
```

543. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Assertion_ID marpa_g_zwa_new(Marpa_Grammar g, int default_value)
{
  ⟨ Return −2 on failure 1229 ⟩
  ZWAID zwa_id;
```

```
    GZWA gzwa;
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if precomputed 1230 ⟩
    if (_MARPA_UNLIKELY(default_value < 0 ∨ default_value > 1)) {
      MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
      return failure_indicator;
    }
    gzwa ⟸ marpa_obs_new(g→t_obs, GZWA_Object, 1);
    zwa_id ⟸ MARPA_DSTACK_LENGTH((g)→t_gzwa_stack);
    *MARPA_DSTACK_PUSH((g)→t_gzwa_stack, GZWA) ⟸ gzwa;
    gzwa→t_id ⟸ zwa_id;
    gzwa→t_default_value ⟸ default_value ? 1 : 0;
    return zwa_id;
  }
```

544. ⟨ Function definitions 41 ⟩ +≡

```
  Marpa_Assertion_ID marpa_g_highest_zwa_id(Marpa_Grammar g)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    return ZWA_Count_of_G(g) − 1;
  }
```

545. An attempt to insert a duplicate is treated as a soft failure, and -1 is returned. On success, returns a non-negative number.

⟨ Function definitions 41 ⟩ +≡

```
  int marpa_g_zwa_place(Marpa_Grammar g, Marpa_Assertion_ID zwaid, Marpa_Rule_ID
          xrl_id, int rhs_ix)
  {
    ⟨ Return −2 on failure 1229 ⟩
    void *avl_insert_result;
    ZWP zwp;
    XRL xrl;
    int xrl_length;
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if precomputed 1230 ⟩
    ⟨ Fail if xrl_id is malformed 1241 ⟩
    ⟨ Soft fail if xrl_id does not exist 1239 ⟩
    ⟨ Fail if zwaid is malformed 1243 ⟩
    ⟨ Fail if zwaid does not exist 1242 ⟩
    xrl ⟸ XRL_by_ID(xrl_id);
    if (rhs_ix < −1) {
      MARPA_ERROR(MARPA_ERR_RHS_IX_NEGATIVE);
      return failure_indicator;
```

```
    }
    xrl_length ⟸ Length_of_XRL(xrl);
    if (xrl_length ≤ rhs_ix) {
      MARPA_ERROR(MARPA_ERR_RHS_IX_OOB);
      return failure_indicator;
    }
    if (rhs_ix ≡ −1) {
      rhs_ix ⟸ XRL_is_Sequence(xrl) ? 1 : xrl_length;
    }
    zwp ⟸ marpa_obs_new(g→t_obs, ZWP_Object, 1);
    XRLID_of_ZWP(zwp) ⟸ xrl_id;
    Dot_of_ZWP(zwp) ⟸ rhs_ix;
    ZWAID_of_ZWP(zwp) ⟸ zwaid;
    avl_insert_result ⟸ _marpa_avl_insert(g→t_zwp_tree, zwp);
    return avl_insert_result ? −1 : 0;
  }
```

546. The direct ZWA's are the zero-width assertions triggered directly by the AHM. ZWA's triggered via predictions are called "indirect".

⟨ Find the direct ZWA's for each AHM 546 ⟩ ≡

```
  {
    AHMID ahm_id;
    const int ahm_count_of_g ⟸ AHM_Count_of_G(g);
    for (ahm_id ⟸ 0; ahm_id < ahm_count_of_g; ahm_id++) {
      ZWP_Object sought_zwp_object;
      ZWP sought_zwp ⟸ &sought_zwp_object;
      ZWP found_zwp;
      MARPA_AVL_TRAV traverser;
      const AHM ahm ⟸ AHM_by_ID(ahm_id);
      const XRL ahm_xrl ⟸ XRL_of_AHM(ahm);
      cil_buffer_clear(&g→t_cilar);
      if (ahm_xrl) {
        const int xrl_dot_end ⟸ Raw_XRL_Position_of_AHM(ahm);
        const int xrl_dot_start ⟸ xrl_dot_end − Null_Count_of_AHM(ahm);
          /* We assume the null count is zero for a sequence rule */
        const XRLID sought_xrlid ⟸ ID_of_XRL(ahm_xrl);
        XRLID_of_ZWP(sought_zwp) ⟸ sought_xrlid;
        Dot_of_ZWP(sought_zwp) ⟸ xrl_dot_start;
        ZWAID_of_ZWP(sought_zwp) ⟸ 0;
        traverser ⟸ _marpa_avl_t_init((g)→t_zwp_tree);
        found_zwp ⟸ _marpa_avl_t_at_or_after(traverser, sought_zwp);
      /* While we are in the dot range of the sought XRL */
```

```
        while (found_zwp ∧ XRLID_of_ZWP(found_zwp) ≡
            sought_xrlid ∧ Dot_of_ZWP(found_zwp) ≤ xrl_dot_end)
          {
          cil_buffer_push(&g→t_cilar, ZWAID_of_ZWP(found_zwp));
          found_zwp ⟸ _marpa_avl_t_next(traverser);
        }
      }
      ZWA_CIL_of_AHM(ahm) ⟸ cil_buffer_add(&g→t_cilar);
    }
  }
```

This code is used in section 368.

547. The indirect ZWA's are the zero-width assertions triggered via predictions. They do **not** include the ZWA's triggered directly by the AHM itself.

⟨ Find the indirect ZWA's for each AHM's 547 ⟩ ≡

```
  {
    AHMID ahm_id;
    const int ahm_count_of_g ⟸ AHM_Count_of_G(g);
    for (ahm_id ⟸ 0; ahm_id < ahm_count_of_g; ahm_id++) {
      const AHM ahm_to_populate ⟸ AHM_by_ID(ahm_id);

      /* The "predicts ZWA" bit was initialized to assume no prediction */
      const CIL prediction_cil ⟸ Predicted_IRL_CIL_of_AHM(ahm_to_populate);
      const int prediction_count ⟸ Count_of_CIL(prediction_cil);
      int cil_ix;
      for (cil_ix ⟸ 0; cil_ix < prediction_count; cil_ix++) {
        const IRLID prediction_irlid ⟸ Item_of_CIL(prediction_cil, cil_ix);
        const AHM prediction_ahm_of_irl ⟸
            First_AHM_of_IRLID(prediction_irlid);
        const CIL zwaids_of_prediction ⟸ ZWA_CIL_of_AHM(prediction_ahm_of_irl);
        if (Count_of_CIL(zwaids_of_prediction) > 0) {
          AHM_predicts_ZWA(ahm_to_populate) ⟸ 1;
          break;
        }
      }
    }
  }
```

This code is used in section 368.

548. Recognizer (R, RECCE) code.

⟨ Public incomplete structures 47 ⟩ +≡
```
struct marpa_r;
typedef struct marpa_r *Marpa_Recognizer;
typedef Marpa_Recognizer Marpa_Recce;
```

549. ⟨ Private typedefs 49 ⟩ +≡
```
typedef struct marpa_r *RECCE;
```

550. ⟨ Recognizer structure 550 ⟩ ≡
```
struct marpa_r {
  ⟨ Widely aligned recognizer elements 558 ⟩
  ⟨ Int aligned recognizer elements 553 ⟩
  ⟨ Bit aligned recognizer elements 562 ⟩
};
```
This code is used in section 1383.

551. The grammar must not be deallocated for the life of the recognizer. In the event of an error creating the recognizer, Λ is returned.

⟨ Function definitions 41 ⟩ +≡
```
Marpa_Recognizer marpa_r_new(Marpa_Grammar g)
{
  RECCE r;
  int nsy_count;
  int irl_count;
  ⟨ Return Λ on failure 1228 ⟩
  ⟨ Fail if not precomputed 1231 ⟩
  nsy_count ⟸ NSY_Count_of_G(g);
  irl_count ⟸ IRL_Count_of_G(g);
  r ⟸ my_malloc(sizeof (struct marpa_r));
  ⟨ Initialize recognizer obstack 616 ⟩
  ⟨ Initialize recognizer elements 554 ⟩
  ⟨ Initialize dot PSAR 1210 ⟩
  ⟨ Initialize recognizer event variables 579 ⟩
  return r;
}
```

552. Reference counting and destructors.

553. ⟨ Int aligned recognizer elements 553 ⟩ ≡
```
int t_ref_count;
```
See also sections 569, 573, 578, 613, and 634.
This code is used in section 550.

554. ⟨ Initialize recognizer elements 554 ⟩ ≡

```
r→t_ref_count ⟸ 1;
```

See also sections 559, 564, 566, 570, 574, 581, 585, 603, 607, 610, 614, 620, 635, 701, 726, 730, 734, 825, 859, 1261, 1268, 1282, and 1290.

This code is used in section 551.

555. Decrement the recognizer reference count.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void recce_unref(RECCE r)
{
  MARPA_ASSERT(r→t_ref_count > 0)r→t_ref_count−−;
  if (r→t_ref_count ≤ 0) {
    recce_free(r);
  }
}
void marpa_r_unref(Marpa_Recognizer r)
{
  recce_unref(r);
}
```

556. Increment the recognizer reference count.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE RECCE recce_ref(RECCE r)
{
  MARPA_ASSERT(r→t_ref_count > 0)r→t_ref_count++;
  return r;
}
Marpa_Recognizer marpa_r_ref(Marpa_Recognizer r)
{
  return recce_ref(r);
}
```

557. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void recce_free(struct marpa_r *r)
{
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Destroy recognizer elements 561 ⟩
  ⟨ Destroy recognizer obstack 617 ⟩
  my_free(r);
}
```

558. Base objects. Initialized in `marpa_r_new`.

#define `G_of_R`(r) ((r)→`t_grammar`)

⟨ Widely aligned recognizer elements 558 ⟩ ≡
  GRAMMAR `t_grammar`;

See also sections 565, 577, 580, 584, 606, 615, 619, 700, 717, 725, 729, 733, 770, 789, 824, 858, 1209, 1260, 1267, 1281, and 1288.

This code is used in section 550.

559. ⟨ Initialize recognizer elements 554 ⟩ +≡
  {
    `G_of_R`(r) ⟸ g;
    `grammar_ref`(g);
  }

560. ⟨ Unpack recognizer objects 560 ⟩ ≡
  const GRAMMAR g ⟸ `G_of_R`(r);

This code is used in sections 557, 567, 582, 583, 586, 588, 590, 592, 604, 605, 612, 639, 640, 641, 642, 653, 710, 719, 737, 773, 802, 821, 822, 832, 833, 836, 837, 1262, 1263, 1264, 1266, 1271, 1273, 1276, 1278, 1279, 1280, 1283, 1285, 1286, 1287, 1292, 1295, 1297, 1300, 1302, 1305, 1308, 1309, 1311, 1313, and 1355.

561. ⟨ Destroy recognizer elements 561 ⟩ ≡
  `grammar_unref`(g);

See also sections 608, 702, 728, 732, 735, 827, 860, and 1211.

This code is used in section 557.

562. Input phase. The recognizer always is in a one of the following phases:

#define `R_BEFORE_INPUT` #1
#define `R_DURING_INPUT` #2
#define `R_AFTER_INPUT` #3

⟨ Bit aligned recognizer elements 562 ⟩ ≡
  BITFIELD `t_input_phase`:2;

See also sections 602, 609, and 1289.

This code is used in section 550.

563. *#define* `Input_Phase_of_R`(r) ((r)→`t_input_phase`)

564. ⟨ Initialize recognizer elements 554 ⟩ +≡
  `Input_Phase_of_R`(r) ⟸ `R_BEFORE_INPUT`;

565. Earley set container.

#define `First_YS_of_R`(r) ((r)→`t_first_earley_set`)

⟨ Widely aligned recognizer elements 558 ⟩ +≡
  YS `t_first_earley_set`;
  YS `t_latest_earley_set`;
  JEARLEME `t_current_earleme`;

566. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
r→t_first_earley_set ⇐ Λ;
r→t_latest_earley_set ⇐ Λ;
r→t_current_earleme ⇐ −1;
```

567. Current earleme.

```
#define Latest_YS_of_R(r) ((r)→t_latest_earley_set)
#define Current_Earleme_of_R(r) ((r)→t_current_earleme)
```

⟨ Function definitions 41 ⟩ +≡

```
Marpa_Earleme marpa_r_current_earleme(Marpa_Recognizer r)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  if (_MARPA_UNLIKELY(Input_Phase_of_R(r) ≡ R_BEFORE_INPUT)) {
    MARPA_ERROR(MARPA_ERR_RECCE_NOT_STARTED);
    return −1;
  }
  return Current_Earleme_of_R(r);
}
```

568. The "Earley set at the current earleme" is always the latest YS, if it is defined. There may not be a YS at the current earleme.

```
#define YS_at_Current_Earleme_of_R(r) ys_at_current_earleme(r)
```

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE YS ys_at_current_earleme(RECCE r)
{
  const YS latest ⇐ Latest_YS_of_R(r);
  if (Earleme_of_YS(latest) ≡ Current_Earleme_of_R(r)) return latest;
  return Λ;
}
```

569. Earley set warning threshold.

```
#define DEFAULT_YIM_WARNING_THRESHOLD (100)
```

⟨ Int aligned recognizer elements 553 ⟩ +≡

```
int t_earley_item_warning_threshold;
```

570. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
r→t_earley_item_warning_threshold ⇐ MAX(DEFAULT_YIM_WARNING_THRESHOLD,
    AHM_Count_of_G(g) * 3);
```

571. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_earley_item_warning_threshold(Marpa_Recognizer r)
{
  return r→t_earley_item_warning_threshold;
}
```

572. Returns true on success, false on failure.

⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_earley_item_warning_threshold_set(Marpa_Recognizer r, int
        threshold)
{
  const int new_threshold ⟸ threshold ≤ 0 ? YIM_FATAL_THRESHOLD : threshold;
  r→t_earley_item_warning_threshold ⟸ new_threshold;
  return new_threshold;
}
```

573. Furthest earleme. The "furthest" or highest-numbered earleme. This is the "furthest out" earleme that the recognizer make reference to. Marpa allows variable length tokens, so it needs to track how far out tokens might be found. No token ends after the furthest earleme.

```
#define Furthest_Earleme_of_R(r) ((r)→t_furthest_earleme)
```

⟨ Int aligned recognizer elements 553 ⟩ +≡

```
JEARLEME t_furthest_earleme;
```

574. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
r→t_furthest_earleme ⟸ 0;
```

575. Always succeeds to allow *unsigned int* to be used for the value. This makes the interface for the furthest earleme non-orthogonal with that for the current earleme, but allows more values for the furthest earleme.

⟨ Function definitions 41 ⟩ +≡

```
unsigned int marpa_r_furthest_earleme(Marpa_Recognizer r)
{
  return (unsigned int) Furthest_Earleme_of_R(r);
}
```

576. Event variables. The count of unmasked XSY events. This count is used to protect recognizers that do not use events from their overhead. All these have to do is check the count against zero. There is no aggressive attempt to optimize on a more fine-grained basis – for recognizer which actually do use completion events, a few instructions per Earley item of overhead is considered reasonable.

577. ⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
Bit_Vector t_lbv_xsyid_completion_event_is_active;
Bit_Vector t_lbv_xsyid_nulled_event_is_active;
Bit_Vector t_lbv_xsyid_prediction_event_is_active;
```

578. ⟨ Int aligned recognizer elements 553 ⟩ +≡

```
int t_active_event_count;
```

579. ⟨ Initialize recognizer event variables 579 ⟩ ≡

```
{
  NSYID xsy_count ⇐ XSY_Count_of_G(g);
  r→t_lbv_xsyid_completion_event_is_active ⇐ lbv_clone(r→t_obs,
      g→t_lbv_xsyid_completion_event_starts_active, xsy_count);
  r→t_lbv_xsyid_nulled_event_is_active ⇐ lbv_clone(r→t_obs,
      g→t_lbv_xsyid_nulled_event_starts_active, xsy_count);
  r→t_lbv_xsyid_prediction_event_is_active ⇐ lbv_clone(r→t_obs,
      g→t_lbv_xsyid_prediction_event_starts_active, xsy_count);
  r→t_active_event_count ⇐ bv_count(g→t_lbv_xsyid_is_completion_event) +
      bv_count(g→t_lbv_xsyid_is_nulled_event) +
      bv_count(g→t_lbv_xsyid_is_prediction_event);
}
```

This code is used in section 551.

580. Expected symbol boolean vector. A boolean vector by symbol ID, with the bits set if the symbol is expected at the current earleme.

⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
Bit_Vector t_bv_nsyid_is_expected;
```

581. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
r→t_bv_nsyid_is_expected ⇐ bv_obs_create(r→t_obs, nsy_count);
```

582. Returns −2 if there was a failure. The buffer is expected to be large enough to hold the result. This will be the case if the length of the buffer is greater than or equal to the number of symbols in the grammar.

⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_terminals_expected(Marpa_Recognizer r, Marpa_Symbol_ID *buffer)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  NSYID xsy_count;
  Bit_Vector bv_terminals;
  int min, max, start;
  int next_buffer_ix ⇐ 0;
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if recognizer not started 1246 ⟩
  xsy_count ⇐ XSY_Count_of_G(g);
  bv_terminals ⇐ bv_create(xsy_count);
  for (start ⇐ 0; bv_scan(r→t_bv_nsyid_is_expected, start, &min, &max);
        start ⇐ max + 2) {
```

```
      NSYID nsyid;
      for (nsyid ⟸ min; nsyid ≤ max; nsyid++) {
        const XSY xsy ⟸ Source_XSY_of_NSYID(nsyid);
        bv_bit_set(bv_terminals, ID_of_XSY(xsy));
      }
    }
    for (start ⟸ 0; bv_scan(bv_terminals, start, &min, &max); start ⟸ max + 2)
      {
      XSYID xsyid;
      for (xsyid ⟸ min; xsyid ≤ max; xsyid++) {
        buffer[next_buffer_ix++] ⟸ xsyid;
      }
    }
    bv_free(bv_terminals);
    return next_buffer_ix;
  }
```

583. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_r_terminal_is_expected(Marpa_Recognizer r, Marpa_Symbol_ID xsy_id)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack recognizer objects 560 ⟩
    XSY xsy;
    NSY nsy;
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if recognizer not started 1246 ⟩
    ⟨ Fail if xsy_id is malformed 1232 ⟩
    ⟨ Fail if xsy_id does not exist 1234 ⟩
    xsy ⟸ XSY_by_ID(xsy_id);
    if (_MARPA_UNLIKELY(¬XSY_is_Terminal(xsy))) {
      return 0;
    }
    nsy ⟸ NSY_of_XSY(xsy);
    if (_MARPA_UNLIKELY(¬nsy)) return 0;    /* It may be an unused terminal */
    return bv_bit_test(r→t_bv_nsyid_is_expected, ID_of_NSY(nsy));
  }
```

584. Expected symbol is event?. A boolean vector by symbol ID, with the bits set if, when that symbol is an expected symbol, an event should be created. Here "expected" means "expected as a terminal". All expected symbols are predicted symbols, but the reverse is not true – predicted non-terminals are not "expected" symbols.

⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
  LBV t_nsy_expected_is_event;
```

585. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
r→t_nsy_expected_is_event ⟸ lbv_obs_new0(r→t_obs, nsy_count);
```

586. Returns −2 if there was a failure. Does not check if `xsy_id` is a terminal, because this is not decided until precomputation, which may not have been performed yet.

⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_expected_symbol_event_set(Marpa_Recognizer r, Marpa_Symbol_ID
        xsy_id, int value)
{
  XSY xsy;
  NSY nsy;
  NSYID nsyid;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  if (_MARPA_UNLIKELY(value < 0 ∨ value > 1)) {
    MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
    return failure_indicator;
  }
  xsy ⟸ XSY_by_ID(xsy_id);
  if (_MARPA_UNLIKELY(XSY_is_Nulling(xsy))) {
    MARPA_ERROR(MARPA_ERR_SYMBOL_IS_NULLING);
    return −2;
  }
  nsy ⟸ NSY_of_XSY(xsy);
  if (_MARPA_UNLIKELY(¬nsy)) {
    MARPA_ERROR(MARPA_ERR_SYMBOL_IS_UNUSED);
    return −2;
  }
  nsyid ⟸ ID_of_NSY(nsy);
  if (value) {
    lbv_bit_set(r→t_nsy_expected_is_event, nsyid);
  }
  else {
    lbv_bit_clear(r→t_nsy_expected_is_event, nsyid);
  }
  return value;
}
```

587. Deactivate symbol completed events.

588. Allows a recognizer to deactivate and reactivate symbol completed events. A `boolean` value of 1 indicates reactivate, a boolean value of 0 indicates deactivate. To be reactivated, the symbol must have been set up for completion events in the grammar. Success occurs non-trivially if the bit can be set to the new value. Success occurs trivially if it was already set as specified. Any other result is a failure. On success, returns the new value. Returns −2 if there was a failure.

⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_completion_symbol_activate(Marpa_Recognizer r, Marpa_Symbol_ID
        xsy_id, int reactivate)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  switch (reactivate) {
  case 0:
    if (lbv_bit_test(r→t_lbv_xsyid_completion_event_is_active, xsy_id)) {
      lbv_bit_clear(r→t_lbv_xsyid_completion_event_is_active, xsy_id);
      r→t_active_event_count−−;
    }
    return 0;
  case 1:
    if (¬lbv_bit_test(g→t_lbv_xsyid_is_completion_event, xsy_id)) {
        /* An attempt to activate a completion event on a symbol which was not set
          up for them. */
      MARPA_ERROR(MARPA_ERR_SYMBOL_IS_NOT_COMPLETION_EVENT);
    }
    if (¬lbv_bit_test(r→t_lbv_xsyid_completion_event_is_active, xsy_id)) {
      lbv_bit_set(r→t_lbv_xsyid_completion_event_is_active, xsy_id);
      r→t_active_event_count++;
    }
    return 1;
  }
  MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
  return failure_indicator;
}
```

589. Deactivate and reactivate symbol nulled events.

590. Allows a recognizer to deactivate and reactivate symbol nulled events. A `boolean` value of 1 indicates reactivate, a boolean value of 0 indicates deactivate. To be reactivated, the symbol must have been set up for nulled events in the grammar. Success occurs non-trivially if the bit can be set to the new value. Success occurs trivially if it was already set as specified. Any other result is a failure. On success, returns the new value. Returns −2 if there was a failure.

```
⟨ Function definitions 41 ⟩ +≡
  int marpa_r_nulled_symbol_activate(Marpa_Recognizer r, Marpa_Symbol_ID
          xsy_id, int reactivate)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if xsy_id is malformed 1232 ⟩
    ⟨ Soft fail if xsy_id does not exist 1233 ⟩
    switch (reactivate) {
    case 0:
      if (lbv_bit_test(r→t_lbv_xsyid_nulled_event_is_active, xsy_id)) {
        lbv_bit_clear(r→t_lbv_xsyid_nulled_event_is_active, xsy_id);
        r→t_active_event_count −−;
      }
      return 0;
    case 1:
      if (¬lbv_bit_test(g→t_lbv_xsyid_is_nulled_event, xsy_id)) {
          /* An attempt to activate a nulled event on a symbol which was not set up
             for them. */
        MARPA_ERROR(MARPA_ERR_SYMBOL_IS_NOT_NULLED_EVENT);
      }
      if (¬lbv_bit_test(r→t_lbv_xsyid_nulled_event_is_active, xsy_id)) {
        lbv_bit_set(r→t_lbv_xsyid_nulled_event_is_active, xsy_id);
        r→t_active_event_count ++;
      }
      return 1;
    }
    MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
    return failure_indicator;
  }
```

591. Deactivate and reactivate symbol prediction events.

592. Allows a recognizer to deactivate and reactivate symbol prediction events. A `boolean` value of 1 indicates reactivate, a boolean value of 0 indicates deactivate. To be reactivated, the symbol must have been set up for prediction events in the grammar. Success occurs non-trivially if the bit can be set to the new value. Success occurs trivially if it was already set as specified. Any other result is a failure. On success, returns the new value. Returns −2 if there was a failure.

```
⟨ Function definitions 41 ⟩ +≡
  int marpa_r_prediction_symbol_activate(Marpa_Recognizer r, Marpa_Symbol_ID
          xsy_id, int reactivate)
```

```
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  switch (reactivate) {
  case 0:
    if (lbv_bit_test(r→t_lbv_xsyid_prediction_event_is_active, xsy_id)) {
      lbv_bit_clear(r→t_lbv_xsyid_prediction_event_is_active, xsy_id);
      r→t_active_event_count−−;
    }
    return 0;
  case 1:
    if (¬lbv_bit_test(g→t_lbv_xsyid_is_prediction_event, xsy_id)) {
        /* An attempt to activate a prediction event on a symbol which was not set
           up for them. */
      MARPA_ERROR(MARPA_ERR_SYMBOL_IS_NOT_PREDICTION_EVENT);
    }
    if (¬lbv_bit_test(r→t_lbv_xsyid_prediction_event_is_active, xsy_id)) {
      lbv_bit_set(r→t_lbv_xsyid_prediction_event_is_active, xsy_id);
      r→t_active_event_count++;
    }
    return 1;
  }
  MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
  return failure_indicator;
}
```

593. Leo-related booleans.

594. Turning Leo logic off and on. A trace flag, set if we are using Leo items. This flag is set by default. It has two uses.

595. This flag is very useful for testing. Since Leo items do not affect function, only effiency, it is possible for the Leo logic to be broken or disabled without most tests noticiing. To make sure the Leo logic is intact, one of `libmarpa`'s tests runs one pass with Leo items off and another with Leo items on and compares them.

596. This flag also allows the Leo logic to be turned off in certain cases in which the Leo logic actually slows things down. The Leo logic could be turned off if the user knows there is no right recursion, although the actual gain, would typically be small or not measurable.

597. A real gain would occur in the case of highly ambiguous grammars, all or most of whose parses are actually evaluated. Since those Earley items eliminated by the Leo logic are actually recreated on an as-needed basis in the evaluation phase, in cases when most of the Earley items are needed for evaluation, the Leo logic would be eliminated Earley items only to have to add most of them later. In these cases, the Leo logic would impose a small overhead.

598. The author's current view is that it is best to start by assuming that the Leo logic should be left on. In the rare event, that it turns out that the Leo logic is counter-productive, this flag can be used to test if turning the Leo logic off is helpful.

599. It should be borne in mind that even when the Leo logic imposes a small cost in typical cases, it may act as a safeguard. The time complexity explosions prevented by Leo logic can easily mean the difference between an impractical computation and a practical one. In most applications, it is worth incurring an small overhead in the average case to prevent failures, even rare ones.

600. There are two booleans. One is a flag that can be set and unset externally, indicating the application's intention to use Leo logic. An internal boolean tracks whether the Leo logic is actually enabled at any given point.

601. The reason for having two booleans is that the Leo logic is only turned on once Earley set 0 is complete. While Earley set 0 is being processed the internal flag will always be unset, while the external flag may be set or unset, as the user decided. After Earley set 0 is complete, both booleans will have the same value.

602. **To Do**: Now that the null parse is special-cased, one boolean may suffice.

⟨ Bit aligned recognizer elements 562 ⟩ +≡
  BITFIELD `t_use_leo_flag`:1;
  BITFIELD `t_is_using_leo`:1;

603. ⟨ Initialize recognizer elements 554 ⟩ +≡
  r→`t_use_leo_flag` ⟸ 1;
  r→`t_is_using_leo` ⟸ 0;

604. Returns 1 if the "use Leo" flag is set, 0 if not, and −2 if there was an error.

⟨ Function definitions 41 ⟩ +≡
  int `_marpa_r_is_use_leo`(*Marpa_Recognizer r*)
  {
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    return *r*→`t_use_leo_flag`;
  }

605. ⟨ Function definitions 41 ⟩ +≡
```
int _marpa_r_is_use_leo_set(Marpa_Recognizer r, int value){ ⟨ Unpack recognizer
        objects 560 ⟩
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if recognizer started 1245 ⟩
    return r→t_use_leo_flag ⟸ value ? 1 : 0; }
```

606. Predicted IRL boolean vector and stack. A boolean vector by IRL ID, used while building the Earley sets. It is set if an IRL has already been predicted, unset otherwise.

⟨ Widely aligned recognizer elements 558 ⟩ +≡
```
Bit_Vector t_bv_irl_seen;
MARPA_DSTACK_DECLARE(t_irl_cil_stack);
```

607. ⟨ Initialize recognizer elements 554 ⟩ +≡
```
r→t_bv_irl_seen ⟸ bv_obs_create(r→t_obs, irl_count);
MARPA_DSTACK_INIT2(r→t_irl_cil_stack, CIL);
```

608. ⟨ Destroy recognizer elements 561 ⟩ +≡
```
MARPA_DSTACK_DESTROY(r→t_irl_cil_stack);
```

609. Is the parser exhausted?. A parser is "exhausted" if it cannot accept any more input. Both successful and failed parses can be "exhausted". In many grammars, the parse is always exhausted as soon as it succeeds. And even if the parse is exhausted at a point where there is no good parse, there may be good parses at earlemes prior to the earleme at which the parse became exhausted.

```
#define R_is_Exhausted(r) ((r)→t_is_exhausted)
```

⟨ Bit aligned recognizer elements 562 ⟩ +≡
```
BITFIELD t_is_exhausted:1;
```

610. ⟨ Initialize recognizer elements 554 ⟩ +≡
```
r→t_is_exhausted ⟸ 0;
```

611. ⟨ Set r exhausted 611 ⟩ ≡
```
{
  R_is_Exhausted(r) ⟸ 1;
  Input_Phase_of_R(r) ⟸ R_AFTER_INPUT;
  event_new(g, MARPA_EVENT_EXHAUSTED);
}
```
This code is used in sections 710, 737, 740, and 802.

612. Exhaustion is a boolean, not a phase. Once exhausted a parse stays exhausted, even though the phase may change.

⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_is_exhausted(Marpa_Recognizer r){ ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    return R_is_Exhausted(r); }
```

613. Is the parser consistent? A parser becomes inconsistent when YIM's or LIM's or ALT's are rejected. It can be made consistent again by calling `marpa_r_consistent()`.

```
#define First_Inconsistent_YS_of_R(r) ((r)→t_first_inconsistent_ys)
#define R_is_Consistent(r) ((r)→t_first_inconsistent_ys < 0)
```

⟨ Int aligned recognizer elements 553 ⟩ +≡

```
YSIDt_first_inconsistent_ys;
```

614. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
r→t_first_inconsistent_ys ⇐ −1;
```

615. The recognizer obstack. Create an obstack with the lifetime of the recognizer. This is a very efficient way of allocating memory which won't be resized and which will have the same lifetime as the recognizer.

⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
struct marpa_obstack *t_obs;
```

616. ⟨ Initialize recognizer obstack 616 ⟩ ≡

```
r→t_obs ⇐ marpa_obs_init;
```

This code is used in section 551.

617. ⟨ Destroy recognizer obstack 617 ⟩ ≡

```
marpa_obs_free(r→t_obs);
```

This code is used in section 557.

618. The ZWA Array.

```
#define ID_of_ZWA(zwa) ((zwa)→t_id)
#define Memo_YSID_of_ZWA(zwa) ((zwa)→t_memoized_ysid)
#define Memo_Value_of_ZWA(zwa) ((zwa)→t_memoized_value)
#define Default_Value_of_ZWA(zwa) ((zwa)→t_default_value)
```

⟨ Private structures 48 ⟩ +≡

```
struct s_r_zwa {
    ZWAID t_id;
    YSIDt_memoized_ysid;
    BITFIELD t_default_value:1;
    BITFIELD t_memoized_value:1;
};
typedef struct s_r_zwa ZWA_Object;
```

619. The grammar and recce ZWA counts are always the same.

#define ZWA_Count_of_R(r) (ZWA_Count_of_G(G_of_R(r)))
#define RZWA_by_ID(id) (&(r)→t_zwas[(zwaid)])

⟨ Widely aligned recognizer elements 558 ⟩ +≡
```
ZWA t_zwas;
```

620. ⟨ Initialize recognizer elements 554 ⟩ +≡
```
{
  ZWAID zwaid;
  const int zwa_count ⇐ ZWA_Count_of_R(r);
  (r)→t_zwas ⇐ marpa_obs_new(r→t_obs, ZWA_Object, ZWA_Count_of_R(r));
  for (zwaid ⇐ 0; zwaid < zwa_count; zwaid++) {
    const GZWA gzwa ⇐ GZWA_by_ID(zwaid);
    const ZWA zwa ⇐ RZWA_by_ID(zwaid);
    ID_of_ZWA(zwa) ⇐ ID_of_GZWA(gzwa);
    Default_Value_of_ZWA(zwa) ⇐ Default_Value_of_GZWA(gzwa);
    Memo_Value_of_ZWA(zwa) ⇐ Default_Value_of_GZWA(gzwa);
    Memo_YSID_of_ZWA(zwa) ⇐ −1;
  }
}
```

621. Earlemes. In most parsers, the input is modeled as a token stream — a sequence of tokens. In this model the idea of location is not complex. The first token is at location 0, the second at location 1, etc.

622. Marpa allows ambiguous and variable length tokens, and requires a more flexible idea of location, with a unit of length. The unit of token length in Marpa is called an Earleme. The locations themselves are often called earlemes.

623. `JEARLEME_THRESHOLD` is less than `INT_MAX` so that I can prevent overflow without getting fancy – overflow by addition is impossible as long as earlemes are below the threshold.

624. I considered defining earlemes as *long* or explicitly as 64-bit integers. But machines with 32-bit int's will in a not very long time become museum pieces. And in the meantime this definition of `JEARLEME_THRESHOLD` probably allows as large as parse as the memories on those machines will be able to handle.

#define `JEARLEME_THRESHOLD` (`INT_MAX`/4)

⟨ Public typedefs 91 ⟩ +≡
typedef int Marpa_Earleme;

625. ⟨ Private typedefs 49 ⟩ +≡
typedef Marpa_Earleme JEARLEME;

626. Earley set (YS) code.

⟨ Public typedefs 91 ⟩ +≡
```
typedef int Marpa_Earley_Set_ID;
```

627. ⟨ Private typedefs 49 ⟩ +≡
```
typedef Marpa_Earley_Set_ID YSID;
```

628.
```
#define Next_YS_of_YS(set) ((set)→t_next_earley_set)
#define Postdot_SYM_Count_of_YS(set) ((set)→t_postdot_sym_count)
#define First_PIM_of_YS_by_NSYID(set, nsyid)
        (first_pim_of_ys_by_nsyid((set), (nsyid)))
#define PIM_NSY_P_of_YS_by_NSYID(set, nsyid) (pim_nsy_p_find((set), (nsyid)))
```
⟨ Private incomplete structures 107 ⟩ +≡
```
struct s_earley_set;
typedef struct s_earley_set *YS;
typedef const struct s_earley_set *YS_Const;
struct s_earley_set_key;
typedef struct s_earley_set_key *YSK;
```

629. ⟨ Private structures 48 ⟩ +≡
```
struct s_earley_set_key {
  JEARLEME t_earleme;
};
typedef struct s_earley_set_key YSK_Object;
```

630. ⟨ Private structures 48 ⟩ +≡
```
struct s_earley_set {
  YSK_Object t_key;
  PIM *t_postdot_ary;
  YS t_next_earley_set;
  ⟨ Widely aligned Earley set elements 632 ⟩
  int t_postdot_sym_count;
  ⟨ Int aligned Earley set elements 631 ⟩
};
typedef struct s_earley_set YS_Object;
```

631. Earley item container.

```
#define YIM_Count_of_YS(set) ((set)→t_yim_count)
```
⟨ Int aligned Earley set elements 631 ⟩ ≡
```
int t_yim_count;
```
See also sections 633 and 637.

This code is used in section 630.

632. *#define* `YIMs_of_YS(set)` `((set)→t_earley_items)`

⟨ Widely aligned Earley set elements 632 ⟩ ≡
  YIM `*t_earley_items;`

See also section 1216.

This code is used in section 630.

633. Ordinal. The ordinal of the Earley set— its number in sequence. It is different from the earleme, because there may be gaps in the earleme sequence. There are never gaps in the sequence of ordinals.

#define `YS_Count_of_R(r)` `((r)→t_earley_set_count)`
#define `Ord_of_YS(set)` `((set)→t_ordinal)`

⟨ Int aligned Earley set elements 631 ⟩ +≡
  int `t_ordinal;`

634. *#define* `YS_Ord_is_Valid(r, ordinal)`
  `((ordinal) ≥ 0 ∧ (ordinal) < YS_Count_of_R(r))`

⟨ Int aligned recognizer elements 553 ⟩ +≡
  int `t_earley_set_count;`

635. ⟨ Initialize recognizer elements 554 ⟩ +≡
  `r→t_earley_set_count ⟸ 0;`

636. ID of Earley set.

#define `Earleme_of_YS(set)` `((set)→t_key.t_earleme)`

637. Values of Earley set. To be used for the application to associate an integer and a pointer value of its choice with each Earley set.

#define `Value_of_YS(set)` `((set)→t_value)`
#define `PValue_of_YS(set)` `((set)→t_pvalue)`

⟨ Int aligned Earley set elements 631 ⟩ +≡
  int `t_value;`
  void `*t_pvalue;`

638. ⟨ Initialize Earley set 638 ⟩ ≡
  `Value_of_YS(set) ⟸ −1;`
  `PValue_of_YS(set) ⟸ Λ;`

See also section 1217.

This code is used in section 643.

639. ⟨ Function definitions 41 ⟩ +≡
  int `marpa_r_earley_set_value(`*Marpa_Recognizer* `r`, *Marpa_Earley_Set_ID* `set_id)`
  {
    ⟨ Return −2 on failure 1229 ⟩
    YS `earley_set;`

```
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if recognizer not started 1246 ⟩
  if (set_id < 0) {
    MARPA_ERROR(MARPA_ERR_INVALID_LOCATION);
    return failure_indicator;
  }
  r_update_earley_sets(r);
  if (¬YS_Ord_is_Valid(r, set_id)) {
    MARPA_ERROR(MARPA_ERR_NO_EARLEY_SET_AT_LOCATION);
    return failure_indicator;
  }
  earley_set ⟸ YS_of_R_by_Ord(r, set_id);
  return Value_of_YS(earley_set);
}
```

640. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_earley_set_values(Marpa_Recognizer r, Marpa_Earley_Set_ID set_id, int
        *p_value, void **p_pvalue)
{
  ⟨ Return −2 on failure 1229 ⟩
  YS earley_set;
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if recognizer not started 1246 ⟩
  if (set_id < 0) {
    MARPA_ERROR(MARPA_ERR_INVALID_LOCATION);
    return failure_indicator;
  }
  r_update_earley_sets(r);
  if (¬YS_Ord_is_Valid(r, set_id)) {
    MARPA_ERROR(MARPA_ERR_NO_EARLEY_SET_AT_LOCATION);
    return failure_indicator;
  }
  earley_set ⟸ YS_of_R_by_Ord(r, set_id);
  if (p_value) *p_value ⟸ Value_of_YS(earley_set);
  if (p_pvalue) *p_pvalue ⟸ PValue_of_YS(earley_set);
  return 1;
}
```

641. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_latest_earley_set_value_set(Marpa_Recognizer r, int value)
{
  YS earley_set;
```

```
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    earley_set ⟸ Latest_YS_of_R(r);
    return Value_of_YS(earley_set) ⟸ value;
  }
```

642. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_r_latest_earley_set_values_set(Marpa_Recognizer r, int value, void
        *pvalue)
  {
    YS earley_set;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    earley_set ⟸ Latest_YS_of_R(r);
    Value_of_YS(earley_set) ⟸ value;
    PValue_of_YS(earley_set) ⟸ pvalue;
    return 1;
  }
```

643. Constructor.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE YS earley_set_new(RECCE r, JEARLEME id)
  {
    YSK_Object key;
    YS set;
    set ⟸ marpa_obs_new(r→t_obs, YS_Object, 1);
    key.t_earleme ⟸ id;
    set→t_key ⟸ key;
    set→t_postdot_ary ⟸ Λ;
    set→t_postdot_sym_count ⟸ 0;
    YIM_Count_of_YS(set) ⟸ 0;
    set→t_ordinal ⟸ r→t_earley_set_count++;
    YIMs_of_YS(set) ⟸ Λ;
    Next_YS_of_YS(set) ⟸ Λ;
    ⟨ Initialize Earley set 638 ⟩
    return set;
  }
```

644. Earley item (YIM) code.

645. Optimization Principles:

- Optimization should favor unambiguous grammars, but not heavily penalize ambiguous grammars.
- Optimization should favor mildly ambiguous grammars, but not heavily penalize very ambiguous grammars.
- Optimization should focus on saving space, perhaps even if at a slight cost in time.

646. Space savings are important because in practical applications there can easily be many millions of Earley items and links. If there are 1M copies of a structure, each byte saved is a 1M saved.

647. The solution arrived at is to optimize for Earley items with a single source, storing that source in the item itself. For Earley item with multiple sources, a special structure of linked lists is used. When a second source is added, the first source is copied into the lists, and its original space used for pointers to the linked lists.

648. This solution is optimized both for the unambiguous case, and for adding the third and additional sources. The only awkwardness takes place when the second source is added, and the first one must be recopied to make way for pointers to the linked lists.

```
#define LHS_NSYID_of_YIM(yim) LHS_NSYID_of_AHM(AHM_of_YIM(yim))
```

649. It might be slightly faster if this boolean is memoized in the Earley item when the Earley item is initialized.

```
#define YIM_is_Completion(item) (AHM_is_Completion(AHM_of_YIM(item)))
```

⟨ Public typedefs 91 ⟩ +≡

```
typedef int Marpa_Earley_Item_ID;
```

650. The ID of the Earley item is per-Earley-set, so that to uniquely specify the Earley item you must also specify the Earley set.

```
#define YS_of_YIM(yim) ((yim)→t_key.t_set)
#define YS_Ord_of_YIM(yim) (Ord_of_YS(YS_of_YIM(yim)))
#define Ord_of_YIM(yim) ((yim)→t_ordinal)
#define Earleme_of_YIM(yim) Earleme_of_YS(YS_of_YIM(yim))
#define AHM_of_YIM(yim) ((yim)→t_key.t_ahm)
#define AHMID_of_YIM(yim) ID_of_AHM(AHM_of_YIM(yim))
#define Postdot_NSYID_of_YIM(yim) Postdot_NSYID_of_AHM(AHM_of_YIM(yim))
#define IRL_of_YIM(yim) IRL_of_AHM(AHM_of_YIM(yim))
#define IRLID_of_YIM(yim) ID_of_IRL(IRL_of_YIM(yim))
#define XRL_of_YIM(yim) XRL_of_AHM(AHM_of_YIM(yim))
#define Origin_Earleme_of_YIM(yim) (Earleme_of_YS(Origin_of_YIM(yim)))
#define Origin_Ord_of_YIM(yim) (Ord_of_YS(Origin_of_YIM(yim)))
#define Origin_of_YIM(yim) ((yim)→t_key.t_origin)
```

⟨ Private incomplete structures 107 ⟩ +≡

```
struct s_earley_item;
typedef struct s_earley_item *YIM;
typedef const struct s_earley_item *YIM_Const;
struct s_earley_item_key;
typedef struct s_earley_item_key *YIK;
```

651. The layout matters a great deal, because there will be lots of them. I reduce the size of the YIM ordinal in order to save one word per YIM. I could widen it beyond the current count, but a limit of over 64,000 Earley items in a single Earley set should not be restrictive in practice.

```
#define YIM_ORDINAL_WIDTH 16
#define YIM_ORDINAL_CLAMP(x) (((1 << (YIM_ORDINAL_WIDTH)) - 1) & (x))
#define YIM_FATAL_THRESHOLD ((1 << (YIM_ORDINAL_WIDTH)) - 2)
#define YIM_is_Rejected(yim) ((yim)->t_is_rejected)
#define YIM_is_Active(yim) ((yim)->t_is_active)
#define YIM_was_Scanned(yim) ((yim)->t_was_scanned)
#define YIM_was_Fusion(yim) ((yim)->t_was_fusion)
```

⟨ Earley item structure 651 ⟩ ≡

```
struct s_earley_item_key {
  AHM t_ahm;
  YS t_origin;
  YS t_set;
};
typedef struct s_earley_item_key YIK_Object;
struct s_earley_item {
  YIK_Object t_key;
  union u_source_container t_container;
  BITFIELD t_ordinal:YIM_ORDINAL_WIDTH;
  BITFIELD t_source_type:3;
  BITFIELD t_is_rejected:1;
  BITFIELD t_is_active:1;
  BITFIELD t_was_scanned:1;
  BITFIELD t_was_fusion:1;
};
typedef struct s_earley_item YIM_Object;
```

This code is used in section 1383.

652. Signed as opposed to the the way it is kept (unsigned, for portability, because it is a bitfield. I may have to change this.

⟨ Private typedefs 49 ⟩ +≡

```
typedef int YIMID;
```

653. Constructor. Find an Earley item object, creating it if it does not exist. Only in a few cases per parse (in Earley set 0), do we already know that the Earley item is unique in the set. These are not worth optimizing for.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE YIM earley_item_create(const RECCE r, const YIK_Object key)
{
  ⟨ Return Λ on failure 1228 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  YIM new_item;
  YIM *end_of_work_stack;
  const YS set ⟸ key.t_set;
  const int count ⟸ ++YIM_Count_of_YS(set);
  ⟨ Check count against Earley item fatal threshold 655 ⟩
  new_item ⟸ marpa_obs_new(r→t_obs, struct s_earley_item, 1);
  new_item→t_key ⟸ key;
  new_item→t_source_type ⟸ NO_SOURCE;
  YIM_is_Rejected(new_item) ⟸ 0;
  YIM_is_Active(new_item) ⟸ 1;
  {
    SRCunique_yim_src ⟸ SRC_of_YIM(new_item);
    SRC_is_Rejected(unique_yim_src) ⟸ 0;
    SRC_is_Active(unique_yim_src) ⟸ 1;
  }
  Ord_of_YIM(new_item) ⟸ YIM_ORDINAL_CLAMP((unsigned int) count − 1);
  end_of_work_stack ⟸ WORK_YIM_PUSH(r);
  *end_of_work_stack ⟸ new_item;
  return new_item;
}
```

654. ⟨ Function definitions 41 ⟩ +≡
```
PRIVATE YIM earley_item_assign(const RECCE r, const YS set, const YS
        origin, const AHM ahm)
{
  const GRAMMAR g ⟸ G_of_R(r);
  YIK_Object key;
  YIM yim;
  PSL psl;
  AHMID ahm_id ⟸ ID_of_AHM(ahm);
  PSL *psl_owner ⟸ &Dot_PSL_of_YS(origin);
  if (¬*psl_owner) {
    psl_claim(psl_owner, Dot_PSAR_of_R(r));
  }
  psl ⟸ *psl_owner;
  yim ⟸ PSL_Datum(psl, ahm_id);
```

```
    if (yim ∧ Earleme_of_YIM(yim) ≡ Earleme_of_YS(set) ∧
          Earleme_of_YS(Origin_of_YIM(yim)) ≡ Earleme_of_YS(origin))
      {
      return yim;
    }
    key.t_origin ⟸ origin;
    key.t_ahm ⟸ ahm;
    key.t_set ⟸ set;
    yim ⟸ earley_item_create(r, key);
    PSL_Datum(psl, ahm_id) ⟸ yim;
    return yim;
  }
```

655. The fatal threshold always applies.

```
⟨Check count against Earley item fatal threshold 655⟩ ≡
  if (_MARPA_UNLIKELY(count ≥ YIM_FATAL_THRESHOLD)) {
      /* Set the recognizer to a fatal error */
    MARPA_FATAL(MARPA_ERR_YIM_COUNT);
    return failure_indicator;
  }
```

This code is used in section 653.

656. The warning threshold does not count against items added by a Leo expansion.

```
⟨Check count against Earley item warning threshold 656⟩ ≡
  {
    const int yim_count ⟸ YIM_Count_of_YS(current_earley_set);
    if (yim_count ≥ r→t_earley_item_warning_threshold) {
      int_event_new(g, MARPA_EVENT_EARLEY_ITEM_THRESHOLD, yim_count);
    }
  }
```

This code is used in section 737.

657. Destructor. No destructor. All earley item elements are either owned by other objects. The Earley item itself is on the obstack.

658. Source of the Earley item.

```
#define NO_SOURCE (0U)
#define SOURCE_IS_TOKEN (1U)
#define SOURCE_IS_COMPLETION (2U)
#define SOURCE_IS_LEO (3U)
#define SOURCE_IS_AMBIGUOUS (4U)
#define Source_Type_of_YIM(item) ((item)→t_source_type)
#define Earley_Item_has_No_Source(item) ((item)→t_source_type ≡ NO_SOURCE)
#define Earley_Item_has_Token_Source(item)
          ((item)→t_source_type ≡ SOURCE_IS_TOKEN)
```

```
#define Earley_Item_has_Complete_Source(item)
        ((item)→t_source_type ≡ SOURCE_IS_COMPLETION)
#define Earley_Item_has_Leo_Source(item)
        ((item)→t_source_type ≡ SOURCE_IS_LEO)
#define Earley_Item_is_Ambiguous(item)
        ((item)→t_source_type ≡ SOURCE_IS_AMBIGUOUS)
```

659. Not inline, because not used in critical paths. This is for creating error messages.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE_NOT_INLINE Marpa_Error_Code invalid_source_type_code(unsigned
        int type)
{
  switch (type) {
  case NO_SOURCE: return MARPA_ERR_SOURCE_TYPE_IS_NONE;
  case SOURCE_IS_TOKEN: return MARPA_ERR_SOURCE_TYPE_IS_TOKEN;
  case SOURCE_IS_COMPLETION: return MARPA_ERR_SOURCE_TYPE_IS_COMPLETION;
  case SOURCE_IS_LEO: return MARPA_ERR_SOURCE_TYPE_IS_LEO;
  case SOURCE_IS_AMBIGUOUS: return MARPA_ERR_SOURCE_TYPE_IS_AMBIGUOUS;
  }
  return MARPA_ERR_SOURCE_TYPE_IS_UNKNOWN;
}
```

660. Earley index (YIX) code. Postdot items are of two kinds: Earley indexes and Leo items. The payload of an Earley index is simple: a pointer to an Earley item. The other elements of the YIX are overhead to support the chain of postdot items for a postdot symbol.

#define `Next_PIM_of_YIX(yix)` ((`yix`)→`t_next`)
#define `YIM_of_YIX(yix)` ((`yix`)→`t_earley_item`)
#define `Postdot_NSYID_of_YIX(yix)` ((`yix`)→`t_postdot_nsyid`)

⟨ Private incomplete structures 107 ⟩ +≡

```
struct s_earley_ix;
typedef struct s_earley_ix *YIX;
```

661. ⟨ Private structures 48 ⟩ +≡

```
struct s_earley_ix {
  PIM t_next;
  NSYID t_postdot_nsyid;
  YIM t_earley_item;   /* NULL iff this is a LIM */
};
typedef struct s_earley_ix YIX_Object;
```

662. Leo item (LIM) code. Leo items originate from the "transition items" of Joop Leo's 1991 paper. They are set up so their first fields are identical to those of the Earley item indexes, so that they can be linked together in the same chain. Because the Earley index is at the beginning of each Leo item, LIMs can be treated as a kind of YIX.

```
#define YIX_of_LIM(lim) ((YIX)(lim))
```

663. Both Earley indexes and Leo items are postdot items, so that Leo items also require the fields to maintain the chain of postdot items. For this reason, Leo items contain an Earley index, but one with a Λ Earley item pointer.

```
#define Postdot_NSYID_of_LIM(leo) (Postdot_NSYID_of_YIX(YIX_of_LIM(leo)))
#define Next_PIM_of_LIM(leo) (Next_PIM_of_YIX(YIX_of_LIM(leo)))
#define Origin_of_LIM(leo) ((leo)→t_origin)
#define Top_AHM_of_LIM(leo) ((leo)→t_top_ahm)
#define Trailhead_AHM_of_LIM(leo) ((leo)→t_trailhead_ahm)
#define Predecessor_LIM_of_LIM(leo) ((leo)→t_predecessor)
#define Trailhead_YIM_of_LIM(leo) ((leo)→t_base)
#define YS_of_LIM(leo) ((leo)→t_set)
#define Earleme_of_LIM(lim) Earleme_of_YS(YS_of_LIM(lim))
#define LIM_is_Rejected(lim) ((lim)→t_is_rejected)
#define LIM_is_Active(lim) ((lim)→t_is_active)
```

```
⟨ Private incomplete structures 107 ⟩ +≡
  struct s_leo_item;
  typedef struct s_leo_item *LIM;
```

664.

```
⟨ Private structures 48 ⟩ +≡
  struct s_leo_item {
    YIX_Object t_earley_ix;
    ⟨ Widely aligned LIM elements 665 ⟩
    YS t_origin;
    AHM t_top_ahm;
    AHM t_trailhead_ahm;
    LIM t_predecessor;
    YIM t_base;
    YS t_set;
    BITFIELD t_is_rejected:1;
    BITFIELD t_is_active:1;
  };
  typedef struct s_leo_item LIM_Object;
```

665.

```
#define CIL_of_LIM(lim) ((lim)→t_cil)
```

```
⟨ Widely aligned LIM elements 665 ⟩ ≡
  CIL t_cil;
```

This code is used in section 664.

666. Postdot item (PIM) code. Postdot items are entries in an index, by postdot symbol, of both the Earley items and the Leo items for each Earley set.

```
#define LIM_of_PIM(pim) ((LIM)(pim))
#define YIX_of_PIM(pim) ((YIX)(pim))
#define Postdot_NSYID_of_PIM(pim) (Postdot_NSYID_of_YIX(YIX_of_PIM(pim)))
#define YIM_of_PIM(pim) (YIM_of_YIX(YIX_of_PIM(pim)))
#define Next_PIM_of_PIM(pim) (Next_PIM_of_YIX(YIX_of_PIM(pim)))
```

667. `PIM_of_LIM` assumes that PIM is in fact a LIM. `PIM_is_LIM` is available to check this.

```
#define PIM_of_LIM(pim) ((PIM)(pim))
#define PIM_is_LIM(pim) (YIM_of_PIM(pim) ≡ Λ)
```

⟨ Public incomplete structures 47 ⟩ +≡

```
union _Marpa_PIM_Object;
```

668. ⟨ Public typedefs 91 ⟩ +≡

```
typedef union _Marpa_PIM_Object *_Marpa_PIM;
```

669. ⟨ Private unions 669 ⟩ ≡

```
union _Marpa_PIM_Object {
  LIM_Object t_leo;
  YIX_Object t_earley;
};
```

This code is used in section 1381.

670. ⟨ Private typedefs 49 ⟩ +≡

```
typedef union _Marpa_PIM_Object PIM_Object;
typedef union _Marpa_PIM_Object *PIM;
```

671. This function searches for the first postdot item for an Earley set and a symbol ID. If successful, it returns that postdot item. If it fails, it returns Λ.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE PIM *pim_nsy_p_find(YS set, NSYID nsyid)
{
  int lo ⇐ 0;
  int hi ⇐ Postdot_SYM_Count_of_YS(set) − 1;
  PIM *postdot_array ⇐ set→t_postdot_ary;
  while (hi ≥ lo) {    /* A binary search */
    int trial ⇐ lo + (hi − lo)/2;    /* guards against overflow */
    PIM trial_pim ⇐ postdot_array[trial];
    NSYID trial_nsyid ⇐ Postdot_NSYID_of_PIM(trial_pim);
    if (trial_nsyid ≡ nsyid) return postdot_array + trial;
    if (trial_nsyid < nsyid) {
      lo ⇐ trial + 1;
```

```
    }
    else {
      hi ⟸ trial − 1;
    }
  }
  return Λ;
}
```

672. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE PIM first_pim_of_ys_by_nsyid(YS set, NSYID nsyid)
{
  PIM *pim_nsy_p ⟸ pim_nsy_p_find(set, nsyid);
  return pim_nsy_p ? *pim_nsy_p : Λ;
}
```

673. Source objects. Nothing internally distinguishes the various source objects by type. It is assumed that their type will be known from the context in which they are used.

674. The relationship between Leo items and ambiguity. The relationship between Leo items and ambiguous sources bears some explaining. Leo sources must be unique, but only when their predecessor's Earley set is considered. That is, for every pairing of Earley item and Earley set, there is only one Leo source in that Earley item with a predecessor in that Earley set. But there may be other sources (both Leo and non-Leo), a long as their predecessors are in different Earley sets.

675. One way to look at these Leo ambiguities is as different "factorings" of the Earley item. Call the last (or transition) symbol of an Earley item its "cause". An Earley item will often have both a predecessor and a cause, and these can "factor", or divide up, the distance between an Earley item's origin and its current set in different ways.

676. The Earley item can have only one origin, and only one transition symbol. But that transition symbol does not have to start at the origin and can start anywhere between the origin and the current set of the Earley item. For example, for an Earley item at earleme 14, with its origin at 10, there may be no predecessor, in which case the "cause" starts at 10. Or there may be a predecessor, in which case the "cause" may start at earlemes 11, 12 or 13. This different divisions between the (possibly null) predecessor and the "cause" are "factorings" of the Earley item.

677. Each factoring may have its own Leo source. At those earlemes without a Leo source, there may be any number of non-Leo sources.

678. Optimization. There will be a lot of these structures in a long parse, so space optimization gets an unusual amount of attention in the source links.

#define `Next_SRCL_of_SRCL(link)` ((`link`)→`t_next`)

679. ⟨ Private typedefs 49 ⟩ +≡

```
struct s_source;
typedef struct s_source *SRC;
typedef const struct s_source *SRC_Const;
```

680. ⟨ Source object structure 680 ⟩ ≡

```
struct s_token_source {
  NSYID t_nsyid;
  int t_value;
};
```

See also sections 681, 683, 684, and 685.

This code is used in section 1383.

681. To Do: There are a lot of these and some tricks to reduce the space used can be justified.

⟨ Source object structure 680 ⟩ +≡
```
struct s_source {
  void *t_predecessor;
  union {
    void *t_completion;
    struct s_token_source t_token;
  } t_cause;
  BITFIELD t_is_rejected:1;
  BITFIELD t_is_active:1;    /* A type field could go here */
};
```

682. ⟨ Private typedefs 49 ⟩ +≡
```
struct s_source_link;
typedef struct s_source_link *SRCL;
```

683. ⟨ Source object structure 680 ⟩ +≡
```
struct s_source_link {
  SRCL t_next;
  struct s_source t_source;
};
typedef struct s_source_link SRCL_Object;
```

684. ⟨ Source object structure 680 ⟩ +≡
```
struct s_ambiguous_source {
  SRCL t_leo;
  SRCL t_token;
  SRCL t_completion;
};
```

685. ⟨ Source object structure 680 ⟩ +≡
```
union u_source_container {
  struct s_ambiguous_source t_ambiguous;
  struct s_source_link t_unique;
};
```

686.
```
#define Source_of_SRCL(link) ((link)→t_source)
#define SRC_of_SRCL(link) (&Source_of_SRCL(link))
#define SRCL_of_YIM(yim) (&(yim)→t_container.t_unique)
#define Source_of_YIM(yim) ((yim)→t_container.t_unique.t_source)
#define SRC_of_YIM(yim) (&Source_of_YIM(yim))
#define Predecessor_of_Source(srcd) ((srcd).t_predecessor)
#define Predecessor_of_SRC(source) Predecessor_of_Source(*(source))
```

```
#define Predecessor_of_YIM(item) Predecessor_of_Source(Source_of_YIM(item))
#define Predecessor_of_SRCL(link) Predecessor_of_Source(Source_of_SRCL(link))
#define LIM_of_SRCL(link) ((LIM) Predecessor_of_SRCL(link))
#define Cause_of_Source(srcd) ((srcd).t_cause.t_completion)
#define Cause_of_SRC(source) Cause_of_Source(*(source))
#define Cause_of_YIM(item) Cause_of_Source(Source_of_YIM(item))
#define Cause_of_SRCL(link) Cause_of_Source(Source_of_SRCL(link))
#define TOK_of_Source(srcd) ((srcd).t_cause.t_token)
#define TOK_of_SRC(source) TOK_of_Source(*(source))
#define TOK_of_YIM(yim) TOK_of_Source(Source_of_YIM(yim))
#define TOK_of_SRCL(link) TOK_of_Source(Source_of_SRCL(link))
#define NSYID_of_Source(srcd) ((srcd).t_cause.t_token.t_nsyid)
#define NSYID_of_SRC(source) NSYID_of_Source(*(source))
#define NSYID_of_YIM(yim) NSYID_of_Source(Source_of_YIM(yim))
#define NSYID_of_SRCL(link) NSYID_of_Source(Source_of_SRCL(link))
#define Value_of_Source(srcd) ((srcd).t_cause.t_token.t_value)
#define Value_of_SRC(source) Value_of_Source(*(source))
#define Value_of_SRCL(link) Value_of_Source(Source_of_SRCL(link))
#define SRC_is_Active(src) ((src)->t_is_active)
#define SRC_is_Rejected(src) ((src)->t_is_rejected)
#define SRCL_is_Active(link) ((link)->t_source.t_is_active)
#define SRCL_is_Rejected(link) ((link)->t_source.t_is_rejected)
```

687.

```
#define Cause_AHMID_of_SRCL(srcl)
        AHMID_of_YIM((YIM) Cause_of_SRCL(srcl))
#define Leo_Transition_NSYID_of_SRCL(leo_source_link)
        Postdot_NSYID_of_LIM(LIM_of_SRCL(leo_source_link))
```

688. Macros for setting and finding the first *SRCL*'s of each type.

```
#define LV_First_Completion_SRCL_of_YIM(item)
        ((item)->t_container.t_ambiguous.t_completion)
#define First_Completion_SRCL_of_YIM(item)
        (Source_Type_of_YIM(item) ≡ SOURCE_IS_COMPLETION ? (SRCL)
            SRCL_of_YIM(item) : Source_Type_of_YIM(item) ≡ SOURCE_IS_AMBIGUOUS ?
            LV_First_Completion_SRCL_of_YIM(item) : Λ)
#define LV_First_Token_SRCL_of_YIM(item)
        ((item)->t_container.t_ambiguous.t_token)
#define First_Token_SRCL_of_YIM(item)
        (Source_Type_of_YIM(item) ≡ SOURCE_IS_TOKEN ? (SRCL)
            SRCL_of_YIM(item) : Source_Type_of_YIM(item) ≡ SOURCE_IS_AMBIGUOUS ?
            LV_First_Token_SRCL_of_YIM(item) : Λ)
#define LV_First_Leo_SRCL_of_YIM(item) ((item)->t_container.t_ambiguous.t_leo)
#define First_Leo_SRCL_of_YIM(item)
        (Source_Type_of_YIM(item) ≡ SOURCE_IS_LEO ? (SRCL)
```

```
    SRCL_of_YIM(item) : Source_Type_of_YIM(item) ≡ SOURCE_IS_AMBIGUOUS ?
    LV_First_Leo_SRCL_of_YIM(item) : Λ)
```

689. Creates unique (that is, not ambiguous) SRCL's.

```
⟨ Function definitions 41 ⟩ +≡
  PRIVATE SRCL unique_srcl_new(struct marpa_obstack *t_obs)
  {
    const SRCL new_srcl ⇐ marpa_obs_new(t_obs, SRCL_Object, 1);
    SRCL_is_Rejected(new_srcl) ⇐ 0;
    SRCL_is_Active(new_srcl) ⇐ 1;
    return new_srcl;
  }
```

690. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void tkn_link_add(RECCE r, YIM item, YIM predecessor,
        ALTalternative)
  {
    SRCL new_link;
    unsigned int previous_source_type ⇐ Source_Type_of_YIM(item);
    if (previous_source_type ≡ NO_SOURCE) {
      const SRCL source_link ⇐ SRCL_of_YIM(item);
      Source_Type_of_YIM(item) ⇐ SOURCE_IS_TOKEN;
      Predecessor_of_SRCL(source_link) ⇐ predecessor;
      NSYID_of_SRCL(source_link) ⇐ NSYID_of_ALT(alternative);
      Value_of_SRCL(source_link) ⇐ Value_of_ALT(alternative);
      Next_SRCL_of_SRCL(source_link) ⇐ Λ;
      return;
    }
    if (previous_source_type ≠ SOURCE_IS_AMBIGUOUS) {
        /* If the sourcing is not already ambiguous, make it so */
      earley_item_ambiguate(r, item);
    }
    new_link ⇐ unique_srcl_new(r→t_obs);
    new_link→t_next ⇐ LV_First_Token_SRCL_of_YIM(item);
    new_link→t_source.t_predecessor ⇐ predecessor;
    NSYID_of_Source(new_link→t_source) ⇐ NSYID_of_ALT(alternative);
    Value_of_Source(new_link→t_source) ⇐ Value_of_ALT(alternative);
    LV_First_Token_SRCL_of_YIM(item) ⇐ new_link;
  }
```

691. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void completion_link_add(RECCE r, YIM item, YIM predecessor, YIM
        cause)
{
  SRCL new_link;
  unsigned int previous_source_type ⟸ Source_Type_of_YIM(item);
  if (previous_source_type ≡ NO_SOURCE) {
    const SRCL source_link ⟸ SRCL_of_YIM(item);
    Source_Type_of_YIM(item) ⟸ SOURCE_IS_COMPLETION;
    Predecessor_of_SRCL(source_link) ⟸ predecessor;
    Cause_of_SRCL(source_link) ⟸ cause;
    Next_SRCL_of_SRCL(source_link) ⟸ Λ;
    return;
  }
  if (previous_source_type ≠ SOURCE_IS_AMBIGUOUS) {
      /* If the sourcing is not already ambiguous, make it so */
    earley_item_ambiguate(r, item);
  }
  new_link ⟸ unique_srcl_new(r→t_obs);
  new_link→t_next ⟸ LV_First_Completion_SRCL_of_YIM(item);
  new_link→t_source.t_predecessor ⟸ predecessor;
  Cause_of_Source(new_link→t_source) ⟸ cause;
  LV_First_Completion_SRCL_of_YIM(item) ⟸ new_link;
}
```

692. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void leo_link_add(RECCE r, YIM item, LIM predecessor, YIM cause)
{
  SRCL new_link;
  unsigned int previous_source_type ⟸ Source_Type_of_YIM(item);
  if (previous_source_type ≡ NO_SOURCE) {
    const SRCL source_link ⟸ SRCL_of_YIM(item);
    Source_Type_of_YIM(item) ⟸ SOURCE_IS_LEO;
    Predecessor_of_SRCL(source_link) ⟸ predecessor;
    Cause_of_SRCL(source_link) ⟸ cause;
    Next_SRCL_of_SRCL(source_link) ⟸ Λ;
    return;
  }
  if (previous_source_type ≠ SOURCE_IS_AMBIGUOUS) {
      /* If the sourcing is not already ambiguous, make it so */
    earley_item_ambiguate(r, item);
  }
  new_link ⟸ unique_srcl_new(r→t_obs);
  new_link→t_next ⟸ LV_First_Leo_SRCL_of_YIM(item);
```

```
    new_link→t_source.t_predecessor ⇐ predecessor;
    Cause_of_Source(new_link→t_source) ⇐ cause;
    LV_First_Leo_SRCL_of_YIM(item) ⇐ new_link;
  }
```

693. Convert an Earley item to an ambiguous one. `earley_item_ambiguate` assumes it is called when there is exactly one source. In other words, is assumes that the Earley item is not unsourced, and that it is not already ambiguous. Ambiguous sources should have more than one source, and `earley_item_ambiguate` is assuming that a new source will be added as followup.

694. Inlining `earley_item_ambiguate` might help in some circumstance, but at this point `earley_item_ambiguate` is not marked *inline*. `earley_item_ambiguate` is not short, it is referenced in several places, it is only called for ambiguous Earley items, and even for these it is only called when the Earley item first becomes ambiguous.

⟨Function definitions 41⟩ +≡

```
  PRIVATE_NOT_INLINE void earley_item_ambiguate(struct marpa_r *r, YIM item)
  {
    unsigned int previous_source_type ⇐ Source_Type_of_YIM(item);
    Source_Type_of_YIM(item) ⇐ SOURCE_IS_AMBIGUOUS;
    switch (previous_source_type) {
    case SOURCE_IS_TOKEN: ⟨Ambiguate token source 695⟩
      return;
    case SOURCE_IS_COMPLETION: ⟨Ambiguate completion source 696⟩
      return;
    case SOURCE_IS_LEO: ⟨Ambiguate Leo source 697⟩
      return;
    }
  }
```

695. ⟨Ambiguate token source 695⟩ ≡

```
  {
    SRCL new_link ⇐ marpa_obs_new(r→t_obs, SRCL_Object, 1);
    *new_link ⇐ *SRCL_of_YIM(item);
    LV_First_Leo_SRCL_of_YIM(item) ⇐ Λ;
    LV_First_Completion_SRCL_of_YIM(item) ⇐ Λ;
    LV_First_Token_SRCL_of_YIM(item) ⇐ new_link;
  }
```

This code is used in section 694.

696. ⟨Ambiguate completion source 696⟩ ≡

```
  {
    SRCL new_link ⇐ marpa_obs_new(r→t_obs, SRCL_Object, 1);
    *new_link ⇐ *SRCL_of_YIM(item);
    LV_First_Leo_SRCL_of_YIM(item) ⇐ Λ;
```

```
    LV_First_Completion_SRCL_of_YIM(item) ⟸ new_link;
    LV_First_Token_SRCL_of_YIM(item) ⟸ Λ;
  }
```

This code is used in section 694.

697. ⟨ Ambiguate Leo source 697 ⟩ ≡

```
  {
    SRCL new_link ⟸ marpa_obs_new(r→t_obs, SRCL_Object, 1);
    *new_link ⟸ *SRCL_of_YIM(item);
    LV_First_Leo_SRCL_of_YIM(item) ⟸ new_link;
    LV_First_Completion_SRCL_of_YIM(item) ⟸ Λ;
    LV_First_Token_SRCL_of_YIM(item) ⟸ Λ;
  }
```

This code is used in section 694.

698. Alternative tokens (ALT) code. Because Marpa allows more than one token at every earleme, Marpa's tokens are also called "alternatives".

⟨ Private incomplete structures 107 ⟩ +≡

```
struct s_alternative;
typedef struct s_alternative *ALT;
typedef const struct s_alternative *ALT_Const;
```

699.

```
#define NSYID_of_ALT(alt) ((alt)→t_nsyid)
#define Value_of_ALT(alt) ((alt)→t_value)
#define ALT_is_Valued(alt) ((alt)→t_is_valued)
#define Start_YS_of_ALT(alt) ((alt)→t_start_earley_set)
#define Start_Earleme_of_ALT(alt) Earleme_of_YS(Start_YS_of_ALT(alt))
#define End_Earleme_of_ALT(alt) ((alt)→t_end_earleme)
```

⟨ Private structures 48 ⟩ +≡

```
struct s_alternative {
  YS t_start_earley_set;
  JEARLEME t_end_earleme;
  NSYID t_nsyid;
  int t_value;
  BITFIELD t_is_valued:1;
};
typedef struct s_alternative ALT_Object;
```

700. ⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
MARPA_DSTACK_DECLARE(t_alternatives);
```

701.

⟨ Initialize recognizer elements 554 ⟩ +≡

```
MARPA_DSTACK_INIT2(r→t_alternatives, ALT_Object);
```

702. ⟨ Destroy recognizer elements 561 ⟩ +≡

```
MARPA_DSTACK_DESTROY(r→t_alternatives);
```

703. This functions returns the index at which to insert a new alternative, or -1 if the new alternative is a duplicate. (Duplicate alternatives should not be inserted.)

704. A variation of binary search.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int alternative_insertion_point(RECCE r, ALT new_alternative)
{
  MARPA_DSTACK alternatives ⇐ &r→t_alternatives;
  ALT alternative;
  int hi ⇐ MARPA_DSTACK_LENGTH(*alternatives) − 1;
  int lo ⇐ 0;
```

```
    int trial;    /* Special case when zero alternatives. */
    if (hi < 0) return 0;
    alternative ⟸ MARPA_DSTACK_BASE(*alternatives, ALT_Object);
    for ( ; ; ) {
      int outcome;
      trial ⟸ lo + (hi − lo)/2;
      outcome ⟸ alternative_cmp(new_alternative, alternative + trial);
      if (outcome ≡ 0) return −1;
      if (outcome > 0) {
        lo ⟸ trial + 1;
      }
      else {
        hi ⟸ trial − 1;
      }
      if (hi < lo) return outcome > 0 ? trial + 1 : trial;
    }
  }
```

705. This is the comparison function for sorting alternatives. The alternatives array also acts as a stack, with the alternatives ending at the lowest numbered earleme on top of the stack. This allows alternatives to be popped off the stack as the earlemes are processed in numerical order.

706. So that the alternatives array can act as a stack, the end earleme of the alternatives must be the major key, and must sort in reverse order. Of the remaining two keys, the more minor key is the start earleme, because that way its slightly costlier evaluation can sometimes be avoided.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE int alternative_cmp(const ALT_Const a, const ALT_Const b)
  {
    int subkey ⟸ End_Earleme_of_ALT(b) − End_Earleme_of_ALT(a);
    if (subkey) return subkey;
    subkey ⟸ NSYID_of_ALT(a) − NSYID_of_ALT(b);
    if (subkey) return subkey;
    return Start_Earleme_of_ALT(a) − Start_Earleme_of_ALT(b);
  }
```

707. This function pops an alternative from the stack, if it matches the earleme argument. If no alternative on the stack has its end earleme at the earleme argument, Λ is returned. The data pointed to by the return value may be overwritten when new alternatives are added, so it must be used before the next call that adds data to the alternatives stack.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE ALT alternative_pop(RECCE r, JEARLEME earleme)
```

```
{
  MARPA_DSTACK alternatives ⟸ &r→t_alternatives;
  ALT end_of_stack ⟸ MARPA_DSTACK_TOP(∗alternatives, ALT_Object);
  if (¬end_of_stack) return Λ;
    /∗ Stop looking if the next alternative is at a later earleme. We do not test for
         earlier earlemes, because we call alternative_pop( ) for each successive
         earleme in integer order. ∗/
  if (earleme < End_Earleme_of_ALT(end_of_stack)) return Λ;
  return MARPA_DSTACK_POP(∗alternatives, ALT_Object);
}
```

708. This function inserts an alternative into the stack, in sorted order, if the alternative is not a duplicate. It returns -1 if the alternative is a duplicate, and the insertion point (which must be zero or more) otherwise.

709. **To Do**: I wonder if this would not have been better implemented as a linked list.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int alternative_insert(RECCE r, ALT new_alternative)
{
  ALT end_of_stack, base_of_stack;
  MARPA_DSTACK alternatives ⟸ &r→t_alternatives;
  int ix;
  int insertion_point ⟸ alternative_insertion_point(r, new_alternative);
  if (insertion_point < 0) return insertion_point;
    /∗ may change base ∗/
  end_of_stack ⟸ MARPA_DSTACK_PUSH(∗alternatives, ALT_Object);
    /∗ base will not change after this ∗/
  base_of_stack ⟸ MARPA_DSTACK_BASE(∗alternatives, ALT_Object);
  for (ix ⟸ (int)(end_of_stack − base_of_stack); ix > insertion_point; ix−−) {
    base_of_stack[ix] ⟸ base_of_stack[ix − 1];
  }
  base_of_stack[insertion_point] ⟸ ∗new_alternative;
  return insertion_point;
}
```

710. Starting recognizer input.

```
⟨ Function definitions 41 ⟩ +≡
  int marpa_r_start_input(Marpa_Recognizer r)
  {
    int return_value ⇐ 1;
    YS set0;
    YIK_Object key;
    IRL start_irl;
    AHM start_ahm;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Fail if recognizer started 1245 ⟩
    {
      ⟨ Declare marpa_r_start_input locals 712 ⟩
      Current_Earleme_of_R(r) ⇐ 0;
      ⟨ Set up terminal-related boolean vectors 718 ⟩
      G_EVENTS_CLEAR(g);
      set0 ⇐ earley_set_new(r, 0);
      Latest_YS_of_R(r) ⇐ set0;
      First_YS_of_R(r) ⇐ set0;
      if (G_is_Trivial(g)) {
        return_value += trigger_trivial_events(r);
        ⟨ Set r exhausted 611 ⟩
        goto CLEANUP;
      }
      Input_Phase_of_R(r) ⇐ R_DURING_INPUT;
      psar_reset(Dot_PSAR_of_R(r));
      ⟨ Allocate recognizer containers 771 ⟩
      ⟨ Initialize Earley item work stacks 727 ⟩
      start_irl ⇐ g→t_start_irl;
      start_ahm ⇐ First_AHM_of_IRL(start_irl);

      /* These will stay constant in every YIM added in this method */
      key.t_origin ⇐ set0;
      key.t_set ⇐ set0;
      key.t_ahm ⇐ start_ahm;
      earley_item_create(r, key);
      bv_clear(r→t_bv_irl_seen);
      bv_bit_set(r→t_bv_irl_seen, ID_of_IRL(start_irl));
      MARPA_DSTACK_CLEAR(r→t_irl_cil_stack);
      *MARPA_DSTACK_PUSH(r→t_irl_cil_stack, CIL) ⇐ LHS_CIL_of_AHM(start_ahm);
      while (1) {
        const CIL *const p_cil ⇐ MARPA_DSTACK_POP(r→t_irl_cil_stack, CIL);
        if (¬p_cil) break;
        {
```

```
        int cil_ix;
        const CIL this_cil ⇐ *p_cil;
        const int prediction_count ⇐ Count_of_CIL(this_cil);
        for (cil_ix ⇐ 0; cil_ix < prediction_count; cil_ix++) {
          const IRLID prediction_irlid ⇐ Item_of_CIL(this_cil, cil_ix);
          if (¬bv_bit_test_then_set(r→t_bv_irl_seen, prediction_irlid)) {
            const IRL prediction_irl ⇐ IRL_by_ID(prediction_irlid);
            const AHM prediction_ahm ⇐ First_AHM_of_IRL(prediction_irl);
    /* If any of the assertions fail, do not add this AHM to the YS, or look at anything
          predicted by it. */
            if (¬evaluate_zwas(r, 0, prediction_ahm)) continue;
            key.t_ahm ⇐ prediction_ahm;
            earley_item_create(r, key);
            *MARPA_DSTACK_PUSH(r→t_irl_cil_stack,
                CIL) ⇐ LHS_CIL_of_AHM(prediction_ahm);
          }
        }
      }
    }
    postdot_items_create(r, bv_ok_for_chain, set0);
    earley_set_update_items(r, set0);
    r→t_is_using_leo ⇐ r→t_use_leo_flag;
    trigger_events(r);
  CLEANUP: ;
    ⟨ Destroy marpa_r_start_input locals 713 ⟩
  }
  return return_value;
}
```

711. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int evaluate_zwas(RECCE r, YSID ysid, AHM ahm)
{
  int cil_ix;
  const CIL zwa_cil ⇐ ZWA_CIL_of_AHM(ahm);
  const int cil_count ⇐ Count_of_CIL(zwa_cil);
  for (cil_ix ⇐ 0; cil_ix < cil_count; cil_ix++) {
    int value;
    const ZWAID zwaid ⇐ Item_of_CIL(zwa_cil, cil_ix);
    const ZWA zwa ⇐ RZWA_by_ID(zwaid);
    /* Use the memoized value, if it is for this YS */
    MARPA_OFF_DEBUG3("At␣%s,␣evaluating␣assertion␣%ld", STRLOC, (long)
        zwaid);
    if (Memo_YSID_of_ZWA(zwa) ≡ ysid) {
```

```
      if (Memo_Value_of_ZWA(zwa)) continue;
      MARPA_OFF_DEBUG3("At␣%s:␣returning␣0␣for␣assertion␣%ld",STRLOC,(long)
          zwaid);
      return 0;
    }
    /* Calculate the value (currently always the default) and memoize it */
    value ⇐ Memo_Value_of_ZWA(zwa) ⇐ Default_Value_of_ZWA(zwa);
    Memo_YSID_of_ZWA(zwa) ⇐ ysid;
    /* If the assertion fails we are done Otherwise, continue to check assertions. */
    if (¬value) {
      MARPA_OFF_DEBUG3("At␣%s:␣returning␣0␣for␣assertion␣%ld",STRLOC,(long)
          zwaid);
      return 0;
    }
    MARPA_OFF_DEBUG3("At␣%s:␣value␣is␣1␣for␣assertion␣%ld",STRLOC,(long)
        zwaid);
  }
  return 1;
}
```

712. ⟨ Declare marpa_r_start_input locals 712 ⟩ ≡

```
const NSYID nsy_count ⇐ NSY_Count_of_G(g);
const NSYID xsy_count ⇐ XSY_Count_of_G(g);
Bit_Vector bv_ok_for_chain ⇐ bv_create(nsy_count);
```

This code is used in section 710.

713. ⟨ Destroy marpa_r_start_input locals 713 ⟩ ≡

```
bv_free(bv_ok_for_chain);
```

This code is used in section 710.

714. Read a token alternative. The ordinary semantics of a parser generator is a token-stream semantics. The input is a sequence of n tokens. Every token is of length 1. The tokens fill the locations from 0 to $n-1$. The first token goes into location 0, the next into location 1, and so on up to location $n-1$.

715. In Marpa terms, a token-stream corresponds to reading exactly one token alternative at every location. In Marpa, the input locations are also called earlemes.

716. Marpa allows other models of the input besides the token stream model. Tokens may be ambiguous – that is, more than one token may occur at any location. Tokens vary in length – tokens may be of any length greater than or equal to one. This means tokens can span multiple earlemes. As a consequence, there may be no tokens at some earlemes.

717. Boolean vectors to track terminals. A number of boolean vectors are used to track the valued status of terminal symbols. Whether a symbol is a terminal or not cannot be changed by the recognizer, but some symbols are "value unlocked" and will be set to valued or unvalued the first time they are encountered.

```
⟨ Widely aligned recognizer elements 558 ⟩ +≡
  LBV t_valued_terminal;
  LBV t_unvalued_terminal;
  LBV t_valued;
  LBV t_unvalued;
  LBV t_valued_locked;
```

718. ⟨ Set up terminal-related boolean vectors 718 ⟩ ≡

```
{
  XSYID xsy_id;
  r→t_valued_terminal ⟸ lbv_obs_new0(r→t_obs, xsy_count);
  r→t_unvalued_terminal ⟸ lbv_obs_new0(r→t_obs, xsy_count);
  r→t_valued ⟸ lbv_obs_new0(r→t_obs, xsy_count);
  r→t_unvalued ⟸ lbv_obs_new0(r→t_obs, xsy_count);
  r→t_valued_locked ⟸ lbv_obs_new0(r→t_obs, xsy_count);
  for (xsy_id ⟸ 0; xsy_id < xsy_count; xsy_id++) {
    const XSY xsy ⟸ XSY_by_ID(xsy_id);
    if (XSY_is_Valued_Locked(xsy)) {
      lbv_bit_set(r→t_valued_locked, xsy_id);
    }
    if (XSY_is_Valued(xsy)) {
      lbv_bit_set(r→t_valued, xsy_id);
      if (XSY_is_Terminal(xsy)) {
        lbv_bit_set(r→t_valued_terminal, xsy_id);
      }
    }
    else {
      lbv_bit_set(r→t_unvalued, xsy_id);
```

```
      if (XSY_is_Terminal(xsy)) {
        lbv_bit_set(r→t_unvalued_terminal, xsy_id);
      }
    }
  }
}
```

This code is used in section 710.

719. `marpa_r_alternative`, by enforcing a limit on token length and on the furthest location, indirectly enforces a limit on the number of earley sets and the maximum earleme location. If tokens ending at location n cannot be scanned, then clearly the parse can never reach location n.

```
⟨Function definitions 41⟩ +≡
  Marpa_Earleme marpa_r_alternative(Marpa_Recognizer r, Marpa_Symbol_ID
          tkn_xsy_id, int value, int length)
  {
    ⟨Unpack recognizer objects 560⟩
    YS current_earley_set;
    const JEARLEME current_earleme ⟸ Current_Earleme_of_R(r);
    JEARLEME target_earleme;
    NSYID tkn_nsyid;
    if (_MARPA_UNLIKELY(¬R_is_Consistent(r))) {
      MARPA_ERROR(MARPA_ERR_RECCE_IS_INCONSISTENT);
      return MARPA_ERR_RECCE_IS_INCONSISTENT;
    }
    if (_MARPA_UNLIKELY(Input_Phase_of_R(r) ≠ R_DURING_INPUT)) {
      MARPA_ERROR(MARPA_ERR_RECCE_NOT_ACCEPTING_INPUT);
      return MARPA_ERR_RECCE_NOT_ACCEPTING_INPUT;
    }
    if (_MARPA_UNLIKELY(XSYID_is_Malformed(tkn_xsy_id))) {
      MARPA_ERROR(MARPA_ERR_INVALID_SYMBOL_ID);
      return MARPA_ERR_INVALID_SYMBOL_ID;
    }
    if (_MARPA_UNLIKELY(¬XSYID_of_G_Exists(tkn_xsy_id))) {
      MARPA_ERROR(MARPA_ERR_NO_SUCH_SYMBOL_ID);
      return MARPA_ERR_NO_SUCH_SYMBOL_ID;
    }
    ⟨marpa_alternative initial check for failure conditions 720⟩
    ⟨Set current_earley_set, failing if token is unexpected 723⟩
    ⟨Set target_earleme or fail 721⟩
    ⟨Insert alternative into stack, failing if token is duplicate 724⟩
    return MARPA_ERR_NONE;
  }
```

720. ⟨ marpa_alternative initial check for failure conditions 720 ⟩ ≡

```
{
  const XSY_Const tkn ⟸ XSY_by_ID(tkn_xsy_id);
  if (length ≤ 0) {
    MARPA_ERROR(MARPA_ERR_TOKEN_LENGTH_LE_ZERO);
    return MARPA_ERR_TOKEN_LENGTH_LE_ZERO;
  }
  if (length ≥ JEARLEME_THRESHOLD) {
    MARPA_ERROR(MARPA_ERR_TOKEN_TOO_LONG);
    return MARPA_ERR_TOKEN_TOO_LONG;
  }
  if (value ∧ _MARPA_UNLIKELY(¬lbv_bit_test(r→t_valued_terminal, tkn_xsy_id)))
    {
    if (¬XSY_is_Terminal(tkn)) {
      MARPA_ERROR(MARPA_ERR_TOKEN_IS_NOT_TERMINAL);
      return MARPA_ERR_TOKEN_IS_NOT_TERMINAL;
    }
    if (lbv_bit_test(r→t_valued_locked, tkn_xsy_id)) {
      MARPA_ERROR(MARPA_ERR_SYMBOL_VALUED_CONFLICT);
      return MARPA_ERR_SYMBOL_VALUED_CONFLICT;
    }
    lbv_bit_set(r→t_valued_locked, tkn_xsy_id);
    lbv_bit_set(r→t_valued_terminal, tkn_xsy_id);
    lbv_bit_set(r→t_valued, tkn_xsy_id);
  }
  if (¬value ∧ _MARPA_UNLIKELY(¬lbv_bit_test(r→t_unvalued_terminal,
        tkn_xsy_id))) {
    if (¬XSY_is_Terminal(tkn)) {
      MARPA_ERROR(MARPA_ERR_TOKEN_IS_NOT_TERMINAL);
      return MARPA_ERR_TOKEN_IS_NOT_TERMINAL;
    }
    if (lbv_bit_test(r→t_valued_locked, tkn_xsy_id)) {
      MARPA_ERROR(MARPA_ERR_SYMBOL_VALUED_CONFLICT);
      return MARPA_ERR_SYMBOL_VALUED_CONFLICT;
    }
    lbv_bit_set(r→t_valued_locked, tkn_xsy_id);
    lbv_bit_set(r→t_unvalued_terminal, tkn_xsy_id);
    lbv_bit_set(r→t_unvalued, tkn_xsy_id);
  }
}
```

This code is used in section 719.

721. ⟨ Set target_earleme or fail 721 ⟩ ≡

```
{
  target_earleme ⟸ current_earleme + length;
  if (target_earleme ≥ JEARLEME_THRESHOLD) {
    MARPA_ERROR(MARPA_ERR_PARSE_TOO_LONG);
    return MARPA_ERR_PARSE_TOO_LONG;
  }
}
```

This code is used in section 719.

722. If no postdot item is found at the current Earley set for this item, the token ID is unexpected, and `soft_failure` is returned. The application can treat this as a fatal error. The application can also use this as a mechanism to test alternatives, in which case, returning `soft_failure` is a perfectly normal data path. This last is part of an important technique: "Ruby Slippers" parsing.

723. Another case of an "unexpected" token is an inaccessible one. (A terminal must be productive but can be inaccessible.) Inaccessible tokens will not have an NSY and, since they don't derive from the start symbol, are always unexpected.

⟨ Set current_earley_set, failing if token is unexpected 723 ⟩ ≡

```
{
  NSY tkn_nsy ⟸ NSY_by_XSYID(tkn_xsy_id);
  if (_MARPA_UNLIKELY(¬tkn_nsy)) {
    MARPA_ERROR(MARPA_ERR_INACCESSIBLE_TOKEN);
    return MARPA_ERR_INACCESSIBLE_TOKEN;
  }
  tkn_nsyid ⟸ ID_of_NSY(tkn_nsy);
  current_earley_set ⟸ YS_at_Current_Earleme_of_R(r);
  if (¬current_earley_set) {
    MARPA_ERROR(MARPA_ERR_NO_TOKEN_EXPECTED_HERE);
    return MARPA_ERR_NO_TOKEN_EXPECTED_HERE;
  }
  if (¬First_PIM_of_YS_by_NSYID(current_earley_set, tkn_nsyid)) {
    MARPA_ERROR(MARPA_ERR_UNEXPECTED_TOKEN_ID);
    return MARPA_ERR_UNEXPECTED_TOKEN_ID;
  }
}
```

This code is used in section 719.

724. Insert an alternative into the alternatives stack, detecting if we are attempting to add the same token twice. Two tokens are considered the same if

- they have the same token ID, and
- they have the same length, and
- they have the same origin. Because `origin + token_length = current_earleme`,

Two tokens at the same current earleme are the same if they have the same token

ID and origin. By the same equation, two tokens at the same current earleme are the same if they have the same token ID and token length. It is up to the higher layers to determine if rejection of a duplicate token is a fatal error. The Earley sets and items will not have been altered by the attempt.

⟨ Insert alternative into stack, failing if token is duplicate 724 ⟩ ≡

```
{
  ALT_Object alternative_object;
    /* This is safe on the stack, because alternative_insert() will copy it if it is
       actually going to be used */
  const ALT alternative ⇐ &alternative_object;
  NSYID_of_ALT(alternative) ⇐ tkn_nsyid;
  Value_of_ALT(alternative) ⇐ value;
  ALT_is_Valued(alternative) ⇐ value ? 1 : 0;
  if (Furthest_Earleme_of_R(r) < target_earleme)
    Furthest_Earleme_of_R(r) ⇐ target_earleme;
  alternative→t_start_earley_set ⇐ current_earley_set;
  End_Earleme_of_ALT(alternative) ⇐ target_earleme;
  if (alternative_insert(r, alternative) < 0) {
    MARPA_ERROR(MARPA_ERR_DUPLICATE_TOKEN);
    return MARPA_ERR_DUPLICATE_TOKEN;
  }
}
```

This code is used in section 719.

725. Complete an Earley set. In the Aycock-Horspool variation of Earley's algorithm, the two main phases are scanning and completion. This section is devoted to the logic for completion.

```
#define Work_YIMs_of_R(r) MARPA_DSTACK_BASE((r)→t_yim_work_stack, YIM)
#define Work_YIM_Count_of_R(r) MARPA_DSTACK_LENGTH((r)→t_yim_work_stack)
#define WORK_YIMS_CLEAR(r) MARPA_DSTACK_CLEAR((r)→t_yim_work_stack)
#define WORK_YIM_PUSH(r) MARPA_DSTACK_PUSH((r)→t_yim_work_stack, YIM)
#define WORK_YIM_ITEM(r, ix)
        (*MARPA_DSTACK_INDEX((r)→t_yim_work_stack, YIM, ix))
```

⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
  MARPA_DSTACK_DECLARE(t_yim_work_stack);
```

726. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
  MARPA_DSTACK_SAFE(r→t_yim_work_stack);
```

727. ⟨ Initialize Earley item work stacks 727 ⟩ ≡

```
  {
    if (¬MARPA_DSTACK_IS_INITIALIZED(r→t_yim_work_stack)) {
      MARPA_DSTACK_INIT2(r→t_yim_work_stack, YIM);
    }
  }
```

See also section 731.

This code is used in section 710.

728. ⟨ Destroy recognizer elements 561 ⟩ +≡

```
  MARPA_DSTACK_DESTROY(r→t_yim_work_stack);
```

729. The completion stack is initialized to a very high-ball estimate of the number of completions per Earley set. It will grow if needed. Large stacks may needed for very ambiguous grammars.

⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
  MARPA_DSTACK_DECLARE(t_completion_stack);
```

730. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
  MARPA_DSTACK_SAFE(r→t_completion_stack);
```

731. ⟨ Initialize Earley item work stacks 727 ⟩ +≡

```
  {
    if (¬MARPA_DSTACK_IS_INITIALIZED(r→t_completion_stack)) {
      MARPA_DSTACK_INIT2(r→t_completion_stack, YIM);
    }
  }
```

732. ⟨ Destroy recognizer elements 561 ⟩ +≡

```
  MARPA_DSTACK_DESTROY(r→t_completion_stack);
```

733. ⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
MARPA_DSTACK_DECLARE(t_earley_set_stack);
```

734. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
MARPA_DSTACK_SAFE(r→t_earley_set_stack);
```

735. ⟨ Destroy recognizer elements 561 ⟩ +≡

```
MARPA_DSTACK_DESTROY(r→t_earley_set_stack);
```

736. This function returns the number of terminals expected on success. On failure, it returns −2. If the completion of the earleme left the parse exhausted, 0 is returned.

737. If the completion of the earleme left the parse exhausted, 0 is returned. The converse is not true – when tokens may be longer than one earleme, zero may be returned even if the parse is not exhausted. In those alternative input models, it is possible that no terminals are expected at the current earleme, but other terminals might be expected at later earlemes. That means that the parse can be continued — it is not exhausted. In those alternative input models, if the distinction between zero terminals expected and an exhausted parse is significant to the higher layers, they must explicitly check the phase whenever this function returns zero.

⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_earleme_complete(Marpa_Recognizer r)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  YIM *cause_p;
  YS current_earley_set;
  JEARLEME current_earleme;

    /* Initialized to -2 just in case. Should be set before returning; */
  JEARLEME return_value ⇐ −2;
  ⟨ Fail if recognizer not accepting input 1247 ⟩
  if (_MARPA_UNLIKELY(¬R_is_Consistent(r))) {
    MARPA_ERROR(MARPA_ERR_RECCE_IS_INCONSISTENT);
    return failure_indicator;
  }
  {
    int count_of_expected_terminals;
    ⟨ Declare marpa_r_earleme_complete locals 738 ⟩
    G_EVENTS_CLEAR(g);
    psar_dealloc(Dot_PSAR_of_R(r));
    bv_clear(r→t_bv_nsyid_is_expected);
    bv_clear(r→t_bv_irl_seen);
    ⟨ Initialize current_earleme 740 ⟩
    ⟨ Return 0 if no alternatives 742 ⟩
```

```
  ⟨ Initialize current_earley_set 741 ⟩
  ⟨ Scan from the alternative stack 743 ⟩
  ⟨ Pre-populate the completion stack 747 ⟩
  while ((cause_p ⟸ MARPA_DSTACK_POP(r→t_completion_stack, YIM))) {
    YIM cause ⟸ *cause_p;
    ⟨ Add new Earley items for cause 748 ⟩
  }
  ⟨ Add predictions to current_earley_set 753 ⟩
  postdot_items_create(r, bv_ok_for_chain, current_earley_set);
  /* If no terminals are expected, and there are no Earley items in uncompleted
       Earley sets, we can make no further progress. The parse is “exhausted”. */
  count_of_expected_terminals ⟸ bv_count(r→t_bv_nsyid_is_expected);
  if (count_of_expected_terminals ≤ 0 ∧
        MARPA_DSTACK_LENGTH(r→t_alternatives) ≤ 0) {
    ⟨ Set r exhausted 611 ⟩
  }
  earley_set_update_items(r, current_earley_set);
  ⟨ Check count against Earley item warning threshold 656 ⟩
  if (r→t_active_event_count > 0) {
    trigger_events(r);
  }
  return_value ⟸ G_EVENT_COUNT(g);
CLEANUP: ;
  ⟨ Destroy marpa_r_earleme_complete locals 739 ⟩
}
return return_value;
}
```

738. Currently, earleme_complete_obs is only used for completion events, and so should only be initialized if they are in use. But I expect to use it for other purposes.

⟨ Declare marpa_r_earleme_complete locals 738 ⟩ ≡

```
const NSYID nsy_count ⟸ NSY_Count_of_G(g);
Bit_Vector bv_ok_for_chain ⟸ bv_create(nsy_count);
struct marpa_obstack *const earleme_complete_obs ⟸ marpa_obs_init;
```

This code is used in section 737.

739. ⟨ Destroy marpa_r_earleme_complete locals 739 ⟩ ≡

```
bv_free(bv_ok_for_chain);
marpa_obs_free(earleme_complete_obs);
```

This code is used in section 737.

740. ⟨ Initialize current_earleme 740 ⟩ ≡

```
{
  current_earleme ⟸ ++(Current_Earleme_of_R(r));
  if (current_earleme > Furthest_Earleme_of_R(r)) {
    ⟨ Set r exhausted 611 ⟩
    MARPA_ERROR(MARPA_ERR_PARSE_EXHAUSTED);
    return_value ⟸ failure_indicator;
    goto CLEANUP;
  }
}
```

This code is used in section 737.

741. Create a new Earley set. We know that it does not exist.

⟨ Initialize current_earley_set 741 ⟩ ≡

```
{
  current_earley_set ⟸ earley_set_new(r, current_earleme);
  Next_YS_of_YS(Latest_YS_of_R(r)) ⟸ current_earley_set;
  Latest_YS_of_R(r) ⟸ current_earley_set;
}
```

This code is used in section 737.

742. If there are no alternatives for this earleme return 0 without creating an Earley set. The return value means success, with no events.

⟨ Return 0 if no alternatives 742 ⟩ ≡

```
{
  ALT end_of_stack ⟸ MARPA_DSTACK_TOP(r→t_alternatives, ALT_Object);
  if (¬end_of_stack ∨ current_earleme ≠ End_Earleme_of_ALT(end_of_stack)) {
    return_value ⟸ 0;
    goto CLEANUP;
  }
}
```

This code is used in section 737.

743. ⟨ Scan from the alternative stack 743 ⟩ ≡

```
{
  ALT alternative;

    /* alternative_pop() does not return inactive alternatives */
  while ((alternative ⟸ alternative_pop(r, current_earleme)))
    ⟨ Scan an Earley item from alternative 745 ⟩
}
```

This code is used in section 737.

744. The consequences of ignoring Leo items here is that a right recursion is always fully expanded when the cause of the Leo trailhead is a terminal. That's usually desireable, because a terminal at the bottom of the Leo trail is usually a sign that this is the trail that will be used in the parse.

745. But there are exceptions. These can occur in input models with ambiguous terminals, and when LHS terminals are used. These cases are not considered in the complexity claims, and as of this writing are not important in practical terms.

⟨ Scan an Earley item from alternative 745 ⟩ ≡

```
{
  YS start_earley_set ⇐ Start_YS_of_ALT(alternative);
  PIM pim ⇐ First_PIM_of_YS_by_NSYID(start_earley_set,
      NSYID_of_ALT(alternative));
  for ( ; pim; pim ⇐ Next_PIM_of_PIM(pim)) {
    /* Ignore Leo items when scanning */
    const YIM predecessor ⇐ YIM_of_PIM(pim);
    if (predecessor ∧ YIM_is_Active(predecessor)) {
      const AHM predecessor_ahm ⇐ AHM_of_YIM(predecessor);
      const AHM scanned_ahm ⇐ Next_AHM_of_AHM(predecessor_ahm);
      ⟨ Create the earley items for scanned_ahm 746 ⟩
    }
  }
}
```

This code is used in section 743.

746. ⟨ Create the earley items for scanned_ahm 746 ⟩ ≡

```
{
  const YIM scanned_earley_item ⇐ earley_item_assign(r, current_earley_set,
      Origin_of_YIM(predecessor), scanned_ahm);
  YIM_was_Scanned(scanned_earley_item) ⇐ 1;
  tkn_link_add(r, scanned_earley_item, predecessor, alternative);
}
```

This code is used in section 745.

747. At this point we know that only scanned items newly added are on the YIM working stack. Since they are newly added, and would not have been added if they were not active, we know that the YIM's on the working stack are all active.

⟨ Pre-populate the completion stack 747 ⟩ ≡

```
{
    /* We know that no new items are added to the stack in this scope */
  YIM *work_earley_items ⇐ MARPA_DSTACK_BASE(r→t_yim_work_stack, YIM);
  int no_of_work_earley_items ⇐ MARPA_DSTACK_LENGTH(r→t_yim_work_stack);
  int ix;
```

```
    MARPA_DSTACK_CLEAR(r→t_completion_stack);
    for (ix ⟸ 0; ix < no_of_work_earley_items; ix++) {
      YIM earley_item ⟸ work_earley_items[ix];
      YIM *end_of_stack;
      if (¬YIM_is_Completion(earley_item)) continue;
      end_of_stack ⟸ MARPA_DSTACK_PUSH(r→t_completion_stack, YIM);
      *end_of_stack ⟸ earley_item;
    }
  }
```

This code is used in section 737.

748. For the current completion cause, add those Earley items it "causes".

```
⟨Add new Earley items for cause 748⟩ ≡
  {
    if (YIM_is_Active(cause) ∧ YIM_is_Completion(cause)) {
      NSYID complete_nsyid ⟸ LHS_NSYID_of_YIM(cause);
      const YS middle ⟸ Origin_of_YIM(cause);
      ⟨Add new Earley items for complete_nsyid and cause 749⟩
    }
  }
```

This code is used in section 737.

749. ⟨Add new Earley items for complete_nsyid and cause 749⟩ ≡

```
  {
    PIM postdot_item;
    for (postdot_item ⟸ First_PIM_of_YS_by_NSYID(middle, complete_nsyid);
          postdot_item; postdot_item ⟸ Next_PIM_of_PIM(postdot_item)) {
      const YIM predecessor ⟸ YIM_of_PIM(postdot_item);
      if (¬predecessor) {
        /* A Leo item */
        const LIM leo_item ⟸ LIM_of_PIM(postdot_item);
        /* A Leo item */    /* If the Leo item is not active, look at the other item in
            the PIM, which might be active. (There should be exactly one other item,
            and it might be active if the LIM was inactive because of its predecessor,
            but had an active Leo trailhead */
        if (¬LIM_is_Active(leo_item)) goto NEXT_PIM;
        ⟨Add effect of leo_item 752⟩
        /* When I encounter an active Leo item, I skip everything else for this postdot
            symbol */
        goto LAST_PIM;
      }
      else {
        /* Not a Leo item */
```

```
        if (¬YIM_is_Active(predecessor)) continue;
        /* If we are here, both cause and predecessor are active */
        ⟨ Add effect_ahm for non-Leo predecessor 750 ⟩
      }
    NEXT_PIM: ;
    }
  LAST_PIM: ;
  }
```

This code is used in section 748.

750. ⟨ Add effect_ahm for non-Leo predecessor 750 ⟩ ≡

```
  {
    const AHM predecessor_ahm ⟸ AHM_of_YIM(predecessor);
    const AHM effect_ahm ⟸ Next_AHM_of_AHM(predecessor_ahm);
    const YS origin ⟸ Origin_of_YIM(predecessor);
    const YIM effect ⟸ earley_item_assign(r, current_earley_set, origin,
        effect_ahm);
    YIM_was_Fusion(effect) ⟸ 1;
    if (Earley_Item_has_No_Source(effect)) {
      /* If it has no source, then it is new */
      if (YIM_is_Completion(effect)) {
        ⟨ Push effect onto completion stack 751 ⟩
      }
    }
    completion_link_add(r, effect, predecessor, cause);
  }
```

This code is used in section 749.

751. The context must make sure any YIM pushed on the stack is active.

⟨ Push effect onto completion stack 751 ⟩ ≡

```
  {
    YIM *end_of_stack ⟸ MARPA_DSTACK_PUSH(r→t_completion_stack, YIM);
    *end_of_stack ⟸ effect;
  }
```

This code is used in sections 750 and 752.

752. If we are here, leo_item is active.

⟨ Add effect of leo_item 752 ⟩ ≡

```
  {
    const YS origin ⟸ Origin_of_LIM(leo_item);
    const AHM effect_ahm ⟸ Top_AHM_of_LIM(leo_item);
    const YIM effect ⟸ earley_item_assign(r, current_earley_set, origin,
        effect_ahm);
    YIM_was_Fusion(effect) ⟸ 1;
```

```
    if (Earley_Item_has_No_Source(effect)) {
      /* If it has no source, then it is new */
      ⟨Push effect onto completion stack 751⟩
    }
    leo_link_add(r, effect, leo_item, cause);
  }
```

This code is used in section 749.

753. ⟨Add predictions to current_earley_set 753⟩ ≡

```
  {
    int ix;
    const int no_of_work_earley_items ⇐
        MARPA_DSTACK_LENGTH(r→t_yim_work_stack);
    for (ix ⇐ 0; ix < no_of_work_earley_items; ix++) {
      YIM earley_item ⇐ WORK_YIM_ITEM(r, ix);
      int cil_ix;
      const AHM ahm ⇐ AHM_of_YIM(earley_item);
      const CIL prediction_cil ⇐ Predicted_IRL_CIL_of_AHM(ahm);
      const int prediction_count ⇐ Count_of_CIL(prediction_cil);
      for (cil_ix ⇐ 0; cil_ix < prediction_count; cil_ix++) {
        const IRLID prediction_irlid ⇐ Item_of_CIL(prediction_cil, cil_ix);
        const IRL prediction_irl ⇐ IRL_by_ID(prediction_irlid);
        const AHM prediction_ahm ⇐ First_AHM_of_IRL(prediction_irl);
        earley_item_assign(r, current_earley_set, current_earley_set,
            prediction_ahm);
      }
    }
  }
```

This code is used in section 737.

754. ⟨Function definitions 41⟩ +≡

```
  PRIVATE void trigger_events(RECCE r)
  {
    const GRAMMAR g ⇐ G_of_R(r);
    const YS current_earley_set ⇐ Latest_YS_of_R(r);
    int min, max, start;
    int yim_ix;
    struct marpa_obstack *const trigger_events_obs ⇐ marpa_obs_init;
    const YIM *yims ⇐ YIMs_of_YS(current_earley_set);
    const XSYID xsy_count ⇐ XSY_Count_of_G(g);
    const int ahm_count ⇐ AHM_Count_of_G(g);
    Bit_Vector bv_completion_event_trigger ⇐ bv_obs_create(trigger_events_obs,
        xsy_count);
```

```
Bit_Vector bv_nulled_event_trigger ⇐ bv_obs_create(trigger_events_obs,
    xsy_count);
Bit_Vector bv_prediction_event_trigger ⇐ bv_obs_create(trigger_events_obs,
    xsy_count);
Bit_Vector bv_ahm_event_trigger ⇐ bv_obs_create(trigger_events_obs,
    ahm_count);
const int working_earley_item_count ⇐ YIM_Count_of_YS(current_earley_set);
for (yim_ix ⇐ 0; yim_ix < working_earley_item_count; yim_ix++) {
  const YIM yim ⇐ yims[yim_ix];
  const AHM root_ahm ⇐ AHM_of_YIM(yim);
  if (AHM_has_Event(root_ahm)) {    /* Note that we go on to look at the Leo
        path, even if the top AHM is not an event AHM */
    bv_bit_set(bv_ahm_event_trigger, ID_of_AHM(root_ahm));
  }
  {    /* Now do the NSYs for any Leo links */
    const SRCL first_leo_source_link ⇐ First_Leo_SRCL_of_YIM(yim);
    SRCL setup_source_link;
    for (setup_source_link ⇐ first_leo_source_link; setup_source_link;
          setup_source_link ⇐ Next_SRCL_of_SRCL(setup_source_link)) {
      int cil_ix;
      const LIM lim ⇐ LIM_of_SRCL(setup_source_link);
      const CIL event_ahmids ⇐ CIL_of_LIM(lim);
      const int event_ahm_count ⇐ Count_of_CIL(event_ahmids);
      for (cil_ix ⇐ 0; cil_ix < event_ahm_count; cil_ix++) {
        const NSYID leo_path_ahmid ⇐ Item_of_CIL(event_ahmids, cil_ix);
        bv_bit_set(bv_ahm_event_trigger, leo_path_ahmid);    /* No need to
            test if AHM is an event AHM – all paths in the LIM's CIL will be */
      }
    }
  }
}
for (start ⇐ 0; bv_scan(bv_ahm_event_trigger, start, &min, &max);
      start ⇐ max + 2) {
  XSYID event_ahmid;
  for (event_ahmid ⇐ (NSYID) min; event_ahmid ≤ (NSYID) max;
        event_ahmid++) {
    int cil_ix;
    const AHM event_ahm ⇐ AHM_by_ID(event_ahmid);
    {
      const CIL completion_xsyids ⇐ Completion_XSYIDs_of_AHM(event_ahm);
      const int event_xsy_count ⇐ Count_of_CIL(completion_xsyids);
      for (cil_ix ⇐ 0; cil_ix < event_xsy_count; cil_ix++) {
        XSYID event_xsyid ⇐ Item_of_CIL(completion_xsyids, cil_ix);
```

```
        bv_bit_set(bv_completion_event_trigger, event_xsyid);
      }
    }
    {
      const CIL nulled_xsyids ⟸ Nulled_XSYIDs_of_AHM(event_ahm);
      const int event_xsy_count ⟸ Count_of_CIL(nulled_xsyids);
      for (cil_ix ⟸ 0; cil_ix < event_xsy_count; cil_ix++) {
        XSYID event_xsyid ⟸ Item_of_CIL(nulled_xsyids, cil_ix);
        bv_bit_set(bv_nulled_event_trigger, event_xsyid);
      }
    }
    {
      const CIL prediction_xsyids ⟸ Prediction_XSYIDs_of_AHM(event_ahm);
      const int event_xsy_count ⟸ Count_of_CIL(prediction_xsyids);
      for (cil_ix ⟸ 0; cil_ix < event_xsy_count; cil_ix++) {
        XSYID event_xsyid ⟸ Item_of_CIL(prediction_xsyids, cil_ix);
        bv_bit_set(bv_prediction_event_trigger, event_xsyid);
      }
    }
  }
}
if (Ord_of_YS(current_earley_set) ≤ 0) {      /* Because we special-case null
      parses, looking at the Earley items of the first Earley does not give us all the
      nulled symbols at earleme 0. If the parse can turn out to be zero length, all
      nullables derived from the start symbol (including itself) will be nulled, and
      therefore all of them should be null events at earleme 0. */
  int cil_ix;
  const XSY start_xsy ⟸ XSY_by_ID(g→t_start_xsy_id);
  const CIL nulled_xsyids ⟸ Nulled_XSYIDs_of_XSY(start_xsy);
  const int cil_count ⟸ Count_of_CIL(nulled_xsyids);
  for (cil_ix ⟸ 0; cil_ix < cil_count; cil_ix++) {
    const XSYID nulled_xsyid ⟸ Item_of_CIL(nulled_xsyids, cil_ix);
    bv_bit_set(bv_nulled_event_trigger, nulled_xsyid);
  }
}
for (start ⟸ 0; bv_scan(bv_completion_event_trigger, start, &min, &max);
      start ⟸ max + 2) {
  XSYID event_xsyid;
  for (event_xsyid ⟸ min; event_xsyid ≤ max; event_xsyid++) {
    if (lbv_bit_test(r→t_lbv_xsyid_completion_event_is_active, event_xsyid))
      {
      int_event_new(g, MARPA_EVENT_SYMBOL_COMPLETED, event_xsyid);
    }
```

```
      }
    }
    for (start ⟸ 0; bv_scan(bv_nulled_event_trigger, start, &min, &max);
          start ⟸ max + 2) {
      XSYID event_xsyid;
      for (event_xsyid ⟸ min; event_xsyid ≤ max; event_xsyid++) {
        if (lbv_bit_test(r→t_lbv_xsyid_nulled_event_is_active, event_xsyid)) {
          int_event_new(g, MARPA_EVENT_SYMBOL_NULLED, event_xsyid);
        }
      }
    }
    for (start ⟸ 0; bv_scan(bv_prediction_event_trigger, start, &min, &max);
          start ⟸ max + 2) {
      XSYID event_xsyid;
      for (event_xsyid ⟸ (NSYID) min; event_xsyid ≤ (NSYID) max;
            event_xsyid++) {
        if (lbv_bit_test(r→t_lbv_xsyid_prediction_event_is_active, event_xsyid))
          {
          int_event_new(g, MARPA_EVENT_SYMBOL_PREDICTED, event_xsyid);
        }
      }
    }
    marpa_obs_free(trigger_events_obs);
  }
```

755. Trigger events for trivial grammars. A trivial grammar is one which only accepts the null string.

This code takes no special measure to ensure that the order of nulled events is the same as in the non-trivial case. No guarantee of the order should be documented.

⟨Function definitions 41⟩ +≡

```
  PRIVATE int trigger_trivial_events(RECCE r)
  {
    int cil_ix;
    int event_count ⟸ 0;
    GRAMMAR g ⟸ G_of_R(r);
    const XSY start_xsy ⟸ XSY_by_ID(g→t_start_xsy_id);
    const CIL nulled_xsyids ⟸ Nulled_XSYIDs_of_XSY(start_xsy);
    const int cil_count ⟸ Count_of_CIL(nulled_xsyids);
    for (cil_ix ⟸ 0; cil_ix < cil_count; cil_ix++) {
      const XSYID nulled_xsyid ⟸ Item_of_CIL(nulled_xsyids, cil_ix);
      if (lbv_bit_test(r→t_lbv_xsyid_nulled_event_is_active, nulled_xsyid)) {
        int_event_new(g, MARPA_EVENT_SYMBOL_NULLED, nulled_xsyid);
        event_count++;
      }
```

```
    }
    return event_count;
  }
```

756. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void earley_set_update_items(RECCE r, YS set)
  {
    YIM *working_earley_items;
    YIM *finished_earley_items;
    int working_earley_item_count;
    int i;
    YIMs_of_YS(set) ⟸ marpa_obs_new(r→t_obs, YIM, YIM_Count_of_YS(set));
    finished_earley_items ⟸ YIMs_of_YS(set);
      /* We know that no new earley items will be added in this scope */
    working_earley_items ⟸ Work_YIMs_of_R(r);
    working_earley_item_count ⟸ Work_YIM_Count_of_R(r);
    for (i ⟸ 0; i < working_earley_item_count; i++) {
      YIM earley_item ⟸ working_earley_items[i];
      int ordinal ⟸ Ord_of_YIM(earley_item);
      finished_earley_items[ordinal] ⟸ earley_item;
    }
    WORK_YIMS_CLEAR(r);
  }
```

757. This function is called exactly once during a normal parse – at the end, when it is time for a bocage to be created. It is also called by trace and debugging methods. It must be used carefully since it takes $O(\log n)$ time, where n is the number of Earley sets. If called after every Earley set, it would make Marpa $O(n \log n)$ in the best case.

```
#define P_YS_of_R_by_Ord(r, ord)
        MARPA_DSTACK_INDEX((r)→t_earley_set_stack, YS, (ord))
#define YS_of_R_by_Ord(r, ord)  (*P_YS_of_R_by_Ord((r), (ord)))
```

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void r_update_earley_sets(RECCE r)
  {
    YS set;
    YS first_unstacked_earley_set;
    if (¬MARPA_DSTACK_IS_INITIALIZED(r→t_earley_set_stack)) {
      first_unstacked_earley_set ⟸ First_YS_of_R(r);
      MARPA_DSTACK_INIT(r→t_earley_set_stack, YS, MAX(1024, YS_Count_of_R(r)));
    }
    else {
      YS *end_of_stack ⟸ MARPA_DSTACK_TOP(r→t_earley_set_stack, YS);
      first_unstacked_earley_set ⟸ Next_YS_of_YS(*end_of_stack);
    }
```

```
for (set ⟸ first_unstacked_earley_set; set; set ⟸ Next_YS_of_YS(set)) {
  YS *end_of_stack ⟸ MARPA_DSTACK_PUSH(r→t_earley_set_stack, YS);
  (*end_of_stack) ⟸ set;
}
}
```

758. Create the postdot items.

759. About Leo items and unit rules.

760. Much of the logic in the code is required to allow the Leo logic to handle unit rules in right recursions. Right recursions that involve only unit rules might be overlooked – they are either finite in length (limited by the number of symbols in the grammar) or involve cycles. Either way, they could reasonably be ignored.

761. But a right recursion often takes place through multiple rules, and in practical cases following an important and lengthy right recursion, one with many non-unit rules, may require following short stretches of unit rules.

762. If a unit rule is the base item of a Leo item, it must be a prediction. This is because the base item will have a dot position that is penultimate – at the dot location just before the final one. In a unit rule this is the beginning of the rule.

763. Unit rules have a special issue when it comes to creating Leo items. Every Leo item, if it is to be useful and continue the recursion, needs to find a Leo predecessor. In the text that follows, recording the predecessor data in an Leo item is called "populating" that item.

764. The Leo predecessor of a unit rule Leo base item will be in the same Earley set that we are working on, and since this is the same Earley set for which we are creating Leo items, it may not have been built yet. Worse, it may be part of a cycle. To solve this problem, the code that follows builds LIM chains – chains of LIM's which require the next one on the chain to be populated. Every LIM on a LIM chain will have a base rule which is a unit rule and a prediction.

765. A chain ends
- when it results in a cycle, in which case the right recursion will not followed further.
- when a LIM is found which is not a unit rule, because that LIM's predecessor will be in a previous Earley set, and its information will be available.
- when it find a unit rule LIM which is populated, perhaps by a run through a previous LIM chain.

766. Code.

767. This function inserts regular and Leo postdot items into the postdot list. Not inlined, because of its size, and because it is used twice – once in initializing the Earley set 0, and once for completing later Earley sets. Earley set 0 is very much a special case, and it might be a good idea to have separate code to handle it, in which case both could be inlined.

768. Leo items are not created for Earley set 0. Originally this was to avoid dealing with the null productions that might be in Earley set 0. These have been eliminated with the special-casing of the null parse. But Leo items are always optional, and may not be worth it for Earley set 0.

769. Further Research: Another look at the degree and kind of memoization here is in order now that I use Leo items only in cases of an actual right recursion. This may require running benchmarks.

770. ⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
Bit_Vector t_bv_lim_symbols;
Bit_Vector t_bv_pim_symbols;
void **t_pim_workarea;
```

771. ⟨ Allocate recognizer containers 771 ⟩ ≡

```
r→t_bv_lim_symbols ⟸ bv_obs_create(r→t_obs, nsy_count);
r→t_bv_pim_symbols ⟸ bv_obs_create(r→t_obs, nsy_count);
r→t_pim_workarea ⟸ marpa_obs_new(r→t_obs, void *, nsy_count);
```

See also section 790.
This code is used in section 710.

772. ⟨ Reinitialize containers used in PIM setup 772 ⟩ ≡

```
bv_clear(r→t_bv_lim_symbols);
bv_clear(r→t_bv_pim_symbols);
```

This code is used in section 773.

773. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE_NOT_INLINE void postdot_items_create(RECCE r, Bit_Vector
        bv_ok_for_chain, const YS current_earley_set)
{
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Reinitialize containers used in PIM setup 772 ⟩
  ⟨ Start YIXes in PIM workarea 774 ⟩
  if (r→t_is_using_leo) {
    ⟨ Start LIMs in PIM workarea 776 ⟩
    ⟨ Add predecessors to LIMs 786 ⟩
  }
  ⟨ Copy PIM workarea to postdot item array 799 ⟩
  bv_and(r→t_bv_nsyid_is_expected, r→t_bv_pim_symbols,
      g→t_bv_nsyid_is_terminal);
}
```

774. This code creates the Earley indexes in the PIM workarea. At this point there are no Leo items.

⟨ Start YIXes in PIM workarea 774 ⟩ ≡

```
{   /* No new Earley items are created in this scope */
  YIM *work_earley_items ⟸ MARPA_DSTACK_BASE(r→t_yim_work_stack, YIM);
  int no_of_work_earley_items ⟸ MARPA_DSTACK_LENGTH(r→t_yim_work_stack);
  int ix;
  for (ix ⟸ 0; ix < no_of_work_earley_items; ix++) {
```

```
      YIM earley_item ⟸ work_earley_items[ix];
      AHM ahm ⟸ AHM_of_YIM(earley_item);
      const NSYID postdot_nsyid ⟸ Postdot_NSYID_of_AHM(ahm);
      if (postdot_nsyid < 0) continue;
      {
        PIM old_pim ⟸ Λ;
        PIM new_pim;    /* Need to be aligned for a PIM */
        new_pim ⟸ marpa__obs_alloc(r→t_obs, sizeof (YIX_Object),
            ALIGNOF(PIM_Object));
        Postdot_NSYID_of_PIM(new_pim) ⟸ postdot_nsyid;
        YIM_of_PIM(new_pim) ⟸ earley_item;
        if (bv_bit_test(r→t_bv_pim_symbols, postdot_nsyid))
          old_pim ⟸ r→t_pim_workarea[postdot_nsyid];
        Next_PIM_of_PIM(new_pim) ⟸ old_pim;
        if (¬old_pim) current_earley_set→t_postdot_sym_count++;
        r→t_pim_workarea[postdot_nsyid] ⟸ new_pim;
        bv_bit_set(r→t_bv_pim_symbols, postdot_nsyid);
      }
    }
  }
```

This code is used in section 773.

775. This code creates the Earley indexes in the PIM workarea. The Leo items do not contain predecessors or have the predecessor-dependent information set at this point.

776. The origin and predecessor will be filled in later, when the predecessor is known. The origin is set to Λ, and that will be used as an indicator that the fields of this Leo item have not been fully populated.

```
#define LIM_is_Populated(leo) (Origin_of_LIM(leo) ≠ Λ)
⟨ Start LIMs in PIM workarea 776 ⟩ ≡
  {
    int min, max, start;
    for (start ⟸ 0; bv_scan(r→t_bv_pim_symbols, start, &min, &max);
          start ⟸ max + 2) {
      NSYID nsyid;
      for (nsyid ⟸ (NSYID) min; nsyid ≤ (NSYID) max; nsyid++) {
        const PIM this_pim ⟸ r→t_pim_workarea[nsyid];
        if (Next_PIM_of_PIM(this_pim)) goto NEXT_NSYID;
            /* Do not create a Leo item if there is more than one YIX */
        {
          const YIM leo_base ⟸ YIM_of_PIM(this_pim);
          AHM potential_leo_penult_ahm ⟸ Λ;
          const AHM leo_base_ahm ⟸ AHM_of_YIM(leo_base);
          const IRL leo_base_irl ⟸ IRL_of_AHM(leo_base_ahm);
```

```
          if (¬IRL_is_Leo(leo_base_irl)) goto NEXT_NSYID;
          potential_leo_penult_ahm ⇐ leo_base_ahm;
          MARPA_ASSERT((int) potential_leo_penult_ahm);
          {
            const AHM trailhead_ahm ⇐
                Next_AHM_of_AHM(potential_leo_penult_ahm);
            if (AHM_is_Leo_Completion(trailhead_ahm)) {
              ⟨ Create a new, unpopulated, LIM 777 ⟩
            }
          }
        }
      NEXT_NSYID: ;
      }
    }
  }
```

This code is used in section 773.

777. The Top AHM of the new LIM is temporarily used to memoize the value of the AHM to-state for the LIM's base YIM. That may become its actual value, once it is populated.

⟨ Create a new, unpopulated, LIM 777 ⟩ ≡

```
  {
    LIM new_lim;
    new_lim ⇐ marpa_obs_new(r→t_obs, LIM_Object, 1);
    LIM_is_Active(new_lim) ⇐ 1;
    LIM_is_Rejected(new_lim) ⇐ 1;
    Postdot_NSYID_of_LIM(new_lim) ⇐ nsyid;
    YIM_of_PIM(new_lim) ⇐ Λ;
    Predecessor_LIM_of_LIM(new_lim) ⇐ Λ;
    Origin_of_LIM(new_lim) ⇐ Λ;
    CIL_of_LIM(new_lim) ⇐ Λ;
    Top_AHM_of_LIM(new_lim) ⇐ trailhead_ahm;
    Trailhead_AHM_of_LIM(new_lim) ⇐ trailhead_ahm;
    Trailhead_YIM_of_LIM(new_lim) ⇐ leo_base;
    YS_of_LIM(new_lim) ⇐ current_earley_set;
    Next_PIM_of_LIM(new_lim) ⇐ this_pim;
    r→t_pim_workarea[nsyid] ⇐ new_lim;
    bv_bit_set(r→t_bv_lim_symbols, nsyid);
  }
```

This code is used in section 776.

778. This code fully populates the data in the LIMs. It determines the Leo predecessors of the LIMs, if any, then populates that datum and the predecessor-dependent data.

779. The algorithm is fast, if not a model of simplicity. The LIMs are processed in an outer loop in order by symbol ID, as well as in an inner loop which processes predecessor chains from bottom to top. It is very much possible that the same LIM will be encountered twice, once in each loop. The code always checks to see if a LIM is already populated, before populating it.

780. The outer loop ensures that all LIMs are eventually populated. It uses the PIM workarea, guided by a boolean vector which indicates the LIM's.

781. It is possible for a LIM to be encountered which may have a predecessor, but which cannot be immediately populated. This is because predecessors link the LIMs in chains, and such chains must be populated in order. Any "links" in the chain of LIMs which are in previous Earley sets will already be populated. But a chain of LIMs may all be in the current Earley set, the one we are currently processing. In this case, there is a chicken-and-egg issue, which is resolved by arranging those LIMs in chain link order, and processing them in that order. This is the business of the inner loop.

782. When a LIM is encountered which cannot be populated immediately, its chain is followed and copied into `t_lim_chain`, which is in effect a stack. The chain ends when it reaches a LIM which can be populated immediately.

783. A special case is when the LIM chain cycles back to the LIM which started the chain. When this happens, the LIM chain is terminated. The bottom of such a chain (which, since it is a cycle, is also the top) is populated with a predecessor of Λ and appropriate predecessor-dependent data.

784. **Theorem**: The number of links in a LIM chain is never more than the number of symbols in the grammar. **Proof**: A LIM chain consists of the predecessors of LIMs, all of which are in the same Earley set. A LIM is uniquely determined by a duple of Earley set and transition symbol. This means, in a single Earley set, there is at most one LIM per symbol. **QED**.

785. **Complexity**: Time complexity is $O(n)$, where n is the number of LIMs. This can be shown as follows:

- The outer loop processes each LIM exactly once.
- A LIM is never put onto a LIM chain if it is already populated.
- A LIM is never taken off a LIM chain without being populated.
- Based on the previous two observations, we know that a LIM will be put onto a LIM chain at most once.
- Ignoring the inner loop processing, the amount of processing done for each LIM in the outer loop LIM is $O(1)$.
- The amount of processing done for each LIM in the inner loop is $O(1)$.
- Total processing for all n LIMs is therefore $n(O(1) + O(1)) = O(n)$.

786. The `bv_ok_for_chain` is a vector of bits by symbol ID. A bit is set if there is a LIM for that symbol ID that is OK for addition to the LIM chain. To be OK for addition to the LIM chain, the postdot item for the symbol ID must

- In fact actually be a Leo item (LIM).
- Must not have been populated.
- Must not have already been added to a LIM chain for this Earley set.

⟨ Add predecessors to LIMs 786 ⟩ ≡

```
{
  int min, max, start;
  bv_copy(bv_ok_for_chain, r→t_bv_lim_symbols);
  for (start ⇐ 0; bv_scan(r→t_bv_lim_symbols, start, &min, &max);
        start ⇐ max + 2) {      /* This is the outer loop. It loops over the symbols
        IDs, visiting only the symbols with LIMs. */
    NSYID main_loop_nsyid;
    for (main_loop_nsyid ⇐ (NSYID) min; main_loop_nsyid ≤ (NSYID) max;
          main_loop_nsyid++) {
      LIM predecessor_lim;
      LIM lim_to_process ⇐ r→t_pim_workarea[main_loop_nsyid];
      if (LIM_is_Populated(lim_to_process)) continue;
          /* LIM may have already been populated in the LIM chain loop */
      ⟨ Find predecessor LIM of unpopulated LIM 788 ⟩
      if (predecessor_lim ∧ LIM_is_Populated(predecessor_lim)) {
        ⟨ Populate lim_to_process from predecessor_lim 796 ⟩
        continue;
      }
      if (¬predecessor_lim) {      /* If there is no predecessor LIM to populate, we
            know that we should populate from the base Earley item */
        ⟨ Populate lim_to_process from its base Earley item 798 ⟩
        continue;
      }
      ⟨ Create and populate a LIM chain 791 ⟩
    }
  }
}
```

This code is used in section 773.

787. Find the predecessor LIM from the PIM workarea. If the predecessor starts at the current Earley set, I need to look in the PIM workarea. Otherwise the PIM item array by symbol is already set up and I can find it there.

788. The LHS of the completed rule and of the applicable rule in the base item will be the same, because the two rules are the same. Given the `main_loop_symbol_id` we can look up either the appropriate rule in the base Earley item's AHM, or the Leo completion's AHM. It is most convenient to find the LHS of the completed rule as the only possible Leo LHS of the Leo completion's AHM. The AHM for the Leo completion is guaranteed to have only one rule. The base Earley item's AHM can have multiple rules, and in its list of rules there can be transitions to Leo completions via several different symbols. The code is used for unpopulated LIMs. In a populated LIM, this will not necessarily be the case.

⟨ Find predecessor LIM of unpopulated LIM 788 ⟩ ≡

```
{
  const YIM base_yim ⇐ Trailhead_YIM_of_LIM(lim_to_process);
  const YS predecessor_set ⇐ Origin_of_YIM(base_yim);
  const AHM trailhead_ahm ⇐ Trailhead_AHM_of_LIM(lim_to_process);
  const NSYID predecessor_transition_nsyid ⇐ LHSID_of_AHM(trailhead_ahm);
  PIM predecessor_pim;
  if (Ord_of_YS(predecessor_set) < Ord_of_YS(current_earley_set)) {
    predecessor_pim ⇐ First_PIM_of_YS_by_NSYID(predecessor_set,
        predecessor_transition_nsyid);
  }
  else {
    predecessor_pim ⇐ r→t_pim_workarea[predecessor_transition_nsyid];
  }
  predecessor_lim ⇐ PIM_is_LIM(predecessor_pim) ?
      LIM_of_PIM(predecessor_pim) : Λ;
}
```

This code is used in sections 786 and 794.

789. ⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
void **t_lim_chain;
```

790. ⟨ Allocate recognizer containers 771 ⟩ +≡

```
r→t_lim_chain ⇐ marpa_obs_new(r→t_obs, void *, 2 * nsy_count);
```

791. ⟨ Create and populate a LIM chain 791 ⟩ ≡

```
{
  int lim_chain_ix;
  ⟨ Create a LIM chain 794 ⟩
  ⟨ Populate the LIMs in the LIM chain 795 ⟩
}
```

This code is used in section 786.

792. At this point we know that

- lim_to_process ≠ Λ
- lim_to_process is not populated
- predecessor_lim ≠ Λ
- predecessor_lim is not populated

793. Cycles can occur in the LIM chain. They are broken by refusing to put the same LIM on LIM chain twice. Since a LIM chain links are one-to-one, ensuring that the LIM on the bottom of the chain is never added to the LIM chain is enough to enforce this.

794. When I am about to add a LIM twice to the LIM chain, instead I break the chain at that point. The top of chain will then have no LIM predecessor, instead of being part of a cycle. Since the LIM information is always optional, and in that case would be useless, breaking the chain in this way causes no problems.

⟨ Create a LIM chain 794 ⟩ ≡

```
{
  NSYID postdot_nsyid_of_lim_to_process ⟸
      Postdot_NSYID_of_LIM(lim_to_process);
  lim_chain_ix ⟸ 0;
  r→t_lim_chain[lim_chain_ix++] ⟸ LIM_of_PIM(lim_to_process);
  bv_bit_clear(bv_ok_for_chain, postdot_nsyid_of_lim_to_process);
    /* Make sure this LIM is not added to a LIM chain again for this Earley set */
  while (1) {
    /* I know at this point that predecessor_lim is unpopulated, so I also know that
         lim_to_process is unpopulated. This means I also know that lim_to_process
         is in the current Earley set, because all LIMs in previous Earley sets are
         already populated. */
    lim_to_process ⟸ predecessor_lim;
    postdot_nsyid_of_lim_to_process ⟸ Postdot_NSYID_of_LIM(lim_to_process);
    if (¬bv_bit_test(bv_ok_for_chain, postdot_nsyid_of_lim_to_process)) {
        /* If I am about to add a previously added LIM to the LIM chain, I break the
           LIM chain at this point. The predecessor LIM has not yet been changed, so
           that it is still appropriate for the LIM at the top of the chain. */
      break;
    }
    ⟨ Find predecessor LIM of unpopulated LIM 788 ⟩
    r→t_lim_chain[lim_chain_ix++] ⟸ LIM_of_PIM(lim_to_process);
      /* lim_to_process is not populated, as shown above */
    bv_bit_clear(bv_ok_for_chain, postdot_nsyid_of_lim_to_process);
      /* Make sure this LIM is not added to a LIM chain again for this Earley set */

    /* predecesssor_lim ⟸ Λ, so that we are forced to break the LIM chain before
         it */
    if (¬predecessor_lim) break;
```

```
      if (LIM_is_Populated(predecessor_lim)) break;    /* predecesssor_lim is
            populated, so that if we break before predecessor_lim, we are ready to
            populate the entire LIM chain. */
    }
  }
```

This code is used in section 791.

795. ⟨ Populate the LIMs in the LIM chain 795 ⟩ ≡

```
  for (lim_chain_ix--; lim_chain_ix ≥ 0; lim_chain_ix--) {
    lim_to_process ⟸ r→t_lim_chain[lim_chain_ix];
    if (predecessor_lim ∧ LIM_is_Populated(predecessor_lim)) {
      ⟨ Populate lim_to_process from predecessor_lim 796 ⟩
    }
    else {
      ⟨ Populate lim_to_process from its base Earley item 798 ⟩
    }
    predecessor_lim ⟸ lim_to_process;
  }
```

This code is used in section 791.

796. This code is optimized for cases where there are no events, or the lists of AHM IDs is "at closure". These are the most frequent and worst case scenarios. The new remaining "worst case" is a recursive series of AHM ID's which stabilizes short of closure. Secondary optimzations ensure this is fairly cheap as well.

⟨ Populate lim_to_process from predecessor_lim 796 ⟩ ≡

```
  {
    const AHM new_top_ahm ⟸ Top_AHM_of_LIM(predecessor_lim);
    const CIL predecessor_cil ⟸ CIL_of_LIM(predecessor_lim);

      /* Initialize to be just the predcessor's list of AHM IDs. Overwrite if we need to
         add another. */
    CIL_of_LIM(lim_to_process) ⟸ predecessor_cil;
    Predecessor_LIM_of_LIM(lim_to_process) ⟸ predecessor_lim;
    Origin_of_LIM(lim_to_process) ⟸ Origin_of_LIM(predecessor_lim);
    if (Event_Group_Size_of_AHM(new_top_ahm) > Count_of_CIL(predecessor_cil)) {
        /* Might we need to add another AHM ID? */
      const AHM trailhead_ahm ⟸ Trailhead_AHM_of_LIM(lim_to_process);
      const CIL trailhead_ahm_event_ahmids ⟸
          Event_AHMIDs_of_AHM(trailhead_ahm);
      if (Count_of_CIL(trailhead_ahm_event_ahmids)) {
        CIL new_cil ⟸ cil_merge_one(&g→t_cilar, predecessor_cil,
            Item_of_CIL(trailhead_ahm_event_ahmids, 0));
        if (new_cil) {
          CIL_of_LIM(lim_to_process) ⟸ new_cil;
        }
```

```
      }
    }
    Top_AHM_of_LIM(lim_to_process) ⟸ new_top_ahm;
  }
```

This code is used in sections 786 and 795.

797. If we have reached this code, either we do not have a predecessor LIM, or we have one which is useless for populating lim_to_process. If a predecessor LIM is not itself populated, it will be useless for populating its successor. An unpopulated predecessor LIM may occur when there is a predecessor LIM which proved impossible to populate because it is part of a cycle.

798. The predecessor LIM and the top AHM to-state were initialized to the appropriate values for this case, and do not need to be changed. The predecessor LIM was initialized to Λ. of the base YIM.

⟨ Populate lim_to_process from its base Earley item 798 ⟩ ≡

```
  {
    const AHM trailhead_ahm ⟸ Trailhead_AHM_of_LIM(lim_to_process);
    const YIM base_yim ⟸ Trailhead_YIM_of_LIM(lim_to_process);
    Origin_of_LIM(lim_to_process) ⟸ Origin_of_YIM(base_yim);
    CIL_of_LIM(lim_to_process) ⟸ Event_AHMIDs_of_AHM(trailhead_ahm);
  }
```

This code is used in sections 786 and 795.

799. ⟨ Copy PIM workarea to postdot item array 799 ⟩ ≡

```
  {
    PIM *postdot_array ⟸ current_earley_set→t_postdot_ary ⟸
        marpa_obs_new(r→t_obs, PIM, current_earley_set→t_postdot_sym_count);
    int min, max, start;
    int postdot_array_ix ⟸ 0;
    for (start ⟸ 0; bv_scan(r→t_bv_pim_symbols, start, &min, &max);
          start ⟸ max + 2) {
      NSYID nsyid;
      for (nsyid ⟸ min; nsyid ≤ max; nsyid++) {
        PIM this_pim ⟸ r→t_pim_workarea[nsyid];
        if (lbv_bit_test(r→t_nsy_expected_is_event, nsyid)) {
          XSY xsy ⟸ Source_XSY_of_NSYID(nsyid);
          int_event_new(g, MARPA_EVENT_SYMBOL_EXPECTED, ID_of_XSY(xsy));
        }
        if (this_pim) postdot_array[postdot_array_ix++] ⟸ this_pim;
      }
    }
  }
```

This code is used in section 773.

800. Rejecting Earley items.

801. Notes for making the recognizer consistent after rejecting tokens:

- Clear all events. Document that you should poll events before any rejections.
- Reset the vector of expected terminals.
- Re-determine if the parse is exhausted.
- What about postdot items? If a LIM is now rejected, I should look at the YIM/PIM, I think, because it was **not** necessarily rejected.

802. Various notes about revision:

- I need to make sure that the reading of alternatives and the rejection of rules and terminals cannot be mixed. Rejected must be made, and revision complete, before any alternatives can be attempted. Or, in other words, attempting to reject a rule or terminal once an alternative has been read must be a fatal error.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_Earleme marpa_r_clean(Marpa_Recognizer r)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack recognizer objects 560 ⟩
    YSID ysid_to_clean;

  const YS current_ys ⇐ Latest_YS_of_R(r);
    const YSID current_ys_id ⇐ Ord_of_YS(current_ys);
    int count_of_expected_terminals;
    ⟨ Declare marpa_r_clean locals 803 ⟩
      /∗ Initialized to -2 just in case. Should be set before returning; ∗/
    const JEARLEME return_value ⇐ −2;
    ⟨ Fail if recognizer not accepting input 1247 ⟩
    G_EVENTS_CLEAR(g);
      /∗ Return success if recognizer is already consistent ∗/
    if (R_is_Consistent(r)) return 0;

        /∗ Note this makes revision O(n log n). I could do better for constant
         "look-behind", but it does not seem worth the bother ∗/
    earley_set_update_items(r, current_ys);
    for (ysid_to_clean ⇐ First_Inconsistent_YS_of_R(r);
          ysid_to_clean ≤ current_ys_id; ysid_to_clean++) {
      ⟨ Clean Earley set ysid_to_clean 805 ⟩
    }
      /∗ All Earley sets are now consistent ∗/
    ⟨ Clean pending alternatives 818 ⟩
    bv_clear(r→t_bv_nsyid_is_expected);
```

```
  ⟨ Clean expected terminals 820 ⟩
  count_of_expected_terminals ⟸ bv_count(r→t_bv_nsyid_is_expected);
  if (count_of_expected_terminals ≤ 0 ∧
        MARPA_DSTACK_LENGTH(r→t_alternatives) ≤ 0) {
    ⟨ Set r exhausted 611 ⟩
  }
  First_Inconsistent_YS_of_R(r) ⟸ −1;
    /* CLEANUP: ; – not used at the moment */
  ⟨ Destroy marpa_r_clean locals 804 ⟩
  return return_value;
}
```

803. ⟨ Declare marpa_r_clean locals 803 ⟩ ≡

```
    /* An obstack whose lifetime is that of the external method */
  struct marpa_obstack *const method_obstack ⟸ marpa_obs_init;
  YIMID *prediction_by_irl ⟸ marpa_obs_new(method_obstack, YIMID,
      IRL_Count_of_G(g));
```

This code is used in section 802.

804. ⟨ Destroy marpa_r_clean locals 804 ⟩ ≡

```
{
  marpa_obs_free(method_obstack);
}
```

This code is used in section 802.

805. ⟨ Clean Earley set ysid_to_clean 805 ⟩ ≡

```
{
  const YS ys_to_clean ⟸ YS_of_R_by_Ord(r, ysid_to_clean);
  const YIM *yims_to_clean ⟸ YIMs_of_YS(ys_to_clean);
  const int yim_to_clean_count ⟸ YIM_Count_of_YS(ys_to_clean);
  Bit_Matrix acceptance_matrix ⟸ matrix_obs_create(method_obstack,
      yim_to_clean_count, yim_to_clean_count);
  ⟨ Map prediction rules to YIM ordinals in array 806 ⟩
  ⟨ First revision pass over ys_to_clean 807 ⟩
  transitive_closure(acceptance_matrix);
  ⟨ Mark accepted YIM's 813 ⟩
  ⟨ Mark un-accepted YIM's rejected 814 ⟩
  ⟨ Mark accepted SRCL's 816 ⟩
  ⟨ Mark rejected LIM's 817 ⟩
}
```

This code is used in section 802.

806. Rules not used in this YS do not need to be initialized because they will never be referred to.

⟨ Map prediction rules to YIM ordinals in array 806 ⟩ ≡

```
{
  int yim_ix ⟸ yim_to_clean_count − 1;
  YIM yim ⟸ yims_to_clean[yim_ix];

    /* Assumes that predictions are last in the YS. There will always be a
       non-prediction to end the loop, because there is always a scanned or an initial
       YIM. */
  while (YIM_was_Predicted(yim)) {
    prediction_by_irl[IRLID_of_YIM(yim)] ⟸ yim_ix;
    yim ⟸ yims_to_clean[−−yim_ix];
  }
}
```

This code is used in section 805.

807. ⟨ First revision pass over ys_to_clean 807 ⟩ ≡

```
{
  int yim_to_clean_ix;
  for (yim_to_clean_ix ⟸ 0; yim_to_clean_ix < yim_to_clean_count;
        yim_to_clean_ix++) {
    const YIM yim_to_clean ⟸ yims_to_clean[yim_to_clean_ix];
    /* The initial YIM is always active and can never be rejected. */
    MARPA_ASSERT(¬YIM_is_Initial(yim_to_clean)∨(YIM_is_Active(yim_to_clean)∧
        ¬YIM_is_Rejected(yim_to_clean)));
    /* Non-initial YIM's are inactive until proven active. */
    if (¬YIM_is_Initial(yim_to_clean)) YIM_is_Active(yim_to_clean) ⟸ 0;
    /* If a YIM is rejected, which at this point means that it was directly rejected, that
          is the end of the story. We don't use it to update the acceptance matrix. */
    if (YIM_is_Rejected(yim_to_clean)) continue;
    /* Add un-rejected predictions to acceptance matrix. */
    ⟨ Add predictions from yim_to_clean to acceptance matrix 808 ⟩
    /* YIM's may have both scanned and fusion links. Change the following so it looks
        at both kinds of link for all YIM's. */
  }
}
```

This code is used in section 805.

808. ⟨ Add predictions from `yim_to_clean` to acceptance matrix 808 ⟩ ≡

```
{
  const NSYID postdot_nsyid ⇐ Postdot_NSYID_of_YIM(yim_to_clean);
  if (postdot_nsyid ≥ 0) {
    int cil_ix;
    const CIL lhs_cil ⇐ LHS_CIL_of_NSYID(postdot_nsyid);
    const int cil_count ⇐ Count_of_CIL(lhs_cil);
    for (cil_ix ⇐ 0; cil_ix < cil_count; cil_ix++) {
      const IRLID irlid ⇐ Item_of_CIL(lhs_cil, cil_ix);
      const int predicted_yim_ix ⇐ prediction_by_irl[irlid];
      const YIM predicted_yim ⇐ yims_to_clean[predicted_yim_ix];
      if (YIM_is_Rejected(predicted_yim)) continue;
      matrix_bit_set(acceptance_matrix, yim_to_clean_ix, predicted_yim_ix);
    }
  }
}
```

This code is used in section 807.

809. Mark YIM's not active if not scanned. If scanned, we can make a preliminary determination whether it is accepted based on the absence direct rejection and the presence of at least one unrejected token link. (A scanned YIM may have fusion links.) If this preliminary determination indicates that the scanned YIM is active, we mark it that way.

810. We need the preliminary indication, because when we compute the accepted YIM's from the transition closure of acceptances, we need a set of YIM's as a starting point. In Earley set 0, the initial YIM is the starting point, but in all later sets, the scanned YIM's are the starting points. We know that every unrejected YIM will trace back, in its YS, to either the initial YIM or an unrejected token SRCL in an unrejected scanned YIM.

811. A scanned YIM may have only rejected token SRCL's, but an accepted fusion SRCL. In effect, after the rejections, it is now a purely fusion YIM. We do not use such a now-purely-fusion, no-longer-scanned YIM as a starting point. We know this is safe, since in order to be accepted, every YIM must trace back to an unrejected YIM with unrejected token SRCL's, or to the initial YIM.

812. If not rejected, scan SRCL's. For each SRCL, reject if predecessor or cause if rejected; otherwise, record as a dependency on cause. Add dependencies to acceptance matrix. If any dependency was recorded, also add any direct predictions of un-rejected YIM's.

813. For every scanned or initial YIM in transitive closure, mark the to-YIM's of the dependency active. Mark all others rejected.

⟨ Mark accepted YIM's 813 ⟩ ≡

```
{
  int cause_yim_ix;
  for (cause_yim_ix ⇐ 0; cause_yim_ix < yim_to_clean_count; cause_yim_ix++) {
    const YIM cause_yim ⇐ yims_to_clean[cause_yim_ix];
    /* We only need look at the indirect effects of initial and scanned YIM's, because
         they are the indirect cause of all other YIM's in the YS. */
    if (¬YIM_is_Initial(cause_yim) ∧ ¬YIM_was_Scanned(cause_yim)) break;
    /* an indirect cause YIM may have been directly rejected, if which cause we do
         not use it, but keep looking for other indirect causes. */
    if (YIM_is_Rejected(cause_yim)) continue;
    {
      const Bit_Vector bv_yims_to_accept ⇐ matrix_row(acceptance_matrix,
          cause_yim_ix);
      int min, max, start;
      for (start ⇐ 0; bv_scan(bv_yims_to_accept, start, &min, &max);
            start ⇐ max + 2) {
        int yim_to_accept_ix;
        for (yim_to_accept_ix ⇐ min; yim_to_accept_ix ≤ max;
              yim_to_accept_ix++) {
          const YIM yim_to_accept ⇐ yims_to_clean[yim_to_accept_ix];
          YIM_is_Active(yim_to_accept) ⇐ 1;
        }
      }
    }
  }
}
```

This code is used in section 805.

814. This pass is probably not necessary, because I should be checking the active boolean from here on. But it restores the "consistent" state where a YIM is either rejected or accepted.

⟨ Mark un-accepted YIM's rejected 814 ⟩ ≡

```
{
  int yim_ix;
  for (yim_ix ⇐ 0; yim_ix < yim_to_clean_count; yim_ix++) {
    const YIM yim ⇐ yims_to_clean[yim_ix];
    if (¬YIM_is_Active(yim)) continue;
    YIM_is_Rejected(yim) ⇐ 1;
  }
```

```
}
```

This code is used in section 805.

815. To Do: Deferred while we are only dealing with YS 0.

816. We now have a full census of accepted and rejected YIM's. Use this to go back over SRCL's. These will all be resolveable one way or the other.

⟨ Mark accepted SRCL's 816 ⟩ ≡

```
{ }
```

This code is used in section 805.

817. Mark LIM's as accepted or rejected, based on their predecessors and trailhead YIM's.

⟨ Mark rejected LIM's 817 ⟩ ≡

```
{
  int postdot_sym_ix;
  const int postdot_sym_count ⟸ Postdot_SYM_Count_of_YS(ys_to_clean);
  const PIM *postdot_array ⟸ ys_to_clean→t_postdot_ary;
    /* For every postdot symbol */
  for (postdot_sym_ix ⟸ 0; postdot_sym_ix < postdot_sym_count;
      postdot_sym_ix++) {
    /* If there is a LIM, there will be only one, and it will be the first PIM. */
    const PIM first_pim ⟸ postdot_array[postdot_sym_ix];
    if (PIM_is_LIM(first_pim)) {
      const LIM lim ⟸ LIM_of_PIM(first_pim);
    /* Reject LIM by default */
      LIM_is_Rejected(lim) ⟸ 1;
      LIM_is_Active(lim) ⟸ 0;
    /* Reject, because the base-to YIM is not active */
      if (¬YIM_is_Active(Trailhead_YIM_of_LIM(lim))) continue;
      {
        const LIM predecessor_lim ⟸ Predecessor_LIM_of_LIM(lim);
    /* Reject, because the predecessor LIM exists and is not active */
        if (predecessor_lim ∧ ¬LIM_is_Active(predecessor_lim)) continue;
      }
    /* No reason found to reject, so accept this LIM */
      LIM_is_Rejected(lim) ⟸ 0;
      LIM_is_Active(lim) ⟸ 1;
    }
  }
}
```

This code is used in section 805.

818. For all pending alternatives, determine if they have unrejected predecessors. If not, remove them from the stack. Readjust furthest earleme. Note that moving the furthest earleme may change the parse to exhausted state.

⟨ Clean pending alternatives 818 ⟩ ≡

```
{
  int old_alt_ix;
  int no_of_alternatives ⇐ MARPA_DSTACK_LENGTH(r→t_alternatives);
    /* Increment old_alt_ix until it is one past the initial run of accept-able
       alternatives. If there were none, this leaves old_alt_ix at 0. If all alternatives
       were acceptable, this leaves old_alt_ix at no_of_alternatives. */
  for (old_alt_ix ⇐ 0; old_alt_ix < no_of_alternatives; old_alt_ix++) {
    const ALT alternative ⇐ MARPA_DSTACK_INDEX(r→t_alternatives,
        ALT_Object, old_alt_ix);
    if (¬alternative_is_acceptable(alternative)) break;
  }
    /* If we found an un-acceptable alternative, we need to adjust the alterntives
         stack. First we shorten the alternatives stack, copying acceptable alternatives
         to newly emptied slots in the stack until there are no gaps left. */
  if (old_alt_ix < no_of_alternatives) {
    /* empty_alt_ix is the empty slot, into which the next acceptable alternative
         should be copied. */
    int empty_alt_ix ⇐ old_alt_ix;
    for (old_alt_ix++; old_alt_ix < no_of_alternatives; old_alt_ix++) {
      const ALT alternative ⇐ MARPA_DSTACK_INDEX(r→t_alternatives,
          ALT_Object, old_alt_ix);
      if (¬alternative_is_acceptable(alternative)) continue;
      *MARPA_DSTACK_INDEX(r→t_alternatives, ALT_Object,
          empty_alt_ix) ⇐ *alternative;
      empty_alt_ix++;
    }
    /* empty_alt_ix points to the first available slot, so it is now the same as the new
          stack length */
    MARPA_DSTACK_COUNT_SET(r→t_alternatives, empty_alt_ix);
    if (empty_alt_ix) {
      Furthest_Earleme_of_R(r) ⇐ Earleme_of_YS(current_ys);
    }
    else {
      const ALT furthest_alternative ⇐
          MARPA_DSTACK_INDEX(r→t_alternatives, ALT_Object, 0);
      Furthest_Earleme_of_R(r) ⇐ End_Earleme_of_ALT(furthest_alternative);
    }
  }
```

```
}
```

This code is used in section 802.

819. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int alternative_is_acceptable(ALT alternative)
{
  PIM pim;
  const NSYID token_symbol_id ⟸ NSYID_of_ALT(alternative);
  const YS start_ys ⟸ Start_YS_of_ALT(alternative);
  for (pim ⟸ First_PIM_of_YS_by_NSYID(start_ys, token_symbol_id); pim;
        pim ⟸ Next_PIM_of_PIM(pim)) {
    YIM predecessor_yim ⟸ YIM_of_PIM(pim);
    /* If the trailhead PIM is non-active, the LIM will not be active, so we don't
        bother looking at the LIM. Instead we will wait for the source, which will be
        next in the list of PIM's */
    if (¬predecessor_yim) continue;    /* We have an active predecessor, so this
          alternative is OK. Move on to look at the next alterntive */
    if (YIM_is_Active(predecessor_yim)) return 1;
  }
  return 0;
}
```

820. ⟨ Clean expected terminals 820 ⟩ ≡

```
{ }
```

This code is used in section 802.

821. Recognizer zero-width assertion code.

⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_zwa_default_set(Marpa_Recognizer r, Marpa_Assertion_ID zwaid, int
        default_value)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  ZWA zwa;
  int old_default_value;
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if zwaid is malformed 1243 ⟩
  ⟨ Fail if zwaid does not exist 1242 ⟩
  if (_MARPA_UNLIKELY(default_value < 0 ∨ default_value > 1)) {
    MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
    return failure_indicator;
  }
  zwa ⟸ RZWA_by_ID(zwaid);
  old_default_value ⟸ Default_Value_of_ZWA(zwa);
  Default_Value_of_ZWA(zwa) ⟸ default_value ? 1 : 0;
  return old_default_value;
}
```

822. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_r_zwa_default(Marpa_Recognizer r, Marpa_Assertion_ID zwaid)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  ZWA zwa;
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if zwaid is malformed 1243 ⟩
  ⟨ Fail if zwaid does not exist 1242 ⟩
  zwa ⟸ RZWA_by_ID(zwaid);
  return Default_Value_of_ZWA(zwa);
}
```

823. Progress report code.

⟨ Private typedefs 49 ⟩ +≡
```
typedef struct marpa_progress_item *PROGRESS;
```

824. ⟨ Widely aligned recognizer elements 558 ⟩ +≡
```
const struct marpa_progress_item *t_current_report_item;
MARPA_AVL_TRAV t_progress_report_traverser;
```

825. ⟨ Initialize recognizer elements 554 ⟩ +≡
```
r→t_current_report_item ⟸ &progress_report_not_ready;
r→t_progress_report_traverser ⟸ Λ;
```

826. ⟨ Clear progress report in r 826 ⟩ ≡
```
r→t_current_report_item ⟸ &progress_report_not_ready;
if (r→t_progress_report_traverser) {
  _marpa_avl_destroy(MARPA_TREE_OF_AVL_TRAV(r→t_progress_report_traverser));
}
r→t_progress_report_traverser ⟸ Λ;
```
This code is used in sections 827, 832, and 836.

827. ⟨ Destroy recognizer elements 561 ⟩ +≡
```
⟨ Clear progress report in r 826 ⟩;
```

828. ⟨ Public structures 44 ⟩ +≡
```
struct marpa_progress_item {
  Marpa_Rule_ID t_rule_id;
  int t_position;
  int t_origin;
};
```

829. A dummy progress report item to allow the macros to produce error reports without having to use a ternary, and getting into issues of evaluation the argument twice.

⟨ Global constant variables 40 ⟩ +≡
```
static const struct marpa_progress_item progress_report_not_ready ⟸ {−2, −2, −2};
```

830.
```
#define RULEID_of_PROGRESS(report) ((report)→t_rule_id)
#define Position_of_PROGRESS(report) ((report)→t_position)
#define Origin_of_PROGRESS(report) ((report)→t_origin)
```

831. ⟨ Function definitions 41 ⟩ +≡
```
PRIVATE_NOT_INLINE int report_item_cmp(const void *ap, const void *bp, void
        *param UNUSED)
{
  const struct marpa_progress_item *const report_a ⟸ ap;
  const struct marpa_progress_item *const report_b ⟸ bp;
```

```
    if (Position_of_PROGRESS(report_a) > Position_of_PROGRESS(report_b))
      return 1;
    if (Position_of_PROGRESS(report_a) < Position_of_PROGRESS(report_b))
      return −1;
    if (RULEID_of_PROGRESS(report_a) > RULEID_of_PROGRESS(report_b)) return 1;
    if (RULEID_of_PROGRESS(report_a) < RULEID_of_PROGRESS(report_b)) return −1;
    if (Origin_of_PROGRESS(report_a) > Origin_of_PROGRESS(report_b)) return 1;
    if (Origin_of_PROGRESS(report_a) < Origin_of_PROGRESS(report_b)) return −1;
    return 0;
  }
```

832. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_r_progress_report_start(Marpa_Recognizer r, Marpa_Earley_Set_ID
          set_id)
  {
    ⟨ Return −2 on failure 1229 ⟩
    YS earley_set;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if recognizer not started 1246 ⟩
    if (set_id < 0) {
      MARPA_ERROR(MARPA_ERR_INVALID_LOCATION);
      return failure_indicator;
    }
    r_update_earley_sets(r);
    if (¬YS_Ord_is_Valid(r, set_id)) {
      MARPA_ERROR(MARPA_ERR_NO_EARLEY_SET_AT_LOCATION);
      return failure_indicator;
    }
    earley_set ⇐ YS_of_R_by_Ord(r, set_id);
    MARPA_OFF_DEBUG3("At␣%s,␣starting␣progress␣report␣Earley␣set␣%ld",
        STRLOC, (long) set_id);
    ⟨ Clear progress report in r 826 ⟩
    {
      const MARPA_AVL_TREE report_tree ⇐
          _marpa_avl_create(report_item_cmp, Λ);
      const YIM *const earley_items ⇐ YIMs_of_YS(earley_set);
      const int earley_item_count ⇐ YIM_Count_of_YS(earley_set);
      int earley_item_id;
      for (earley_item_id ⇐ 0; earley_item_id < earley_item_count;
            earley_item_id++) {
        const YIM earley_item ⇐ earley_items[earley_item_id];
        if (¬YIM_is_Active(earley_item)) continue;
        ⟨ Do the progress report for earley_item 834 ⟩
```

```
      }
      r→t_progress_report_traverser ⇐ _marpa_avl_t_init(report_tree);
      return (int) marpa_avl_count(report_tree);
    }
  }
```

833. Start the progress report again.

```
⟨ Function definitions 41 ⟩ +≡
  int marpa_r_progress_report_reset(Marpa_Recognizer r)
  {
    ⟨ Return −2 on failure 1229 ⟩
    MARPA_AVL_TRAV traverser ⇐ r→t_progress_report_traverser;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if recognizer not started 1246 ⟩
    ⟨ Fail if no traverser 838 ⟩
    _marpa_avl_t_reset(traverser);
    return 1;
  }
```

834. Caller ensures this YIM is active.

```
⟨ Do the progress report for earley_item 834 ⟩ ≡
  {
    SRCL leo_source_link ⇐ Λ;
    MARPA_OFF_DEBUG2("At␣%s,␣Do␣the␣progress␣report", STRLOC);
    progress_report_items_insert(report_tree, AHM_of_YIM(earley_item),
        earley_item);
    for (leo_source_link ⇐ First_Leo_SRCL_of_YIM(earley_item); leo_source_link;
          leo_source_link ⇐ Next_SRCL_of_SRCL(leo_source_link)) {
      LIM leo_item;
      MARPA_OFF_DEBUG3("At␣%s,␣Leo␣source␣link␣%p", STRLOC, leo_source_link);
      if (¬SRCL_is_Active(leo_source_link)) continue;
      MARPA_OFF_DEBUG3("At␣%s,␣active␣Leo␣source␣link␣%p", STRLOC,
          leo_source_link);
      /* If the SRCL at the Leo summit is active, then the whole path is active. */
      for (leo_item ⇐ LIM_of_SRCL(leo_source_link); leo_item;
            leo_item ⇐ Predecessor_LIM_of_LIM(leo_item)) {
        const YIM trailhead_yim ⇐ Trailhead_YIM_of_LIM(leo_item);
        const AHM trailhead_ahm ⇐ Trailhead_AHM_of_LIM(leo_item);
        progress_report_items_insert(report_tree, trailhead_ahm, trailhead_yim);
      }
      MARPA_OFF_DEBUG3("At␣%s,␣finished␣Leo␣source␣link␣%p", STRLOC,
          leo_source_link);
```

```
    }
  }
```

This code is used in section 832.

835. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void progress_report_items_insert(MARPA_AVL_TREE
        report_tree, AHM report_ahm, YIM origin_yim)
{
  const XRL source_xrl ⇐ XRL_of_AHM(report_ahm);
  MARPA_OFF_DEBUG5("%s␣Calling␣progress_report_items_insert(%p,␣%p,␣%p)",
      STRLOC, report_tree, report_ahm, origin_yim);
  if (¬source_xrl) return;
    /* If LHS is a brick symbol, we are done – insert the report item and return */
  if (¬IRL_has_Virtual_LHS(IRL_of_YIM(origin_yim))) {
    int xrl_position ⇐ XRL_Position_of_AHM(report_ahm);
    int origin_of_xrl ⇐ Origin_Ord_of_YIM(origin_yim);
    XRLID xrl_id ⇐ ID_of_XRL(source_xrl);
    PROGRESS new_report_item ⇐
        marpa_obs_new(MARPA_AVL_OBSTACK(report_tree),
        struct marpa_progress_item, 1);
    MARPA_OFF_DEBUG2("%s,␣===␣Adding␣report␣item␣===", STRLOC);
    MARPA_OFF_DEBUG3("%s,␣report␣irl␣=␣%d", STRLOC, IRLID_of_AHM(report_ahm));
    MARPA_OFF_DEBUG3("%s,␣report␣irl␣position␣=␣%d", STRLOC,
        Position_of_AHM(report_ahm));
    MARPA_OFF_DEBUG3("%s,␣xrl␣=␣%d", STRLOC, ID_of_XRL(source_xrl));
    MARPA_OFF_DEBUG3("%s,␣xrl␣dot␣=␣%d", STRLOC,
        XRL_Position_of_AHM(report_ahm));
    MARPA_OFF_DEBUG3("%s,␣origin␣ord␣=␣%d", STRLOC,
        Origin_Ord_of_YIM(origin_yim));
    Position_of_PROGRESS(new_report_item) ⇐ xrl_position;
    Origin_of_PROGRESS(new_report_item) ⇐ origin_of_xrl;
    RULEID_of_PROGRESS(new_report_item) ⇐ xrl_id;
    _marpa_avl_insert(report_tree, new_report_item);
    return;
  }
    /* If here, LHS is a mortar symbol */

    /* We don't recurse on sequence rules – we only need to look at the top rules,
        which have brick LHS's */
  if (XRL_is_Sequence(source_xrl)) return;
    /* Look at the predecessor items for the origin of the XRL. At this point, only
        CHAF rules do this. Source rules and sequence rules were specifically excluded
        above. And BNF rules will also have a non-virtual LHS. */
```

```
    {
      const NSYID lhs_nsyid ⇐ LHS_NSYID_of_YIM(origin_yim);
      const YS origin_of_origin_ys ⇐ Origin_of_YIM(origin_yim);
      PIM pim ⇐ First_PIM_of_YS_by_NSYID(origin_of_origin_ys, lhs_nsyid);
      for ( ; pim; pim ⇐ Next_PIM_of_PIM(pim)) {
        const YIM predecessor ⇐ YIM_of_PIM(pim);
      /* Ignore PIM chains with Leo items in them. (Leo items will always be first.) */
        if (¬predecessor) return;
        if (YIM_is_Active(predecessor)) {
          progress_report_items_insert(report_tree, report_ahm, predecessor);
        }
      }
    }
  }
```

836. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_r_progress_report_finish(Marpa_Recognizer r)
  {
    const int success ⇐ 1;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack recognizer objects 560 ⟩
    const MARPA_AVL_TRAV traverser ⇐ r→t_progress_report_traverser;
    ⟨ Fail if recognizer not started 1246 ⟩
    ⟨ Fail if no traverser 838 ⟩
    ⟨ Clear progress report in r 826 ⟩
    return success;
  }
```

837. ⟨ Function definitions 41 ⟩ +≡

```
  Marpa_Rule_ID marpa_r_progress_item(Marpa_Recognizer r, int *position,
          Marpa_Earley_Set_ID *origin)
  {
    ⟨ Return −2 on failure 1229 ⟩
    PROGRESS report_item;
    MARPA_AVL_TRAV traverser;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if recognizer not started 1246 ⟩
    traverser ⇐ r→t_progress_report_traverser;
    if (_MARPA_UNLIKELY(¬position ∨ ¬origin)) {
      MARPA_ERROR(MARPA_ERR_POINTER_ARG_NULL);
      return failure_indicator;
    }
```

```
    ⟨ Fail if no traverser 838 ⟩
    report_item ⟸ _marpa_avl_t_next(traverser);
    if (¬report_item) {
      MARPA_ERROR(MARPA_ERR_PROGRESS_REPORT_EXHAUSTED);
      return −1;
    }
    *position ⟸ Position_of_PROGRESS(report_item);
    *origin ⟸ Origin_of_PROGRESS(report_item);
    return RULEID_of_PROGRESS(report_item);
  }
```

838. ⟨ Fail if no traverser 838 ⟩ ≡

```
  {
    if (¬traverser) {
      MARPA_ERROR(MARPA_ERR_PROGRESS_REPORT_NOT_STARTED);
      return failure_indicator;
    }
  }
```

This code is used in sections 833, 836, and 837.

839. Some notes on evaluation.

840. Sources of Leo path items. A Leo path consists of a series of Earley items:
- at the bottom, exactly one Leo base item;
- at the top, exactly one Leo completion item;
- in between, zero or more Leo path items.

841. Leo base items and Leo completion items can have a variety of non-Leo sources. Leo completion items can have multiple Leo sources, though no other source can have the same middle earleme as a Leo source.

842. When expanded, Leo path items can have multiple sources. However, the sources of a single Leo path item will result from the same Leo predecessor. As consequences:
- All the sources of an expanded Leo path item will have the same Earley item predecessor, the Leo base item of the Leo predecessor.
- All these sources will also have the same middle earleme and the same origin, both taken from the Earley item predecessor.
- If the cause is a token, the transition symbol will be the token symbol. Only one source may have a token cause.
- If the cause is a rule completion, the transition symbol will be the LHS of that rule. Several source may have rule completion causes, but the maximum number is limited by the number of rule's with the transition symbol on their LHS.
- The number of sources of a Leo path item is therefore limited by a constant that depends on the grammar.

843. To Do: Determine exactly when Leo path items may come from multiple souces.
- When can a Leo path item also be an item from a non-Leo source? The top item can, but can any others?
- In the case of LHS terminals, any item can be scanned.
- A top item on a path is **not** a transition over a Leo symbol, and so may have any number of predecessors, as long as any Leo sources have a unique middle Earley set.
- The bottom item does result does match a Leo transition, and so can only be matched one predecessor. But it itself may have many sources. It may, for example, be the top item of a Leo path for a different right recursion.

844. In the following, I refer to Leo path bases, and Leo path top items. It is assumed that these Earley items are active items in a consistent parse. Also, any SRCL's referred to are assumed to be active SRCL's in a consistent parse.

845. Also in the following:
- Origin(y_{YIM}) is the origin, or start, location of the YIM y_{YIM}.
- Symbol(*cause*) if the LHS symbol of the YIM's rule is *cause* is a YIM. Symbol(*cause*) is the token symbol if *cause* is a token.

846. Theorem: Consider a Leo path with a base b, which is the cause of a Leo SRCL in the Leo path top YIM, t. b will only be the base of that SRCL in that YIM.

847. Proof: Suppose it was the base of two different SRCL's. Since both SRCL's will have the same middle (the origin of b) and the same transition symbol (either the token symbol of b, or its LHS, call that sym), both will have the same Leo transition. SRCL must have a LIM at Origin(b) with transition symbol sym. By the construction of LIM's, there can be other predecessor for b at Origin(b). So b's Leo SRCL in t is the only SRCL in which it is the cause. **QED**

848. Note, in the above theorem, that while b must be unique to its SRCL, this is not true of Leo predecessors. A Leo predecessor may be in more than one SRCL, so long as the symbols of the cause's in those SRCL's are the same: sym. This means the number of SRCL's which can contain a given predecessor is a constant that depends on the grammar. (Specifically, it is the number of rules with sym on their LHS, plus one for a terminal.)

849. Theorem: Consider a item on a Leo path other than the top item. Call this item p_i. p_i must have an effect YIM, p_{i+1}. In other words, there must be an YIM above it on the Leo path.

850. Proof: Since we assumed that the top and bottom items are active items in a consistent parse, by the properties of Earley parsing we know that p_i has a predecessor, and an effect. **QED**

851. Theorem: Consider, p_i, a item on a Leo path other than the top item. All SRCL's containing p_i as a cause have the same predecessor.

852. Proof: Since p_i is on a Leo path, the transition over Symbol(p_i) from Origin(p_i) must be from a unique YIM. This YIM is Pred(p_i), the unique predecessor of p_i. **QED**

853. Theorem: Consider, p_i, a item on a Leo path other than the top item. Its effect, p_{i+1} is unique.

854. Proof: Consider multiple effect YIM's of p_i. Call two of these p_{i+1}, q_{i+1}. By a previous theorem, both have the same predecessor, Pred(p_i). Because p_{i+1} and q_{i+1} have the same predecessor and the same cause (p_i), we know that p_{i+1} and q_{i+1} also have the same origin, dotted rule and current earley set. If two YIM's have the same origin, dotted rule, and current earley set, they are identical. This shows that the effect YIM of the cause p_i is unique. **QED**

855. Ur-node (UR) code. Ur is a German word for "primordial", which is used a lot in academic writing to designate precursors — for example, scholars who believe that Shakespeare's *Hamlet* is based on another, now lost, play, call this play the ur-Hamlet. My ur-nodes are precursors of and-nodes and or-nodes.

⟨ Private incomplete structures 107 ⟩ +≡
```
struct s_ur_node_stack;
struct s_ur_node;
typedef struct s_ur_node_stack *URS;
typedef struct s_ur_node *UR;
typedef const struct s_ur_node *UR_Const;
```

856. To Do: It may make sense to reuse this stack for the alternatives. In that case some of these structures will need to be changed.

```
#define Prev_UR_of_UR(ur) ((ur)→t_prev)
#define Next_UR_of_UR(ur) ((ur)→t_next)
#define YIM_of_UR(ur) ((ur)→t_earley_item)
```

⟨ Private structures 48 ⟩ +≡
```
struct s_ur_node_stack {
  struct marpa_obstack *t_obs;
  UR t_base;
  UR t_top;
};
```

857. ⟨ Private structures 48 ⟩ +≡
```
struct s_ur_node {
  UR t_prev;
  UR t_next;
  YIM t_earley_item;
};
typedef struct s_ur_node UR_Object;
```

858.
```
#define URS_of_R(r) (&(r)→t_ur_node_stack)
```

⟨ Widely aligned recognizer elements 558 ⟩ +≡
```
struct s_ur_node_stack t_ur_node_stack;
```

859. To Do: The lifetime of this stack should be reexamined once its uses are settled.

⟨ Initialize recognizer elements 554 ⟩ +≡
```
ur_node_stack_init(URS_of_R(r));
```

860. ⟨ Destroy recognizer elements 561 ⟩ +≡
```
ur_node_stack_destroy(URS_of_R(r));
```

861. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void ur_node_stack_init(URS stack)
{
  stack→t_obs ⇐ marpa_obs_init;
  stack→t_base ⇐ ur_node_new(stack, 0);
  ur_node_stack_reset(stack);
}
```

862. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void ur_node_stack_reset(URS stack)
{
  stack→t_top ⇐ stack→t_base;
}
```

863. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void ur_node_stack_destroy(URS stack)
{
  if (stack→t_base) marpa_obs_free(stack→t_obs);
  stack→t_base ⇐ Λ;
}
```

864. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE UR ur_node_new(URS stack, UR prev)
{
  UR new_ur_node;
  new_ur_node ⇐ marpa_obs_new(stack→t_obs, UR_Object, 1);
  Next_UR_of_UR(new_ur_node) ⇐ 0;
  Prev_UR_of_UR(new_ur_node) ⇐ prev;
  return new_ur_node;
}
```

865. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void ur_node_push(URS stack, YIM earley_item)
{
  UR old_top ⇐ stack→t_top;
  UR new_top ⇐ Next_UR_of_UR(old_top);
  YIM_of_UR(old_top) ⇐ earley_item;
  if (¬new_top) {
    new_top ⇐ ur_node_new(stack, old_top);
    Next_UR_of_UR(old_top) ⇐ new_top;
  }
  stack→t_top ⇐ new_top;
}
```

866. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE UR ur_node_pop(URS stack)
{
  UR new_top ⇐ Prev_UR_of_UR(stack→t_top);
  if (¬new_top) return Λ;
  stack→t_top ⇐ new_top;
  return new_top;
}
```

867. **To Do**: No predictions are used in creating or-nodes. Most (all?) are eliminating in creating the PSI data. But I think predictions are tested for, when creating or-nodes, which should not be necessary. I need to decide where to look at this.

⟨ Populate the PSI data 867 ⟩ ≡

```
{
  UR_Const ur_node;
  const URS ur_node_stack ⇐ URS_of_R(r);
  ur_node_stack_reset(ur_node_stack);
    /* start_yim is never rejected */
  push_ur_if_new(per_ys_data, ur_node_stack, start_yim);
  while ((ur_node ⇐ ur_node_pop(ur_node_stack))) {
      /* rejected YIM's are never put on the ur-node stack */
    const YIM parent_earley_item ⇐ YIM_of_UR(ur_node);
    MARPA_ASSERT(¬YIM_was_Predicted(parent_earley_item))
    ⟨ Push child Earley items from token sources 870 ⟩
    ⟨ Push child Earley items from completion sources 872 ⟩
    ⟨ Push child Earley items from Leo sources 873 ⟩
  }
}
```

This code is used in section 942.

868. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void push_ur_if_new(struct s_bocage_setup_per_ys *per_ys_data, URS
        ur_node_stack, YIM yim)
{
  if (¬psi_test_and_set(per_ys_data, yim)) {
    ur_node_push(ur_node_stack, yim);
  }
}
```

869. The PSI is a container of data that is per Earley-set, and within that, per Earley item. (In the past, it has also been called the PSIA.) This function ensures that the appropriate PSI boolean is set. It returns that boolean's value **prior** to the call.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int psi_test_and_set(struct s_bocage_setup_per_ys *per_ys_data, YIM
        earley_item)
{
  const YSID set_ordinal ⇐ YS_Ord_of_YIM(earley_item);
  const int item_ordinal ⇐ Ord_of_YIM(earley_item);
  const OR previous_or_node ⇐ OR_by_PSI(per_ys_data, set_ordinal,
      item_ordinal);
  if (¬previous_or_node) {
    OR_by_PSI(per_ys_data, set_ordinal, item_ordinal) ⇐ dummy_or_node;
    return 0;
  }
  return 1;
}
```

870. ⟨ Push child Earley items from token sources 870 ⟩ ≡

```
{
  SRCL source_link;
  for (source_link ⇐ First_Token_SRCL_of_YIM(parent_earley_item);
        source_link; source_link ⇐ Next_SRCL_of_SRCL(source_link)) {
    YIM predecessor_earley_item;
    if (¬SRCL_is_Active(source_link)) continue;
    predecessor_earley_item ⇐ Predecessor_of_SRCL(source_link);
    if (¬predecessor_earley_item) continue;
    if (YIM_was_Predicted(predecessor_earley_item)) {
      Set_boolean_in_PSI_for_initial_nulls(per_ys_data,
          predecessor_earley_item);
      continue;
    }
    push_ur_if_new(per_ys_data, ur_node_stack, predecessor_earley_item);
  }
}
```

This code is used in section 867.

871. If there are initial nulls, set a boolean in the PSI so that I will know to create the chain of or-nodes for them. We don't need to stack the prediction, because it can have no other descendants.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void Set_boolean_in_PSI_for_initial_nulls(struct s_bocage_setup_per_ys
        *per_ys_data, YIM yim)
{
```

```
    const AHM ahm ⟸ AHM_of_YIM(yim);
    if (Null_Count_of_AHM(ahm)) psi_test_and_set(per_ys_data, (yim));
  }
```

872. ⟨ Push child Earley items from completion sources 872 ⟩ ≡

```
  {
    SRCL source_link;
    for (source_link ⟸ First_Completion_SRCL_of_YIM(parent_earley_item);
          source_link; source_link ⟸ Next_SRCL_of_SRCL(source_link)) {
      YIM predecessor_earley_item;
      YIM cause_earley_item;
      if (¬SRCL_is_Active(source_link)) continue;
      cause_earley_item ⟸ Cause_of_SRCL(source_link);
      push_ur_if_new(per_ys_data, ur_node_stack, cause_earley_item);
      predecessor_earley_item ⟸ Predecessor_of_SRCL(source_link);
      if (¬predecessor_earley_item) continue;
      if (YIM_was_Predicted(predecessor_earley_item)) {
        Set_boolean_in_PSI_for_initial_nulls(per_ys_data,
            predecessor_earley_item);
        continue;
      }
      push_ur_if_new(per_ys_data, ur_node_stack, predecessor_earley_item);
    }
  }
```

This code is used in section 867.

873. ⟨ Push child Earley items from Leo sources 873 ⟩ ≡

```
  {
    SRCL source_link;
      /* For every Leo source link */
    for (source_link ⟸ First_Leo_SRCL_of_YIM(parent_earley_item); source_link;
          source_link ⟸ Next_SRCL_of_SRCL(source_link)) {
      LIM leo_predecessor;
      YIM cause_earley_item;
      /* Ignore if not active – if it is active, then the whole chain must be */
      if (¬SRCL_is_Active(source_link)) continue;
      cause_earley_item ⟸ Cause_of_SRCL(source_link);
      push_ur_if_new(per_ys_data, ur_node_stack, cause_earley_item);
      for (leo_predecessor ⟸ LIM_of_SRCL(source_link); leo_predecessor;
      /* Follow the predecessors chain back */
      leo_predecessor ⟸ Predecessor_LIM_of_LIM(leo_predecessor)) {
        const YIM leo_base_yim ⟸ Trailhead_YIM_of_LIM(leo_predecessor);
        if (YIM_was_Predicted(leo_base_yim)) {
```

```
          Set_boolean_in_PSI_for_initial_nulls(per_ys_data, leo_base_yim);
        }
        else {
          push_ur_if_new(per_ys_data, ur_node_stack, leo_base_yim);
        }
      }
    }
  }
```

This code is used in section 867.

874. Or-node (OR) code. The or-nodes are part of the parse bocage and are similar to the or-nodes of a standard parse forest. Unlike a parse forest, a parse bocage can contain cycles.

⟨ Public typedefs 91 ⟩ +≡
  typedef int Marpa_Or_Node_ID;

875. ⟨ Private typedefs 49 ⟩ +≡
  typedef Marpa_Or_Node_ID ORID;

876. ⟨ Private incomplete structures 107 ⟩ +≡
  union u_or_node;
  *typedef union u_or_node *OR*;

877. The type is contained in same word as the position is for final or-nodes.

```
#define DUMMY_OR_NODE −1
#define MAX_TOKEN_OR_NODE −2
#define VALUED_TOKEN_OR_NODE −2
#define NULLING_TOKEN_OR_NODE −3
#define UNVALUED_TOKEN_OR_NODE −4
#define OR_is_Token(or) (Type_of_OR(or) ≤ MAX_TOKEN_OR_NODE)
#define Position_of_OR(or) ((or)→t_final.t_position)
#define Type_of_OR(or) ((or)→t_final.t_position)
#define IRL_of_OR(or) ((or)→t_final.t_irl)
#define IRLID_of_OR(or) ID_of_IRL(IRL_of_OR(or))
#define Origin_Ord_of_OR(or) ((or)→t_final.t_start_set_ordinal)
#define ID_of_OR(or) ((or)→t_final.t_id)
#define YS_Ord_of_OR(or) ((or)→t_draft.t_end_set_ordinal)
#define Length_of_OR(or) (YS_Ord_of_OR(or) − Origin_Ord_of_OR(or))
#define DANDs_of_OR(or) ((or)→t_draft.t_draft_and_node)
#define First_ANDID_of_OR(or) ((or)→t_final.t_first_and_node_id)
#define AND_Count_of_OR(or) ((or)→t_final.t_and_node_count)
```

878. C89 guarantees that common initial sequences may be accessed via different members of a union.

⟨ Or-node common initial sequence 878 ⟩ ≡
  int t_position;

This code is used in sections 879 and 882.

879. ⟨ Or-node less common initial sequence 879 ⟩ ≡
  ⟨ Or-node common initial sequence 878 ⟩
  int t_end_set_ordinal;
  int t_start_set_ordinal;
  ORID t_id;
  IRL t_irl;

This code is used in sections 880 and 881.

880. ⟨ Private structures 48 ⟩ +≡

```
struct s_draft_or_node {
  ⟨ Or-node less common initial sequence 879 ⟩
  DAND t_draft_and_node;
};
```

881. ⟨ Private structures 48 ⟩ +≡

```
struct s_final_or_node {
  ⟨ Or-node less common initial sequence 879 ⟩
  int t_first_and_node_id;
  int t_and_node_count;
};
```

882. ⟨ Private structures 48 ⟩ +≡

```
struct s_valued_token_or_node {
  ⟨ Or-node common initial sequence 878 ⟩
  NSYID t_nsyid;
  int t_value;
};
```

883.

```
#define NSYID_of_OR(or) ((or)→t_token.t_nsyid)
#define Value_of_OR(or) ((or)→t_token.t_value)
```

⟨ Private structures 48 ⟩ +≡

```
union u_or_node {
  struct s_draft_or_node t_draft;
  struct s_final_or_node t_final;
  struct s_valued_token_or_node t_token;
};
typedef union u_or_node OR_Object;
```

884. ⟨ Global constant variables 40 ⟩ +≡

```
static const int dummy_or_node_type ⇐ DUMMY_OR_NODE;
static const OR dummy_or_node ⇐ (OR) &dummy_or_node_type;
```

885.

```
#define ORs_of_B(b) ((b)→t_or_nodes)
#define OR_of_B_by_ID(b, id) (ORs_of_B(b)[(id)])
#define OR_Count_of_B(b) ((b)→t_or_node_count)
#define OR_Capacity_of_B(b) ((b)→t_or_node_capacity)
#define ANDs_of_B(b) ((b)→t_and_nodes)
#define AND_Count_of_B(b) ((b)→t_and_node_count)
#define Top_ORID_of_B(b) ((b)→t_top_or_node_id)
```

⟨ Widely aligned bocage elements 885 ⟩ ≡

```
OR *t_or_nodes;
```

```
AND t_and_nodes;
```

See also sections 889, 940, and 943.
This code is used in section 937.

886. ⟨ Int aligned bocage elements 886 ⟩ ≡

```
int t_or_node_capacity;
int t_or_node_count;
int t_and_node_count;
ORID t_top_or_node_id;
```

See also sections 957 and 961.
This code is used in section 937.

887. ⟨ Initialize bocage elements 887 ⟩ ≡

```
ORs_of_B(b) ⟸ Λ;
OR_Count_of_B(b) ⟸ 0;
ANDs_of_B(b) ⟸ Λ;
AND_Count_of_B(b) ⟸ 0;
Top_ORID_of_B(b) ⟸ −1;
```

See also sections 890, 944, 958, 962, and 969.
This code is used in section 942.

888. ⟨ Destroy bocage elements, main phase 888 ⟩ ≡

```
{
  OR *or_nodes ⟸ ORs_of_B(b);
  AND and_nodes ⟸ ANDs_of_B(b);
  grammar_unref(G_of_B(b));
  my_free(or_nodes);
  ORs_of_B(b) ⟸ Λ;
  my_free(and_nodes);
  ANDs_of_B(b) ⟸ Λ;
}
```

This code is used in section 965.

889. #define G_of_B(b) ((b)→t_grammar)

⟨ Widely aligned bocage elements 885 ⟩ +≡

```
GRAMMAR t_grammar;
```

890. ⟨ Initialize bocage elements 887 ⟩ +≡

```
{
  G_of_B(b) ⟸ G_of_R(r);
  grammar_ref(g);
}
```

891. Create the or-nodes.

⟨ Create the or-nodes for all earley sets 891 ⟩ ≡

```
{
  PSAR_Object or_per_ys_arena;
  const PSAR or_psar ⟸ &or_per_ys_arena;
  int work_earley_set_ordinal;
  OR_Capacity_of_B(b) ⟸ count_of_earley_items_in_parse;
  ORs_of_B(b) ⟸ marpa_new(OR, OR_Capacity_of_B(b));
  psar_init(or_psar, SYMI_Count_of_G(g));
  for (work_earley_set_ordinal ⟸ 0; work_earley_set_ordinal <
        earley_set_count_of_r; work_earley_set_ordinal++) {
    const YS_Const earley_set ⟸ YS_of_R_by_Ord(r, work_earley_set_ordinal);
    YIM *const yims_of_ys ⟸ YIMs_of_YS(earley_set);
    const int item_count ⟸ YIM_Count_of_YS(earley_set);
    PSL this_earley_set_psl;
    psar_dealloc(or_psar);
    this_earley_set_psl ⟸ psl_claim_by_es(or_psar, per_ys_data,
        work_earley_set_ordinal);
    ⟨ Create the or-nodes for work_earley_set_ordinal 892 ⟩
    ⟨ Create draft and-nodes for work_earley_set_ordinal 908 ⟩
  }
  psar_destroy(or_psar);
  ORs_of_B(b) ⟸ marpa_renew(OR, ORs_of_B(b), OR_Count_of_B(b));
}
```

This code is used in section 942.

892. ⟨ Create the or-nodes for work_earley_set_ordinal 892 ⟩ ≡

```
{
  int item_ordinal;
  for (item_ordinal ⟸ 0; item_ordinal < item_count; item_ordinal++) {
    if (OR_by_PSI(per_ys_data, work_earley_set_ordinal, item_ordinal)) {
      const YIM work_earley_item ⟸ yims_of_ys[item_ordinal];
      {
        ⟨ Create the or-nodes for work_earley_item 893 ⟩
      }
    }
  }
}
```

This code is used in section 891.

893. ⟨ Create the or-nodes for `work_earley_item` 893 ⟩ ≡

```
{
  AHM ahm ⇐ AHM_of_YIM(work_earley_item);
  const int working_ys_ordinal ⇐ YS_Ord_of_YIM(work_earley_item);
  const int working_yim_ordinal ⇐ Ord_of_YIM(work_earley_item);
  const int work_origin_ordinal ⇐ Ord_of_YS(Origin_of_YIM(work_earley_item));
  SYMI ahm_symbol_instance;
  OR psi_or_node ⇐ Λ;
  ahm_symbol_instance ⇐ SYMI_of_AHM(ahm);
  {
    PSL or_psl ⇐ psl_claim_by_es(or_psar, per_ys_data, work_origin_ordinal);
    OR last_or_node ⇐ Λ;
    ⟨ Add main or-node 895 ⟩
    ⟨ Add nulling token or-nodes 898 ⟩
  }
    /* The following assertion is now not necessarily true. it is kept for documentation,
         but eventually should be removed */
  MARPA_OFF_ASSERT(psi_or_node)
    /* Replace the dummy or-node with the last one added */
  OR_by_PSI(per_ys_data, working_ys_ordinal,
      working_yim_ordinal) ⇐ psi_or_node;
  ⟨ Add Leo or-nodes for work_earley_item 899 ⟩
}
```

This code is used in section 892.

894. Non-Leo or-nodes.

895. Add the main or-node — the one that corresponds directly to this AHM. The exception are predicted AHM's. Or-nodes are not added for predicted AHM's.

⟨ Add main or-node 895 ⟩ ≡

```
{
  if (ahm_symbol_instance ≥ 0) {
    OR or_node;
    MARPA_ASSERT(ahm_symbol_instance < SYMI_Count_of_G(g))
    or_node ⇐ PSL_Datum(or_psl, ahm_symbol_instance);
    if (¬or_node ∨ YS_Ord_of_OR(or_node) ≠ work_earley_set_ordinal) {
      const IRL irl ⇐ IRL_of_AHM(ahm);
      or_node ⇐ last_or_node ⇐ or_node_new(b);
      PSL_Datum(or_psl, ahm_symbol_instance) ⇐ last_or_node;
      Origin_Ord_of_OR(or_node) ⇐ Origin_Ord_of_YIM(work_earley_item);
      YS_Ord_of_OR(or_node) ⇐ work_earley_set_ordinal;
      IRL_of_OR(or_node) ⇐ irl;
      Position_of_OR(or_node) ⇐ ahm_symbol_instance − SYMI_of_IRL(irl) + 1;
```

```
        }
        psi_or_node ⟸ or_node;
      }
    }
```

This code is used in section 893.

896. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE OR or_node_new(BOCAGE b)
  {
    const int or_node_id ⟸ OR_Count_of_B(b)++;
    const OR new_or_node ⟸ (OR) marpa_obs_new(OBS_of_B(b), OR_Object, 1);
    ID_of_OR(new_or_node) ⟸ or_node_id;
    DANDs_of_OR(new_or_node) ⟸ Λ;
    if (_MARPA_UNLIKELY(or_node_id ≥ OR_Capacity_of_B(b))) {
      OR_Capacity_of_B(b) *= 2;
      ORs_of_B(b) ⟸ marpa_renew(OR, ORs_of_B(b), OR_Capacity_of_B(b));
    }
    OR_of_B_by_ID(b, or_node_id) ⟸ new_or_node;
    return new_or_node;
  }
```

897. In the following logic, the order matters. The one added last in this logic, or in the logic for adding the main item, will be used as the or-node in the PSI.

898. In building the final or-node, the predecessor can be determined using the PSI for symbol_instance − 1. The exception is where there is no predecessor, and this is the case if Position_of_OR(or_node) ≡ 0.

⟨ Add nulling token or-nodes 898 ⟩ ≡

```
  {
    const int null_count ⟸ Null_Count_of_AHM(ahm);
    if (null_count > 0) {
      const IRL irl ⟸ IRL_of_AHM(ahm);
      const int symbol_instance_of_rule ⟸ SYMI_of_IRL(irl);
      const int first_null_symbol_instance ⟸ ahm_symbol_instance < 0 ?
          symbol_instance_of_rule : ahm_symbol_instance + 1;
      int i;
      for (i ⟸ 0; i < null_count; i++) {
        const int symbol_instance ⟸ first_null_symbol_instance + i;
        OR or_node ⟸ PSL_Datum(or_psl, symbol_instance);
        if (¬or_node ∨ YS_Ord_of_OR(or_node) ≠ work_earley_set_ordinal) {
          const int rhs_ix ⟸ symbol_instance − symbol_instance_of_rule;
          const OR predecessor ⟸ rhs_ix ? last_or_node : Λ;
          const OR cause ⟸ Nulling_OR_by_NSYID(RHSID_of_IRL(irl, rhs_ix));
```

```
          or_node ⟸ PSL_Datum(or_psl,
              symbol_instance) ⟸ last_or_node ⟸ or_node_new(b);
          Origin_Ord_of_OR(or_node) ⟸ work_origin_ordinal;
          YS_Ord_of_OR(or_node) ⟸ work_earley_set_ordinal;
          IRL_of_OR(or_node) ⟸ irl;
          Position_of_OR(or_node) ⟸ rhs_ix + 1;
          MARPA_ASSERT(Position_of_OR(or_node) ≤ 1 ∨ predecessor);
          draft_and_node_add(bocage_setup_obs, or_node, predecessor, cause);
        }
        psi_or_node ⟸ or_node;
      }
    }
  }
```

This code is used in section 893.

899. Leo or-nodes.

⟨ Add Leo or-nodes for `work_earley_item` 899 ⟩ ≡

```
  {
    SRCL source_link;
    for (source_link ⟸ First_Leo_SRCL_of_YIM(work_earley_item); source_link;
          source_link ⟸ Next_SRCL_of_SRCL(source_link)) {
      LIM leo_predecessor ⟸ LIM_of_SRCL(source_link);
      if (leo_predecessor) {
        ⟨ Add or-nodes for chain starting with leo_predecessor 900 ⟩
      }
    }
  }
```

This code is used in section 893.

900. The main loop in this code deliberately skips the first Leo predecessor. The successor of the first Leo predecessor is the base of the Leo path, which already exists, and therefore the first Leo predecessor is not expanded.

⟨ Add or-nodes for chain starting with `leo_predecessor` 900 ⟩ ≡

```
  {
    LIM this_leo_item ⟸ leo_predecessor;
    LIM previous_leo_item ⟸ this_leo_item;
    while ((this_leo_item ⟸ Predecessor_LIM_of_LIM(this_leo_item))) {
      const int ordinal_of_set_of_this_leo_item ⟸
          Ord_of_YS(YS_of_LIM(this_leo_item));
      const AHM path_ahm ⟸ Trailhead_AHM_of_LIM(previous_leo_item);
      const IRL path_irl ⟸ IRL_of_AHM(path_ahm);
      const int symbol_instance_of_path_ahm ⟸ SYMI_of_AHM(path_ahm);
      {
        OR last_or_node ⟸ Λ;
```

```
      ⟨ Add main Leo path or-node 901 ⟩
      ⟨ Add Leo path nulling token or-nodes 902 ⟩
    }
    previous_leo_item ⟸ this_leo_item;
  }
}
```

This code is used in section 899.

901. Adds the main Leo path or-node — the non-nulling or-node which corresponds to the Leo predecessor.

```
⟨ Add main Leo path or-node 901 ⟩ ≡
{
  {
    OR or_node;
    PSL leo_psl ⟸ psl_claim_by_es(or_psar, per_ys_data,
        ordinal_of_set_of_this_leo_item);
    or_node ⟸ PSL_Datum(leo_psl, symbol_instance_of_path_ahm);
    if (¬or_node ∨ YS_Ord_of_OR(or_node) ≠ work_earley_set_ordinal) {
      last_or_node ⟸ or_node_new(b);
      PSL_Datum(leo_psl,
          symbol_instance_of_path_ahm) ⟸ or_node ⟸ last_or_node;
      Origin_Ord_of_OR(or_node) ⟸ ordinal_of_set_of_this_leo_item;
      YS_Ord_of_OR(or_node) ⟸ work_earley_set_ordinal;
      IRL_of_OR(or_node) ⟸ path_irl;
      Position_of_OR(or_node) ⟸ symbol_instance_of_path_ahm −
          SYMI_of_IRL(path_irl) + 1;
    }
  }
}
```

This code is used in section 900.

902. In building the final or-node, the predecessor can be determined using the PSI for `symbol_instance` − 1. There will always be a predecessor, since these nulling or-nodes follow a completion.

```
⟨ Add Leo path nulling token or-nodes 902 ⟩ ≡
{
  int i;
  const int null_count ⟸ Null_Count_of_AHM(path_ahm);
  for (i ⟸ 1; i ≤ null_count; i++) {
    const int symbol_instance ⟸ symbol_instance_of_path_ahm + i;
    OR or_node ⟸ PSL_Datum(this_earley_set_psl, symbol_instance);
    MARPA_ASSERT(symbol_instance < SYMI_Count_of_G(g))
    if (¬or_node ∨ YS_Ord_of_OR(or_node) ≠ work_earley_set_ordinal) {
      const int rhs_ix ⟸ symbol_instance − SYMI_of_IRL(path_irl);
```

```
        MARPA_ASSERT(rhs_ix < Length_of_IRL(path_irl))
        const OR predecessor ⇐ rhs_ix ? last_or_node : Λ;
        const OR cause ⇐ Nulling_OR_by_NSYID(RHSID_of_IRL(path_irl, rhs_ix));
        MARPA_ASSERT(symbol_instance < Length_of_IRL(path_irl))
        MARPA_ASSERT(symbol_instance ≥ 0)
        or_node ⇐ last_or_node ⇐ or_node_new(b);
        PSL_Datum(this_earley_set_psl, symbol_instance) ⇐ or_node;
        Origin_Ord_of_OR(or_node) ⇐ ordinal_of_set_of_this_leo_item;
        YS_Ord_of_OR(or_node) ⇐ work_earley_set_ordinal;
        IRL_of_OR(or_node) ⇐ path_irl;
        Position_of_OR(or_node) ⇐ rhs_ix + 1;
        MARPA_ASSERT(Position_of_OR(or_node) ≤ 1 ∨ predecessor);
        draft_and_node_add(bocage_setup_obs, or_node, predecessor, cause);
      }
      MARPA_ASSERT(Position_of_OR(or_node) ≤ SYMI_of_IRL(path_irl) +
          Length_of_IRL(path_irl))
      MARPA_ASSERT(Position_of_OR(or_node) ≥ SYMI_of_IRL(path_irl))
    }
  }
```

This code is used in section 900.

903. Whole element ID (WHEID) code. The "whole elements" of the grammar are the symbols and the completed rules. **To Do**: **Restriction**: Note that this puts a limit on the number of symbols and internal rules in a grammar — their total must fit in an int.

```
#define WHEID_of_NSYID(nsyid) (irl_count + (nsyid))
#define WHEID_of_IRLID(irlid) (irlid)
#define WHEID_of_IRL(irl) WHEID_of_IRLID(ID_of_IRL(irl))
#define WHEID_of_OR(or)
        (wheid ⟸ OR_is_Token(or) ? WHEID_of_NSYID(NSYID_of_OR(or)) :
            WHEID_of_IRL(IRL_of_OR(or)))
```

⟨ Private typedefs 49 ⟩ +≡

```
typedef int WHEID;
```

904. Draft and-node (DAND) code. The draft and-nodes are used while the bocage is being built. Both draft and final and-nodes contain the predecessor and cause. Draft and-nodes need to be in a linked list, so they have a link to the next and-node.

```
⟨ Private incomplete structures 107 ⟩ +≡
  struct s_draft_and_node;
  typedef struct s_draft_and_node *DAND;
```

905.

```
#define Next_DAND_of_DAND(dand) ((dand)→t_next)
#define Predecessor_OR_of_DAND(dand) ((dand)→t_predecessor)
#define Cause_OR_of_DAND(dand) ((dand)→t_cause)
⟨ Private structures 48 ⟩ +≡
  struct s_draft_and_node {
    DAND t_next;
    OR t_predecessor;
    OR t_cause;
  };
  typedef struct s_draft_and_node DAND_Object;
```

906. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE DAND draft_and_node_new(struct marpa_obstack *obs, OR
          predecessor, OR cause)
  {
    DAND draft_and_node ⇐ marpa_obs_new(obs, DAND_Object, 1);
    Predecessor_OR_of_DAND(draft_and_node) ⇐ predecessor;
    Cause_OR_of_DAND(draft_and_node) ⇐ cause;
    MARPA_ASSERT(cause ≠ Λ);
    return draft_and_node;
  }
```

907. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void draft_and_node_add(struct marpa_obstack *obs, OR parent, OR
          predecessor, OR cause)
  {
    MARPA_OFF_ASSERT(Position_of_OR(parent) ≤ 1 ∨ predecessor)
        const DAND new ⇐ draft_and_node_new(obs, predecessor, cause);
    Next_DAND_of_DAND(new) ⇐ DANDs_of_OR(parent);
    DANDs_of_OR(parent) ⇐ new;
  }
```

908. ⟨ Create draft and-nodes for work_earley_set_ordinal 908 ⟩ ≡

```
{
  int item_ordinal;
  for (item_ordinal ⇐ 0; item_ordinal < item_count; item_ordinal++) {
    OR or_node ⇐ OR_by_PSI(per_ys_data, work_earley_set_ordinal,
        item_ordinal);
    const YIM work_earley_item ⇐ yims_of_ys[item_ordinal];
    const int work_origin_ordinal ⇐
        Ord_of_YS(Origin_of_YIM(work_earley_item));
    ⟨ Reset or_node to proper predecessor 909 ⟩
    if (or_node) {
      ⟨ Create draft and-nodes for or_node 910 ⟩
    }
  }
}
```

This code is used in section 891.

909. From an or-node, which may be nulling, determine its proper predecessor. Set or_node to 0 if there is none.

⟨ Reset or_node to proper predecessor 909 ⟩ ≡

```
{
  while (or_node) {
    DAND draft_and_node ⇐ DANDs_of_OR(or_node);
    OR predecessor_or;
    if (¬draft_and_node) break;
    predecessor_or ⇐ Predecessor_OR_of_DAND(draft_and_node);
    if (predecessor_or ∧ YS_Ord_of_OR(predecessor_or) ≠ work_earley_set_ordinal)
      break;
    or_node ⇐ predecessor_or;
  }
}
```

This code is used in section 908.

910. ⟨ Create draft and-nodes for or_node 910 ⟩ ≡

```
{
  const AHM work_ahm ⇐ AHM_of_YIM(work_earley_item);
  MARPA_ASSERT(work_ahm ≥ AHM_by_ID(1))
  const int work_symbol_instance ⇐ SYMI_of_AHM(work_ahm);
  const OR work_proper_or_node ⇐ or_by_origin_and_symi(per_ys_data,
      work_origin_ordinal, work_symbol_instance);
  ⟨ Create Leo draft and-nodes 912 ⟩
  ⟨ Create draft and-nodes for token sources 924 ⟩
  ⟨ Create draft and-nodes for completion sources 926 ⟩
}
```

This code is used in section 908.

911. To Do: I believe there's an easier and faster way to do this. I need to double-check the proofs, but it relies on these facts:

- Each item on a Leo path, other than the top node, had one and only one effect node.
- Each expanded item on a Leo path has exactly one Leo SRCL. (An expanded YIM is a YIM which was not in the Earley sets, but which needed to be expanded later. All Leo YIM's, except the summit and trailhead YIM's are expanded nodes.)
- In ascending a Leo trail, adding SRCL as I proceed, I can stop when I hit the first YIM that already has a Leo SRCL, because I can assume that the process that added its Leo SRCL must have added Leo SRCL's to all the current Leo trail YIM's indirect effect YIM's, which are above it on this Leo trail.

912. Therefore, the following should work: For each draft or-node track whether it is a Leo trail or-node, and whether it has a Leo SRCL. (This is two booleans.) The summit Leo or-node counts as a Leo trail or-node for this purpose. The summit Leo YIM will have its "Leo-SRCL-added" boolean set when it is initialized. All other Leo trail or-nodes will have the "Leo-SRCL-added" bits unset, initially. For each Leo trailhead, ascend the trail, adding SRCL's as I climb, until I find a Leo path item with the "Leo-SRCL-added" bit set. At that point I can stop the ascent.

```
⟨ Create Leo draft and-nodes 912 ⟩ ≡
  {
    SRCL source_link;
    for (source_link ⇐ First_Leo_SRCL_of_YIM(work_earley_item); source_link;
         source_link ⇐ Next_SRCL_of_SRCL(source_link)) {
      YIM cause_earley_item;
      LIM leo_predecessor;

      /* If source_link is active, everything on the Leo path is active. */
      if (¬SRCL_is_Active(source_link)) continue;
      cause_earley_item ⇐ Cause_of_SRCL(source_link);
      leo_predecessor ⇐ LIM_of_SRCL(source_link);
      if (leo_predecessor) {
        ⟨ Add draft and-nodes for chain starting with leo_predecessor 913 ⟩
      }
    }
  }
```

This code is used in section 910.

913. Note that in a trivial path the bottom is also the top.

```
⟨ Add draft and-nodes for chain starting with leo_predecessor 913 ⟩ ≡
  {    /* The rule for the Leo path Earley item */
    IRL path_irl ⇐ Λ;    /* The rule for the previous Leo path Earley item */
    IRL previous_path_irl;
    LIM path_leo_item ⇐ leo_predecessor;
    LIM higher_path_leo_item ⇐ Predecessor_LIM_of_LIM(path_leo_item);
    OR dand_predecessor;
```

```
    OR path_or_node;
    YIM base_earley_item ⟸ Trailhead_YIM_of_LIM(path_leo_item);
    dand_predecessor ⟸ set_or_from_yim(per_ys_data, base_earley_item);
    ⟨ Set path_or_node 914 ⟩
    ⟨ Add draft and-nodes to the bottom or-node 916 ⟩
    previous_path_irl ⟸ path_irl;
    while (higher_path_leo_item) {
      path_leo_item ⟸ higher_path_leo_item;
      higher_path_leo_item ⟸ Predecessor_LIM_of_LIM(path_leo_item);
      base_earley_item ⟸ Trailhead_YIM_of_LIM(path_leo_item);
      dand_predecessor ⟸ set_or_from_yim(per_ys_data, base_earley_item);
      ⟨ Set path_or_node 914 ⟩
      ⟨ Add the draft and-nodes to an upper Leo path or-node 919 ⟩
      previous_path_irl ⟸ path_irl;
    }
  }
```

This code is used in section 912.

914. ⟨ Set path_or_node 914 ⟩ ≡

```
  {
    if (higher_path_leo_item) {
      ⟨ Use Leo base data to set path_or_node 923 ⟩
    }
    else {
      path_or_node ⟸ work_proper_or_node;
    }
  }
```

This code is used in section 913.

915. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE OR or_by_origin_and_symi(struct s_bocage_setup_per_ys
          *per_ys_data, YSID origin, SYMI symbol_instance)
  {
    const PSL or_psl_at_origin ⟸ per_ys_data[(origin)].t_or_psl;
    return PSL_Datum(or_psl_at_origin, (symbol_instance));
  }
```

916. ⟨ Add draft and-nodes to the bottom or-node 916 ⟩ ≡

```
  {
    const OR dand_cause ⟸ set_or_from_yim(per_ys_data, cause_earley_item);
    if (¬dand_is_duplicate(path_or_node, dand_predecessor, dand_cause)) {
      draft_and_node_add(bocage_setup_obs, path_or_node, dand_predecessor,
          dand_cause);
    }
  }
```

This code is used in section 913.

917. The test for duplication is necessary, because while a single Leo path is deterministic, there can be multiple Leo paths, and they can overlap, and they can overlap with nodes from other sources.

918. To Do: I need to justify the claim that the time complexity is not altered by the check for duplicates. In the case of unambiguous grammars, there is only one Leo path and only once source, so the proof is straightforward. For ambiguous grammars, I believe I can show that the number of traversals of each Leo path item is bounded by a constant, and the time complexity bound follows.

919. To Do: On the more practical side, I conjecture that, once a duplicate has been found when ascending a Leo path, it can be assumed that all attempts to add *DAND*'s to higher Leo path items will also duplicate. If so, the loop that ascends the Leo path can be ended at that point.

⟨ Add the draft and-nodes to an upper Leo path or-node 919 ⟩ ≡

```
{
  const SYMI symbol_instance ⇐ SYMI_of_Completed_IRL(previous_path_irl);
  const int origin ⇐ Ord_of_YS(YS_of_LIM(path_leo_item));
  const OR dand_cause ⇐ or_by_origin_and_symi(per_ys_data, origin,
      symbol_instance);
  if (¬dand_is_duplicate(path_or_node, dand_predecessor, dand_cause)) {
    draft_and_node_add(bocage_setup_obs, path_or_node, dand_predecessor,
        dand_cause);
  }
}
```

This code is used in section 913.

920. Assuming they have the same parent, would the DANDs made up from these OR node's be equivalent. For locations, the parent dictates the beginning and end, so only the start of the cause and the end of predecessor matter. These must be the same (the "middle" location) so that only this middle location needs to be compared. For the predecessors, dotted rule is a function of the parent. For token causes, the alternative reading logic guaranteed that there would be no two tokens which differed only in value, so only the symbols needs to be compared. For component causes, they are always completions, so that only the IRL ID needs to be compared.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int dands_are_equal(OR predecessor_a, OR cause_a, OR
        predecessor_b, OR cause_b)
{
  const int a_is_token ⇐ OR_is_Token(cause_a);
  const int b_is_token ⇐ OR_is_Token(cause_b);
  if (a_is_token ≠ b_is_token) return 0;
  {     /* -1 means equal to the start of the parent, which is sufficient for comparision
          purposes */
    const int middle_of_a ⇐ predecessor_a ? YS_Ord_of_OR(predecessor_a) : −1;
```

```
    const int middle_of_b ⟸ predecessor_b ? YS_Ord_of_OR(predecessor_b) : −1;
    if (middle_of_a ≠ middle_of_b) return 0;
  }
  if (a_is_token) {
    const NSYID nsyid_of_a ⟸ NSYID_of_OR(cause_a);
    const NSYID nsyid_of_b ⟸ NSYID_of_OR(cause_b);
    return nsyid_of_a ≡ nsyid_of_b;
  }
  {   /* If here, we know that both causes are rule completions. */
    const IRLID irlid_of_a ⟸ IRLID_of_OR(cause_a);
    const IRLID irlid_of_b ⟸ IRLID_of_OR(cause_b);
    return irlid_of_a ≡ irlid_of_b;
  }   /* Not reached */
}
```

921. Return 1 if a new dand made up of `predecessor` and `cause` would duplicate any already in `parent`. Otherwise, return 0.

```
⟨ Function definitions 41 ⟩ +≡
PRIVATE int dand_is_duplicate(OR parent, OR predecessor, OR cause)
{
  DAND dand;
  for (dand ⟸ DANDs_of_OR(parent); dand; dand ⟸ Next_DAND_of_DAND(dand)) {
    if (dands_are_equal(predecessor, cause, Predecessor_OR_of_DAND(dand),
          Cause_OR_of_DAND(dand))) {
      return 1;
    }
  }
  return 0;
}
```

922. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE OR set_or_from_yim(struct s_bocage_setup_per_ys *per_ys_data, YIM
      psi_yim)
{
  const YIM psi_earley_item ⟸ psi_yim;
  const int psi_earley_set_ordinal ⟸ YS_Ord_of_YIM(psi_earley_item);
  const int psi_item_ordinal ⟸ Ord_of_YIM(psi_earley_item);
  return OR_by_PSI(per_ys_data, psi_earley_set_ordinal, psi_item_ordinal);
}
```

923. ⟨ Use Leo base data to set path_or_node 923 ⟩ ≡

```
{
  int symbol_instance;
  const int origin_ordinal ⇐ Origin_Ord_of_YIM(base_earley_item);
  const AHM ahm ⇐ AHM_of_YIM(base_earley_item);
  path_irl ⇐ IRL_of_AHM(ahm);
  symbol_instance ⇐ Last_Proper_SYMI_of_IRL(path_irl);
  path_or_node ⇐ or_by_origin_and_symi(per_ys_data, origin_ordinal,
      symbol_instance);
}
```

This code is used in section 914.

924. Token or-nodes are pseudo-or-nodes. They are not included in the count of or-nodes, are not coverted to final or-nodes, and are not traversed when traversing or-nodes by ID.

⟨ Create draft and-nodes for token sources 924 ⟩ ≡

```
{
  SRCL tkn_source_link;
  for (tkn_source_link ⇐ First_Token_SRCL_of_YIM(work_earley_item);
        tkn_source_link;  tkn_source_link ⇐
        Next_SRCL_of_SRCL(tkn_source_link)) {
    OR new_token_or_node;
    const NSYID token_nsyid ⇐ NSYID_of_SRCL(tkn_source_link);
    const YIM predecessor_earley_item ⇐
        Predecessor_of_SRCL(tkn_source_link);
    const OR dand_predecessor ⇐ safe_or_from_yim(per_ys_data,
        predecessor_earley_item);
    if (NSYID_is_Valued_in_B(b, token_nsyid)) {
    /* I probably can and should use a smaller allocation, sized just for a token
          or-node */
      new_token_or_node ⇐ (OR) marpa_obs_new(OBS_of_B(b), OR_Object, 1);
      Type_of_OR(new_token_or_node) ⇐ VALUED_TOKEN_OR_NODE;
      NSYID_of_OR(new_token_or_node) ⇐ token_nsyid;
      Value_of_OR(new_token_or_node) ⇐ Value_of_SRCL(tkn_source_link);
    }
    else {
      new_token_or_node ⇐ Unvalued_OR_by_NSYID(token_nsyid);
    }
    draft_and_node_add(bocage_setup_obs, work_proper_or_node,
        dand_predecessor, new_token_or_node);
  }
}
```

This code is used in section 910.

925. "Safe" because it does not require called to ensure the such an or-node exists.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE OR safe_or_from_yim(struct s_bocage_setup_per_ys *per_ys_data, YIM
        yim)
{
  if (Position_of_AHM(AHM_of_YIM(yim)) < 1) return Λ;
  return set_or_from_yim(per_ys_data, yim);
}
```

926. ⟨ Create draft and-nodes for completion sources 926 ⟩ ≡

```
{
  SRCL source_link;
  for (source_link ⇐ First_Completion_SRCL_of_YIM(work_earley_item);
        source_link; source_link ⇐ Next_SRCL_of_SRCL(source_link)) {
    YIM predecessor_earley_item ⇐ Predecessor_of_SRCL(source_link);
    YIM cause_earley_item ⇐ Cause_of_SRCL(source_link);
    const int middle_ordinal ⇐ Origin_Ord_of_YIM(cause_earley_item);
    const AHM cause_ahm ⇐ AHM_of_YIM(cause_earley_item);
    const SYMI cause_symbol_instance ⇐
        SYMI_of_Completed_IRL(IRL_of_AHM(cause_ahm));
    OR dand_predecessor ⇐ safe_or_from_yim(per_ys_data,
        predecessor_earley_item);
    const OR dand_cause ⇐ or_by_origin_and_symi(per_ys_data,
        middle_ordinal, cause_symbol_instance);
    draft_and_node_add(bocage_setup_obs, work_proper_or_node,
        dand_predecessor, dand_cause);
  }
}
```

This code is used in section 910.

927. The need for this count is a vestige of duplicate checking. Now that duplicates no longer occur, the whole process probably can and should be simplified.

⟨ Count draft and-nodes 927 ⟩ ≡

```
{
  const int or_node_count_of_b ⇐ OR_Count_of_B(b);
  int or_node_id ⇐ 0;
  while (or_node_id < or_node_count_of_b) {
    const OR work_or_node ⇐ OR_of_B_by_ID(b, or_node_id);
    DAND dand ⇐ DANDs_of_OR(work_or_node);
    while (dand) {
      unique_draft_and_node_count++;
      dand ⇐ Next_DAND_of_DAND(dand);
    }
    or_node_id++;
```

```
    }
  }
```

This code is used in section 932.

928. And-node (AND) code. The and-nodes are part of the parse bocage. They are analogous to the and-nodes of a standard parse forest, except that they are binary – restricted to two children. This means that the parse bocage stores the parse in a kind of Chomsky Normal Form. (A second difference between a parse bocage and a parse forest, is that the parse bocage can contain cycles.)

```
⟨ Public typedefs 91 ⟩ +≡
  typedef int Marpa_And_Node_ID;
```

929.
```
⟨ Private typedefs 49 ⟩ +≡
  typedef Marpa_And_Node_ID ANDID;
```

930.
```
⟨ Private incomplete structures 107 ⟩ +≡
  struct s_and_node;
  typedef struct s_and_node *AND;
```

931.
```
#define OR_of_AND(and) ((and)→t_current)
#define Predecessor_OR_of_AND(and) ((and)→t_predecessor)
#define Cause_OR_of_AND(and) ((and)→t_cause)
⟨ Private structures 48 ⟩ +≡
  struct s_and_node {
    OR t_current;
    OR t_predecessor;
    OR t_cause;
  };
  typedef struct s_and_node AND_Object;
```

932.
```
⟨ Create the final and-nodes for all earley sets 932 ⟩ ≡
  {
    int unique_draft_and_node_count ⇐ 0;
    ⟨ Count draft and-nodes 927 ⟩
    ⟨ Create the final and-node array 933 ⟩
  }
```
This code is used in section 942.

933.
```
⟨ Create the final and-node array 933 ⟩ ≡
  {
    const int or_count_of_b ⇐ OR_Count_of_B(b);
    int or_node_id;
    int and_node_id ⇐ 0;
    const AND ands_of_b ⇐ ANDs_of_B(b) ⇐ marpa_new(AND_Object,
        unique_draft_and_node_count);
    for (or_node_id ⇐ 0; or_node_id < or_count_of_b; or_node_id++) {
      int and_count_of_parent_or ⇐ 0;
      const OR or_node ⇐ OR_of_B_by_ID(b, or_node_id);
```

```
      DAND dand ⟸ DANDs_of_OR(or_node);
      First_ANDID_of_OR(or_node) ⟸ and_node_id;
      while (dand) {
        const OR cause_or_node ⟸ Cause_OR_of_DAND(dand);
        const AND and_node ⟸ ands_of_b + and_node_id;
        OR_of_AND(and_node) ⟸ or_node;
        Predecessor_OR_of_AND(and_node) ⟸ Predecessor_OR_of_DAND(dand);
        Cause_OR_of_AND(and_node) ⟸ cause_or_node;
        and_node_id++;
        and_count_of_parent_or++;
        dand ⟸ Next_DAND_of_DAND(dand);
      }
      AND_Count_of_OR(or_node) ⟸ and_count_of_parent_or;
      if (and_count_of_parent_or > 1) Ambiguity_Metric_of_B(b) ⟸ 2;
    }
    AND_Count_of_B(b) ⟸ and_node_id;
    MARPA_ASSERT(and_node_id ≡ unique_draft_and_node_count);
  }
```

This code is used in section 932.

934. Parse bocage code (B, BOCAGE).

935. Pre-initialization is making the elements safe for the deallocation logic to be called. Often it is setting the value to zero, so that the deallocation logic knows when **not** to try deallocating a not-yet uninitialized value.

⟨ Public incomplete structures 47 ⟩ +≡
```
struct marpa_bocage;
typedef struct marpa_bocage *Marpa_Bocage;
```

936. ⟨ Private incomplete structures 107 ⟩ +≡
```
typedef struct marpa_bocage *BOCAGE;
```

937. ⟨ Bocage structure 937 ⟩ ≡
```
struct marpa_bocage {
  ⟨ Widely aligned bocage elements 885 ⟩
  ⟨ Int aligned bocage elements 886 ⟩
  ⟨ Bit aligned bocage elements 968 ⟩
};
```
This code is used in section 1383.

938. The base objects of the bocage.

939. ⟨ Unpack bocage objects 939 ⟩ ≡
```
const GRAMMAR g UNUSED ⟸ G_of_B(b);
```
This code is used in sections 955, 959, 966, 970, 977, 984, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1332, 1334, 1335, 1336, 1337, 1338, and 1339.

940. The bocage obstack. An obstack with the lifetime of the bocage.

```
#define OBS_of_B(b) ((b)→t_obs)
```
⟨ Widely aligned bocage elements 885 ⟩ +≡
```
struct marpa_obstack *t_obs;
```

941. ⟨ Destroy bocage elements, final phase 941 ⟩ ≡
```
marpa_obs_free(OBS_of_B(b));
```
This code is used in section 965.

942. Bocage construction.

⟨ Function definitions 41 ⟩ +≡
```
Marpa_Bocage marpa_b_new(Marpa_Recognizer r, Marpa_Earley_Set_ID ordinal_arg)
{
  ⟨ Return Λ on failure 1228 ⟩
  ⟨ Declare bocage locals 945 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  if (_MARPA_UNLIKELY(ordinal_arg ≤ −2)) {
    MARPA_ERROR(MARPA_ERR_INVALID_LOCATION);
    return failure_indicator;
```

```
  }
  ⟨ Fail if recognizer not started 1246 ⟩
  {
    struct marpa_obstack *const obstack ⟸ marpa_obs_init;
    b ⟸ marpa_obs_new(obstack, struct marpa_bocage, 1);
    OBS_of_B(b) ⟸ obstack;
  }
  ⟨ Initialize bocage elements 887 ⟩
  if (G_is_Trivial(g)) {
    switch (ordinal_arg) {
    default: goto NO_PARSE;
    case 0: case −1: break;
    }
    B_is_Nulling(b) ⟸ 1;
    return b;
  }
  r_update_earley_sets(r);
  ⟨ Set end_of_parse_earley_set and end_of_parse_earleme 949 ⟩
  if (end_of_parse_earleme ≡ 0) {
    if (¬XSY_is_Nullable(XSY_by_ID(g→t_start_xsy_id))) goto NO_PARSE;
    B_is_Nulling(b) ⟸ 1;
    return b;
  }
  ⟨ Find start_yim 952 ⟩
  if (¬start_yim) goto NO_PARSE;
  bocage_setup_obs ⟸ marpa_obs_init;
  ⟨ Allocate bocage setup working data 950 ⟩
  ⟨ Populate the PSI data 867 ⟩
  ⟨ Create the or-nodes for all earley sets 891 ⟩
  ⟨ Create the final and-nodes for all earley sets 932 ⟩
  ⟨ Set top or node id in b 953 ⟩;
  marpa_obs_free(bocage_setup_obs);
  return b;
NO_PARSE: ;
  MARPA_ERROR(MARPA_ERR_NO_PARSE);
  if (b) {
    ⟨ Destroy bocage elements, all phases 965 ⟩;
  }
  return Λ;
}
```

943.
```
#define Valued_BV_of_B(b) ((b)→t_valued_bv)
#define Valued_Locked_BV_of_B(b) ((b)→t_valued_locked_bv)
#define XSYID_is_Valued_in_B(b, xsyid) (lbv_bit_test(Valued_BV_of_B(b), (xsyid)))
#define NSYID_is_Valued_in_B(b, nsyid)
        XSYID_is_Valued_in_B((b), Source_XSYID_of_NSYID(nsyid))
```

⟨ Widely aligned bocage elements 885 ⟩ +≡
```
LBV t_valued_bv;
LBV t_valued_locked_bv;
```

944. ⟨ Initialize bocage elements 887 ⟩ +≡
```
Valued_BV_of_B(b) ⟸ lbv_clone(b→t_obs, r→t_valued, xsy_count);
Valued_Locked_BV_of_B(b) ⟸ lbv_clone(b→t_obs, r→t_valued_locked, xsy_count);
```

945. ⟨ Declare bocage locals 945 ⟩ ≡
```
const GRAMMAR g ⟸ G_of_R(r);
const int xsy_count ⟸ XSY_Count_of_G(g);
BOCAGE b ⟸ Λ;
YS end_of_parse_earley_set;
JEARLEME end_of_parse_earleme;
YIM start_yim ⟸ Λ;
struct marpa_obstack *bocage_setup_obs ⟸ Λ;
int count_of_earley_items_in_parse;
const int earley_set_count_of_r ⟸ YS_Count_of_R(r);
```
See also section 948.
This code is used in section 942.

946. ⟨ Private incomplete structures 107 ⟩ +≡
```
struct s_bocage_setup_per_ys;
```

947. These macros were introduced for development. They may be worth keeping.
```
#define OR_by_PSI(psi_data, set_ordinal, item_ordinal)
        (((psi_data)[(set_ordinal)].t_or_node_by_item)[(item_ordinal)])
```
⟨ Private structures 48 ⟩ +≡
```
struct s_bocage_setup_per_ys {
  OR *t_or_node_by_item;
  PSL t_or_psl;
  PSL t_and_psl;
};
```

948. ⟨ Declare bocage locals 945 ⟩ +≡
```
struct s_bocage_setup_per_ys *per_ys_data ⟸ Λ;
```

949. ⟨ Set end_of_parse_earley_set and end_of_parse_earleme 949 ⟩ ≡

```
{
  if (ordinal_arg ≡ −1) {
    end_of_parse_earley_set ⟸ YS_at_Current_Earleme_of_R(r);
  }
  else {      /* ordinal_arg != -1 */
    if (¬YS_Ord_is_Valid(r, ordinal_arg)) {
      MARPA_ERROR(MARPA_ERR_INVALID_LOCATION);
      return failure_indicator;
    }
    end_of_parse_earley_set ⟸ YS_of_R_by_Ord(r, ordinal_arg);
  }
  if (¬end_of_parse_earley_set) goto NO_PARSE;
  end_of_parse_earleme ⟸ Earleme_of_YS(end_of_parse_earley_set);
}
```

This code is used in section 942.

950.

⟨ Allocate bocage setup working data 950 ⟩ ≡

```
{
  int earley_set_ordinal;
  int earley_set_count ⟸ YS_Count_of_R(r);
  count_of_earley_items_in_parse ⟸ 0;
  per_ys_data ⟸ marpa_obs_new(bocage_setup_obs, struct s_bocage_setup_per_ys,
      earley_set_count);
  for (earley_set_ordinal ⟸ 0; earley_set_ordinal < earley_set_count;
        earley_set_ordinal++) {
    const YS_Const earley_set ⟸ YS_of_R_by_Ord(r, earley_set_ordinal);
    const int item_count ⟸ YIM_Count_of_YS(earley_set);
    count_of_earley_items_in_parse += item_count;
    {
      int item_ordinal;
      struct s_bocage_setup_per_ys *per_ys ⟸ per_ys_data + earley_set_ordinal;
      per_ys→t_or_node_by_item ⟸ marpa_obs_new(bocage_setup_obs, OR,
          item_count);
      per_ys→t_or_psl ⟸ Λ;
      per_ys→t_and_psl ⟸ Λ;
      for (item_ordinal ⟸ 0; item_ordinal < item_count; item_ordinal++) {
        OR_by_PSI(per_ys_data, earley_set_ordinal, item_ordinal) ⟸ Λ;
      }
    }
  }
}
```

This code is used in section 942.

951. Predicted AHFA states can be skipped since they contain no completions. Note that AHFA state 0 is not marked as a predicted AHFA state, even though it can contain a predicted AHM.

952. The search for the start Earley item is done once per parse — $O(s)$, where s is the size of the end of parse Earley set. This makes it very hard to justify any precomputations to help the search, because if they have to be done once per Earley set, that is a $O(|w| \cdot s')$ overhead, where $|w|$ is the length of the input, and where s' is the average size of an Earley set. It is hard to believe that for practical grammars that $O(|w| \cdot s') <= O(s)$, which is what it would take for any per-Earley set overhead to make sense.

⟨ Find start_yim 952 ⟩ ≡

```
{
  int yim_ix;
  YIM *const earley_items ⇐ YIMs_of_YS(end_of_parse_earley_set);
  const IRL start_irl ⇐ g→t_start_irl;
  const IRLID sought_irl_id ⇐ ID_of_IRL(start_irl);
  const int earley_item_count ⇐ YIM_Count_of_YS(end_of_parse_earley_set);
  for (yim_ix ⇐ 0; yim_ix < earley_item_count; yim_ix++) {
    const YIM earley_item ⇐ earley_items[yim_ix];
    if (Origin_Earleme_of_YIM(earley_item) > 0) continue;
        /* Not a start YIM */
    if (YIM_was_Predicted(earley_item)) continue;
    {
      const AHM ahm ⇐ AHM_of_YIM(earley_item);
      if (IRLID_of_AHM(ahm) ≡ sought_irl_id) {
        start_yim ⇐ earley_item;
        break;
      }
    }
  }
}
```

This code is used in section 942.

953. ⟨ Set top or node id in b 953 ⟩ ≡

```
{
  const YSID end_of_parse_ordinal ⇐ Ord_of_YS(end_of_parse_earley_set);
  const int start_earley_item_ordinal ⇐ Ord_of_YIM(start_yim);
  const OR root_or_node ⇐ OR_by_PSI(per_ys_data, end_of_parse_ordinal,
      start_earley_item_ordinal);
  Top_ORID_of_B(b) ⇐ ID_of_OR(root_or_node);
}
```

This code is used in section 942.

954. Top or-node.

955. If b is nulling, the top Or node ID will be -1.

⟨ Function definitions 41 ⟩ +≡

```
Marpa_Or_Node_ID _marpa_b_top_or_node(Marpa_Bocage b)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  return Top_ORID_of_B(b);
}
```

956. Ambiguity metric. An ambiguity metric, named vaguely because it is vaguely defined. It is 1 if the parse in not ambiguous, and greater than 1 if it is ambiguous. For convenience, it is initialized to 1.

```
#define Ambiguity_Metric_of_B(b) ((b)→t_ambiguity_metric)
```

957. ⟨ Int aligned bocage elements 886 ⟩ +≡

```
int t_ambiguity_metric;
```

958. ⟨ Initialize bocage elements 887 ⟩ +≡

```
Ambiguity_Metric_of_B(b) ⟸ 1;
```

959. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_b_ambiguity_metric(Marpa_Bocage b)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  return Ambiguity_Metric_of_B(b);
}
```

960. Reference counting and destructors.

961. ⟨ Int aligned bocage elements 886 ⟩ +≡

```
int t_ref_count;
```

962. ⟨ Initialize bocage elements 887 ⟩ +≡

```
b→t_ref_count ⟸ 1;
```

963. Decrement the bocage reference count.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void bocage_unref(BOCAGE b)
{
  MARPA_ASSERT(b→t_ref_count > 0)b→t_ref_count−−;
  if (b→t_ref_count ≤ 0) {
    bocage_free(b);
  }
```

```
}
void marpa_b_unref(Marpa_Bocage b)
{
  bocage_unref(b);
}
```

964. Increment the bocage reference count.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE BOCAGE bocage_ref(BOCAGE b)
{
  MARPA_ASSERT(b→t_ref_count > 0)b→t_ref_count++;
  return b;
}
Marpa_Bocage marpa_b_ref(Marpa_Bocage b)
{
  return bocage_ref(b);
}
```

965. Bocage destruction.

⟨ Destroy bocage elements, all phases 965 ⟩ ≡

```
⟨ Destroy bocage elements, main phase 888 ⟩;
⟨ Destroy bocage elements, final phase 941 ⟩;
```

This code is used in sections 942 and 966.

966. This function is safe to call even if the bocage already has been freed, or was never initialized.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void bocage_free(BOCAGE b)
{
  ⟨ Unpack bocage objects 939 ⟩
  if (b) {
    ⟨ Destroy bocage elements, all phases 965 ⟩;
  }
}
```

967. Bocage is nulling?. Is this bocage for a nulling parse?

```
#define B_is_Nulling(b) ((b)→t_is_nulling)
```

968. ⟨ Bit aligned bocage elements 968 ⟩ ≡

```
BITFIELD t_is_nulling:1;
```

This code is used in section 937.

969. ⟨ Initialize bocage elements 887 ⟩ +≡

```
B_is_Nulling(b) ⟸ 0;
```

970. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_b_is_null(Marpa_Bocage b)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  return B_is_Nulling(b);
}
```

971. Ordering (O, ORDER) code.

⟨ Public incomplete structures 47 ⟩ +≡
    struct marpa_order;
    typedef struct marpa_order ∗*Marpa_Order*;

972. ⟨ Public incomplete structures 47 ⟩ +≡
    typedef Marpa_Order ORDER;

973. `t_ordering_obs` is an obstack which contains the ordering information for non-default orderings. It is non-null if and only if `t_and_node_orderings` is non-null.

#define `OBS_of_O(order)` `((order)→t_ordering_obs)`
#define `O_is_Default(order)` `(¬OBS_of_O(order))`
#define `O_is_Frozen(o)` `((o)→t_is_frozen)`

⟨ Private structures 48 ⟩ +≡
    struct marpa_order {
        struct `marpa_obstack` ∗`t_ordering_obs`;
        ANDID ∗∗`t_and_node_orderings`;
        ⟨ Widely aligned order elements 976 ⟩
        ⟨ Int aligned order elements 979 ⟩
        ⟨ Bit aligned order elements 990 ⟩
        BITFIELD `t_is_frozen`:1;
    };

974. ⟨ Pre-initialize order elements 974 ⟩ ≡
    {
        o→`t_and_node_orderings` ⟸ Λ;
        o→`t_is_frozen` ⟸ 0;
        `OBS_of_O`(*o*) ⟸ Λ;
    }

See also sections 980 and 993.

This code is used in section 977.

975. The base objects of the bocage.

976. *#define* `B_of_O(b)` `((b)→t_bocage)`

⟨ Widely aligned order elements 976 ⟩ ≡
    BOCAGE `t_bocage`;

This code is used in section 973.

977. ⟨ Function definitions 41 ⟩ +≡
    Marpa_Order `marpa_o_new`(*Marpa_Bocage b*)
    {
        ⟨ Return Λ on failure 1228 ⟩
        ⟨ Unpack bocage objects 939 ⟩
        ORDER o;

```
    ⟨ Fail if fatal error 1249 ⟩
    o ⇐ my_malloc(sizeof (*o));
    B_of_O(o) ⇐ b;
    bocage_ref(b);
    ⟨ Pre-initialize order elements 974 ⟩
    O_is_Nulling(o) ⇐ B_is_Nulling(b);
    Ambiguity_Metric_of_O(o) ⇐ −1;
    return o;
  }
```

978. Reference counting and destructors.

979. ⟨ Int aligned order elements 979 ⟩ ≡

```
  int t_ref_count;
```

See also sections 986 and 992.

This code is used in section 973.

980. ⟨ Pre-initialize order elements 974 ⟩ +≡

```
  o→t_ref_count ⇐ 1;
```

981. Decrement the order reference count.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void order_unref(ORDER o)
  {
    MARPA_ASSERT(o→t_ref_count > 0)o→t_ref_count−−;
    if (o→t_ref_count ≤ 0) {
      order_free(o);
    }
  }
  void marpa_o_unref(Marpa_Order o)
  {
    order_unref(o);
  }
```

982. Increment the order reference count.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE ORDER order_ref(ORDER o)
  {
    MARPA_ASSERT(o→t_ref_count > 0)o→t_ref_count++;
    return o;
  }
  Marpa_Order marpa_o_ref(Marpa_Order o)
  {
    return order_ref(o);
  }
```

983. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void order_free(ORDER o)
{
  ⟨ Unpack order objects 984 ⟩
  bocage_unref(b);
  marpa_obs_free(OBS_of_O(o));
  my_free(o);
}
```

984. ⟨ Unpack order objects 984 ⟩ ≡

```
const BOCAGE b ⟸ B_of_O(o);
⟨ Unpack bocage objects 939 ⟩
```

This code is used in sections 983, 987, 991, 994, 995, 999, 1008, 1023, 1025, 1329, and 1330.

985. Ambiguity metric. An ambiguity metric, named vaguely because it is vaguely defined. It is 1 if the parse in not ambiguous, and greater than 1 if it is ambiguous. For convenience, it is initialized to 1.

```
#define Ambiguity_Metric_of_O(o) ((o)→t_ambiguity_metric)
```

986. ⟨ Int aligned order elements 979 ⟩ +≡

```
int t_ambiguity_metric;
```

987. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_o_ambiguity_metric(Marpa_Order o)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack order objects 984 ⟩
  const int old_ambiguity_metric_of_o ⟸ Ambiguity_Metric_of_O(o);
  const int ambiguity_metric_of_b ⟸ (Ambiguity_Metric_of_B(b) ≤ 1 ? 1 : 2);
  ⟨ Fail if fatal error 1249 ⟩
  O_is_Frozen(o) ⟸ 1;
  if (old_ambiguity_metric_of_o ≥ 0) return old_ambiguity_metric_of_o;
  if (ambiguity_metric_of_b < 2    /* If bocage is unambiguous */
  ∨ O_is_Default(o)    /* or we are using the default order */
  ∨ High_Rank_Count_of_O(o) ≤ 0    /* or we are not using high rank order */
  ) {    /* then ... */
    Ambiguity_Metric_of_O(o) ⟸ ambiguity_metric_of_b;
      /* copy the bocage metric */
    return ambiguity_metric_of_b;    /* and return it. */
  }
  ⟨ Compute ambiguity metric of ordering by high rank 988 ⟩
  return Ambiguity_Metric_of_O(o);
}
```

988. If we are here, the caller has made sure the bocage is ambiguous, and that we are using the high rank order.

⟨ Compute ambiguity metric of ordering by high rank 988 ⟩ ≡

```
{
  ANDID **const and_node_orderings ⇐ o→t_and_node_orderings;
  const AND and_nodes ⇐ ANDs_of_B(b);
  ORID *top_of_stack;
  const ORID root_or_id ⇐ Top_ORID_of_B(b);
  FSTACK_DECLARE(or_node_stack, ORID)
  const int or_count ⇐ OR_Count_of_B(b);
  Bit_Vector bv_orid_was_stacked;    /* do not stack an ORID twice */
  Ambiguity_Metric_of_O(o) ⇐ 1;
    /* initialize the ambiguity metric to unambiguous */
  bv_orid_was_stacked ⇐ bv_create(or_count);
  FSTACK_INIT(or_node_stack, ORID, or_count);
  *(FSTACK_PUSH(or_node_stack)) ⇐ root_or_id;
  bv_bit_set(bv_orid_was_stacked, root_or_id);
  while ((top_of_stack ⇐ FSTACK_POP(or_node_stack))) {
    const ORID or_id ⇐ *top_of_stack;
    const OR or_node ⇐ OR_of_B_by_ID(b, or_id);
    ANDID *ordering ⇐ and_node_orderings[or_id];
    int and_count ⇐ ordering ? ordering[0] : AND_Count_of_OR(or_node);
    if (and_count > 1) {    /* If there the and-node count is greater than 1, the
          and-node, is ambiguous */
      Ambiguity_Metric_of_O(o) ⇐ 2;    /* ... and so is the entire ordering */
      goto END_OR_NODE_LOOP;    /* ... and we are done */
    }
    {
      const ANDID and_id ⇐ ordering ? ordering[1] :
          First_ANDID_of_OR(or_node);
      const AND and_node ⇐ and_nodes + and_id;
      const OR predecessor_or ⇐ Predecessor_OR_of_AND(and_node);
      const OR cause_or ⇐ Cause_OR_of_AND(and_node);
      if (predecessor_or) {
        const ORID predecessor_or_id ⇐ ID_of_OR(predecessor_or);
        if (¬bv_bit_test_then_set(bv_orid_was_stacked, predecessor_or_id)) {
          *(FSTACK_PUSH(or_node_stack)) ⇐ predecessor_or_id;
        }
      }
      if (cause_or ∧ ¬OR_is_Token(cause_or)) {
        const ORID cause_or_id ⇐ ID_of_OR(cause_or);
        if (¬bv_bit_test_then_set(bv_orid_was_stacked, cause_or_id)) {
          *(FSTACK_PUSH(or_node_stack)) ⇐ cause_or_id;
```

```
          }
        }
      }
    }
  END_OR_NODE_LOOP: ;
    FSTACK_DESTROY(or_node_stack);
    bv_free(bv_orid_was_stacked);    /* for now copy the bocage metric */
  }
```

This code is used in section 987.

989. Order is nulling?. Is this order for a nulling parse?

```
#define O_is_Nulling(o) ((o)→t_is_nulling)
```

990. ⟨ Bit aligned order elements 990 ⟩ ≡

```
  BITFIELD t_is_nulling:1;
```

This code is used in section 973.

991. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_o_is_null(Marpa_Order o)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack order objects 984 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    return O_is_Nulling(o);
  }
```

992. In the future perhaps, a "high rank count" of n might indicate that the n highest ranks should be included. Right now the only values allowed are 0 (allow everything) and 1.

```
#define High_Rank_Count_of_O(order) ((order)→t_high_rank_count)
```

⟨ Int aligned order elements 979 ⟩ +≡

```
  int t_high_rank_count;
```

993. ⟨ Pre-initialize order elements 974 ⟩ +≡

```
  High_Rank_Count_of_O(o) ⟸ 1;
```

994. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_o_high_rank_only_set(Marpa_Order o, int count)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack order objects 984 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    if (O_is_Frozen(o)) {
      MARPA_ERROR(MARPA_ERR_ORDER_FROZEN);
      return failure_indicator;
```

```
    }
    if (_MARPA_UNLIKELY(count < 0 ∨ count > 1)) {
      MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
      return failure_indicator;
    }
    return High_Rank_Count_of_O(o) ⟸ count;
  }
```

995.

```
⟨ Function definitions 41 ⟩ +≡
  int marpa_o_high_rank_only(Marpa_Order o)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack order objects 984 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    return High_Rank_Count_of_O(o);
  }
```

996. Set the order of and-nodes. This function sets the order in which the and-nodes of an or-node are used.

997. Using a boolean vector for the index of an and-node within an or-node, instead of the and-node ID, would seem to allow an space efficiency: the size of the boolean vector could be reduced to the maximum number of descendents of any or-node. But in fact, improvements from this approach are elusive.

In the worst cases, these counts are the same, or almost the same. Any attempt to economize on space seems to always be counter-productive in terms of speed. And since allocating a boolean vector for the worst case does not increase the memory high water mark, it would seems to be the most reasonable tradeoff.

This in turn suggests there is no advantage is using a within-or-node index to index the boolean vector, instead of using the and-node id to index the boolean vector. Using the and-node ID does have the advantage that the bit vector does not need to be cleared for each or-node.

998. The first position in each `and_node_orderings` array is not actually an *ANDID*, but a count. A purist might insist this needs to be reflected in a structure, but to my mind doing this portably makes the code more obscure, not less.

999. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_o_rank(Marpa_Order o)
  {
    ANDID **and_node_orderings;
    struct marpa_obstack *obs;
    int bocage_was_reordered ⟸ 0;
    ⟨ Return −2 on failure 1229 ⟩
```

```
    ⟨ Unpack order objects 984 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    if (O_is_Frozen(o)) {
      MARPA_ERROR(MARPA_ERR_ORDER_FROZEN);
      return failure_indicator;
    }
    ⟨ Initialize obs and and_node_orderings 1005 ⟩
    if (High_Rank_Count_of_O(o)) {
      ⟨ Sort bocage for "high rank only" 1000 ⟩
    }
    else {
      ⟨ Sort bocage for "rank by rule" 1003 ⟩
    }
    if (¬bocage_was_reordered) {
      marpa_obs_free(obs);
      OBS_of_O(o) ⟸ Λ;
      o→t_and_node_orderings ⟸ Λ;
    }
    O_is_Frozen(o) ⟸ 1;
    return 1;
  }
```

1000. ⟨ Sort bocage for "high rank only" 1000 ⟩ ≡

```
  {
    const AND and_nodes ⟸ ANDs_of_B(b);
    const int or_node_count_of_b ⟸ OR_Count_of_B(b);
    int or_node_id ⟸ 0;
    while (or_node_id < or_node_count_of_b) {
      const OR work_or_node ⟸ OR_of_B_by_ID(b, or_node_id);
      const ANDID and_count_of_or ⟸ AND_Count_of_OR(work_or_node);
      ⟨ Sort work_or_node for "high rank only" 1001 ⟩
      or_node_id++;
    }
  }
```

This code is used in section 999.

1001. ⟨ Sort work_or_node for "high rank only" 1001 ⟩ ≡

```
  {
    if (and_count_of_or > 1) {
      int high_rank_so_far ⟸ INT_MIN;
      const ANDID first_and_node_id ⟸ First_ANDID_of_OR(work_or_node);
      const ANDID last_and_node_id ⟸ (first_and_node_id + and_count_of_or) − 1;
      ANDID *const order_base ⟸ marpa_obs_start(obs, sizeof(ANDID) * ((size_t)
          and_count_of_or + 1), ALIGNOF(ANDID));
      ANDID *order ⟸ order_base + 1;
```

```
      ANDID and_node_id;
      bocage_was_reordered ⇐ 1;
      for (and_node_id ⇐ first_and_node_id; and_node_id ≤ last_and_node_id;
            and_node_id++) {
        const AND and_node ⇐ and_nodes + and_node_id;
        int and_node_rank;
        ⟨ Set and_node_rank from and_node 1002 ⟩
        if (and_node_rank > high_rank_so_far) {
          order ⇐ order_base + 1;
          high_rank_so_far ⇐ and_node_rank;
        }
        if (and_node_rank ≥ high_rank_so_far) *order++ ⇐ and_node_id;
      }
      {
        int final_count ⇐ (int)(order − order_base) − 1;
        *order_base ⇐ final_count;
        marpa_obs_confirm_fast(obs, (int) sizeof(ANDID) * (final_count + 1));
        and_node_orderings[or_node_id] ⇐ marpa_obs_finish(obs);
      }
    }
  }
```

This code is used in section 1000.

1002. ⟨ Set and_node_rank from and_node 1002 ⟩ ≡

```
  {
    const OR cause_or ⇐ Cause_OR_of_AND(and_node);
    if (OR_is_Token(cause_or)) {
      const NSYID nsy_id ⇐ NSYID_of_OR(cause_or);
      and_node_rank ⇐ Rank_of_NSY(NSY_by_ID(nsy_id));
    }
    else {
      and_node_rank ⇐ Rank_of_IRL(IRL_of_OR(cause_or));
    }
  }
```

This code is used in sections 1001 and 1003.

1003. ⟨ Sort bocage for "rank by rule" 1003 ⟩ ≡

```
  {
    const AND and_nodes ⇐ ANDs_of_B(b);
    const int or_node_count_of_b ⇐ OR_Count_of_B(b);
    const int and_node_count_of_b ⇐ AND_Count_of_B(b);
    int or_node_id ⇐ 0;
    int *rank_by_and_id ⇐ marpa_new(int, and_node_count_of_b);
    int and_node_id;
```

```
    for (and_node_id ⟸ 0; and_node_id < and_node_count_of_b; and_node_id++) {
      const AND and_node ⟸ and_nodes + and_node_id;
      int and_node_rank;
      ⟨ Set and_node_rank from and_node 1002 ⟩
      rank_by_and_id[and_node_id] ⟸ and_node_rank;
    }
    while (or_node_id < or_node_count_of_b) {
      const OR work_or_node ⟸ OR_of_B_by_ID(b, or_node_id);
      const ANDID and_count_of_or ⟸ AND_Count_of_OR(work_or_node);
      ⟨ Sort work_or_node for "rank by rule" 1004 ⟩
      or_node_id++;
    }
    my_free(rank_by_and_id);
  }
```

This code is used in section 999.

1004. An insertion sort is used here, which is $O(n^2)$. The average case (and the root mean square case) in practice will be small number, and this is probably optimal in those terms. Note that none of my complexity claims includes the ranking of ambiguous parses – that is "extra".

For the and-node ranks, I create an array the size of the bocage's and-node count. I could arrange, with some trouble, to just create one the size of the maximum and-node count per or-node. But there seems to be no advantage of any kind gained for the trouble. First, it does not help the worst case. Second, in practice, it does not help with memory issues, because an array of this size will be created with the tree iterator, so I am not establishing a memory "high water mark", and in that sense the space is "free". And third, computationally, pre-computing the and-node ranks is fast and easy, so I am gaining real speed and code-size savings in exchange for the space.

```
⟨ Sort work_or_node for "rank by rule" 1004 ⟩ ≡
  {
    if (and_count_of_or > 1) {
      const ANDID first_and_node_id ⟸ First_ANDID_of_OR(work_or_node);
      ANDID *const order_base ⟸ marpa_obs_new(obs, ANDID, and_count_of_or+1);
      ANDID *order ⟸ order_base + 1;
      int nodes_inserted_so_far;
      bocage_was_reordered ⟸ 1;
      and_node_orderings[or_node_id] ⟸ order_base;
      *order_base ⟸ and_count_of_or;
      for (nodes_inserted_so_far ⟸ 0; nodes_inserted_so_far < and_count_of_or;
            nodes_inserted_so_far++) {
        const ANDID new_and_node_id ⟸ first_and_node_id +
            nodes_inserted_so_far;
        int pre_insertion_ix ⟸ nodes_inserted_so_far − 1;
        while (pre_insertion_ix ≥ 0) {
```

```
          if (rank_by_and_id[new_and_node_id] ≤
                rank_by_and_id[order[pre_insertion_ix]]) break;
          order[pre_insertion_ix + 1] ⟸ order[pre_insertion_ix];
          pre_insertion_ix−−;
        }
        order[pre_insertion_ix + 1] ⟸ new_and_node_id;
      }
    }
  }
```

This code is used in section 1003.

1005. ⟨ Initialize obs and and_node_orderings 1005 ⟩ ≡

```
  {
    int and_id;
    const int and_count_of_r ⟸ AND_Count_of_B(b);
    obs ⟸ OBS_of_O(o) ⟸ marpa_obs_init;
    o→t_and_node_orderings ⟸ and_node_orderings ⟸ marpa_obs_new(obs, ANDID
        ∗, and_count_of_r);
    for (and_id ⟸ 0; and_id < and_count_of_r; and_id++) {
      and_node_orderings[and_id] ⟸ (ANDID ∗) Λ;
    }
  }
```

This code is used in section 999.

1006. Check that ix is the index of a valid and-node in or_node.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE ANDID and_order_ix_is_valid(ORDER o, OR or_node, int ix)
  {
    if (ix ≥ AND_Count_of_OR(or_node)) return 0;
    if (¬O_is_Default(o)) {
      ANDID ∗∗const and_node_orderings ⟸ o→t_and_node_orderings;
      ORID or_node_id ⟸ ID_of_OR(or_node);
      ANDID ∗ordering ⟸ and_node_orderings[or_node_id];
      if (ordering) {
        int length ⟸ ordering[0];
        if (ix ≥ length) return 0;
      }
    }
    return 1;
  }
```

1007. Get the `ix`'th and-node of an or-node. It is up to the caller to ensure that `ix` is valid.

```
⟨ Function definitions 41 ⟩ +≡
  PRIVATE ANDID and_order_get(ORDER o, OR or_node, int ix)
  {
    if (¬O_is_Default(o)) {
      ANDID **const and_node_orderings ⇐ o→t_and_node_orderings;
      ORID or_node_id ⇐ ID_of_OR(or_node);
      ANDID *ordering ⇐ and_node_orderings[or_node_id];
      if (ordering) return ordering[1 + ix];
    }
    return First_ANDID_of_OR(or_node) + ix;
  }
```

1008.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_And_Node_ID _marpa_o_and_order_get(Marpa_Order o, Marpa_Or_Node_ID
          or_node_id, int ix)
  {
    OR or_node;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack order objects 984 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Check or_node_id 1316 ⟩
    ⟨ Set or_node or fail 1317 ⟩
    if (ix < 0) {
      MARPA_ERROR(MARPA_ERR_ANDIX_NEGATIVE);
      return failure_indicator;
    }
    if (¬and_order_ix_is_valid(o, or_node, ix)) return −1;
    return and_order_get(o, or_node, ix);
  }
```

1009. Nook (NOOK) code.

1010. In Marpa, a nook is any node of a parse tree. The usual term is "node", but within Marpa, the word "node" is already heavily overloaded. So what most texts call "tree nodes" are here called "nooks". "Nook" can be thought of as a pun on both "node" and "fork".

1011. For valuation, we need an and-node. The and-node is not kept explicitly. Instead, so we can iterate the trees more readily, we keep an or-node and a choice, and these imply the and-node.

1012. Also, for the purposes of iterating the tree which contains the nooks, we track the parent nook of each nook, whether the current nook is the predecessor or cause of its parent, whether the predecessor of the current nook has been expanded, and whether the cause of the current nook has been expanded.

1013. ⟨ Public typedefs 91 ⟩ +≡

```
typedef int Marpa_Nook_ID;
```

1014. ⟨ Private typedefs 49 ⟩ +≡

```
typedef Marpa_Nook_ID NOOKID;
```

1015. ⟨ Private incomplete structures 107 ⟩ +≡

```
struct s_nook;
typedef struct s_nook *NOOK;
```

1016.

```
#define OR_of_NOOK(nook) ((nook)→t_or_node)
#define Choice_of_NOOK(nook) ((nook)→t_choice)
#define Parent_of_NOOK(nook) ((nook)→t_parent)
#define NOOK_Cause_is_Expanded(nook) ((nook)→t_is_cause_ready)
#define NOOK_is_Cause(nook) ((nook)→t_is_cause_of_parent)
#define NOOK_Predecessor_is_Expanded(nook) ((nook)→t_is_predecessor_ready)
#define NOOK_is_Predecessor(nook) ((nook)→t_is_predecessor_of_parent)
```

⟨ NOOK structure 1016 ⟩ ≡

```
struct s_nook {
  OR t_or_node;
  int t_choice;
  NOOKID t_parent;
  BITFIELD t_is_cause_ready:1;
  BITFIELD t_is_predecessor_ready:1;
  BITFIELD t_is_cause_of_parent:1;
  BITFIELD t_is_predecessor_of_parent:1;
};
typedef struct s_nook NOOK_Object;
```

This code is used in section 1022.

1017. Parse tree (T, TREE) code. In this document, when it makes sense in context, the term "tree" means a parse tree. Trees are, of course, a very common data structure, and are used for all sorts of things. But the most important trees in Marpa's universe are its parse trees.

1018. Marpa's parse trees are produced by iterating the Marpa bocage. Therefore, Marpa parse trees are also bocage iterators.

1019. A tree is a stack whose bottom is the top of the tree. The tree is in depth-first, cause-then-predecessor order. Because it is in cause-then-predecessor order, it is lexically in right-to-left order.

1020. ⟨ Public incomplete structures 47 ⟩ +≡

```
struct marpa_tree;
typedef struct marpa_tree *Marpa_Tree;
```

1021. ⟨ Private incomplete structures 107 ⟩ +≡

```
typedef Marpa_Tree TREE;
```

1022. An exhausted bocage iterator (or parse tree) does not need a worklist or a stack, so they are destroyed. If the bocage iterator has a parse count, but no stack, it is exhausted.

```
#define Size_of_TREE(tree) MARPA_DSTACK_LENGTH((tree)→t_nook_stack)
#define NOOK_of_TREE_by_IX(tree, nook_id)
        MARPA_DSTACK_INDEX((tree)→t_nook_stack, NOOK_Object, nook_id)
#define O_of_T(t) ((t)→t_order)
```

⟨ Private structures 48 ⟩ +≡

```
  ⟨ NOOK structure 1016 ⟩
  ⟨ VALUE structure 1071 ⟩
  struct marpa_tree {
    MARPA_DSTACK_DECLARE(t_nook_stack);
    MARPA_DSTACK_DECLARE(t_nook_worklist);
    Bit_Vector t_or_node_in_use;
    Marpa_Order t_order;
    ⟨ Int aligned tree elements 1028 ⟩
    ⟨ Bit aligned tree elements 1041 ⟩
    int t_parse_count;
  };
```

1023. ⟨ Unpack tree objects 1023 ⟩ ≡

```
  ORDER o ⟸ O_of_T(t);
  ⟨ Unpack order objects 984 ⟩;
```

This code is used in sections 1039, 1066, 1083, 1090, 1342, 1343, 1344, 1345, 1346, 1347, and 1348.

1024. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void tree_exhaust(TREE t)
{
  if (MARPA_DSTACK_IS_INITIALIZED(t→t_nook_stack)) {
    MARPA_DSTACK_DESTROY(t→t_nook_stack);
    MARPA_DSTACK_SAFE(t→t_nook_stack);
  }
  if (MARPA_DSTACK_IS_INITIALIZED(t→t_nook_worklist)) {
    MARPA_DSTACK_DESTROY(t→t_nook_worklist);
    MARPA_DSTACK_SAFE(t→t_nook_worklist);
  }
  bv_free(t→t_or_node_in_use);
  t→t_or_node_in_use ⟸ Λ;
  T_is_Exhausted(t) ⟸ 1;
}
```

1025. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Tree marpa_t_new(Marpa_Order o)
{
  ⟨ Return Λ on failure 1228 ⟩
  TREE t;
  ⟨ Unpack order objects 984 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  t ⟸ my_malloc(sizeof (*t));
  O_of_T(t) ⟸ o;
  order_ref(o);
  O_is_Frozen(o) ⟸ 1;
  ⟨ Pre-initialize tree elements 1042 ⟩
  ⟨ Initialize tree elements 1026 ⟩
  return t;
}
```

1026. ⟨ Initialize tree elements 1026 ⟩ ≡

```
{
  t→t_parse_count ⟸ 0;
  if (O_is_Nulling(o)) {
    T_is_Nulling(t) ⟸ 1;
    t→t_or_node_in_use ⟸ Λ;
    MARPA_DSTACK_SAFE(t→t_nook_stack);
    MARPA_DSTACK_SAFE(t→t_nook_worklist);
  }
  else {
    const int and_count ⟸ AND_Count_of_B(b);
    const int or_count ⟸ OR_Count_of_B(b);
```

```
    T_is_Nulling(t) ⟸ 0;
    t→t_or_node_in_use ⟸ bv_create(or_count);
    MARPA_DSTACK_INIT(t→t_nook_stack, NOOK_Object, and_count);
    MARPA_DSTACK_INIT(t→t_nook_worklist, NOOKID, and_count);
  }
}
```

See also sections 1029 and 1036.

This code is used in section 1025.

1027. Reference counting and destructors.

1028. ⟨ Int aligned tree elements 1028 ⟩ ≡

```
int t_ref_count;
```

See also section 1035.

This code is used in section 1022.

1029. ⟨ Initialize tree elements 1026 ⟩ +≡

```
t→t_ref_count ⟸ 1;
```

1030. Decrement the tree reference count.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void tree_unref(TREE t)
{
  MARPA_ASSERT(t→t_ref_count > 0)t→t_ref_count−−;
  if (t→t_ref_count ≤ 0) {
    tree_free(t);
  }
}
void marpa_t_unref(Marpa_Tree t)
{
  tree_unref(t);
}
```

1031. Increment the tree reference count.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE TREE tree_ref(TREE t)
{
  MARPA_ASSERT(t→t_ref_count > 0)t→t_ref_count++;
  return t;
}
Marpa_Tree marpa_t_ref(Marpa_Tree t)
{
  return tree_ref(t);
}
```

1032. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void tree_free(TREE t)
{
  order_unref(O_of_T(t));
  tree_exhaust(t);
  my_free(t);
}
```

1033. Tree pause counting. Trees referenced by an active *VALUE* object cannot be iterated for the lifetime of that *VALUE* object. This is enforced by "pausing" the tree. Because there may be multiple *VALUE* objects for each *TREE* object, a pause counter is used.

1034. The *TREE* object's pause counter works much the same as a reference counter. And the two are tied together. Every time the pause counter is incremented, the *TREE* object's reference counter is also incremented. Similarly, every time the pause counter is decremented, the *TREE* object's reference counter is also decremented. For this reason, it is important that every tree "pause" be matched with a "tree unpause"

1035. "Pausing" is used because the expected use of multiple *VALUE* objects is to evaluation a single tree instance in multiple ways — *VALUE* objects are not expected to need to live into the next iteration of the *TREE* object. If a more complex relationship between *TREE* objects and *VALUE* objects becomes desirable, a cloning mechanism could be introduced. At this point, *TREE* objects are iterated directly for efficiency — copying the *TREE* iterator to a tree instance would impose an overhead, one which adds absolutely no value for most applications.

```
#define T_is_Paused(t) ((t)->t_pause_counter > 0)
```

⟨ Int aligned tree elements 1028 ⟩ +≡

```
  int t_pause_counter;
```

1036. ⟨ Initialize tree elements 1026 ⟩ +≡

```
  t->t_pause_counter = 0;
```

1037. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void tree_pause(TREE t)
{
  MARPA_ASSERT(t->t_pause_counter >= 0);
  MARPA_ASSERT(t->t_ref_count >= t->t_pause_counter);
  t->t_pause_counter++;
  tree_ref(t);
}
```

1038. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void tree_unpause(TREE t)
{
  MARPA_ASSERT(t→t_pause_counter > 0);
  MARPA_ASSERT(t→t_ref_count ≥ t→t_pause_counter);
  t→t_pause_counter−−;
  tree_unref(t);
}
```

1039. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_t_next(Marpa_Tree t)
{
  ⟨ Return −2 on failure 1229 ⟩
  const int termination_indicator ⇐ −1;
  int is_first_tree_attempt ⇐ (t→t_parse_count < 1);
  ⟨ Unpack tree objects 1023 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  if (T_is_Paused(t)) {
    MARPA_ERROR(MARPA_ERR_TREE_PAUSED);
    return failure_indicator;
  }
  if (T_is_Exhausted(t)) {
    MARPA_ERROR(MARPA_ERR_TREE_EXHAUSTED);
    return termination_indicator;
  }
  if (T_is_Nulling(t)) {
    if (is_first_tree_attempt) {
      t→t_parse_count++;
      return 0;
    }
    else {
      goto TREE_IS_EXHAUSTED;
    }
  }
  while (1) {
    const AND ands_of_b ⇐ ANDs_of_B(b);
    if (is_first_tree_attempt) {
      is_first_tree_attempt ⇐ 0;
      ⟨ Initialize the tree iterator 1048 ⟩
    }
    else {
      ⟨ Start a new iteration of the tree 1049 ⟩
    }
    ⟨ Finish tree if possible 1053 ⟩
```

```
    }
  TREE_IS_FINISHED: ;
    t→t_parse_count++;
    return MARPA_DSTACK_LENGTH(t→t_nook_stack);
  TREE_IS_EXHAUSTED: ;
    tree_exhaust(t);
    MARPA_ERROR(MARPA_ERR_TREE_EXHAUSTED);
    return termination_indicator;
  }
```

1040. Tree is exhausted?. Is this tree for an exhausted parse?

```
#define T_is_Exhausted(t) ((t)→t_is_exhausted)
```

1041. ⟨ Bit aligned tree elements 1041 ⟩ ≡

```
  BITFIELD t_is_exhausted:1;
```

See also section 1044.

This code is used in section 1022.

1042. ⟨ Pre-initialize tree elements 1042 ⟩ ≡

```
  T_is_Exhausted(t) ⟸ 0;
```

This code is used in section 1025.

1043. Tree is nulling?. Is this tree for a nulling parse?

```
#define T_is_Nulling(t) ((t)→t_is_nulling)
```

1044. ⟨ Bit aligned tree elements 1041 ⟩ +≡

```
  BITFIELD t_is_nulling:1;
```

1045. Claiming and releasing or-nodes. To avoid cycles, the same or node is not allowed to occur twice in the parse tree, unless it is zero-length. A boolean vector, accessed by these functions, enforces this.

1046. Try to claim the or-node. If it was already claimed, return 0, otherwise claim it (that is, set the bit) and return 1.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE int tree_or_node_try(TREE tree, ORID or_node_id)
  {
    return ¬bv_bit_test_then_set(tree→t_or_node_in_use, or_node_id);
  }
```

1047. Release the or-node by unsetting its bit. This may be called on unclaimed or-nodes, in which case it a no-op. which is necessary b

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void tree_or_node_release(TREE tree, ORID or_node_id)
  {
    bv_bit_clear(tree→t_or_node_in_use, or_node_id);
  }
```

1048. Iterating the tree. The initial or-node is the root or-node, whose rule is the augmented start rule of the grammar. The augment rule has a dedicated LHS symbol, which does not appear on any RHS, so the augment rule, and therefore the root or-node, cannot be part of cycle.

⟨ Initialize the tree iterator 1048 ⟩ ≡

```
{
  MARPA_DEBUG1("Initialize␣tree");
  ORID root_or_id ⟸ Top_ORID_of_B(b);
  OR root_or_node ⟸ OR_of_B_by_ID(b, root_or_id);
  NOOK nook;    /* Due to skipping, it is possible for even the top or-node to have
      no valid choices, in which case there is no parse */
  const int choice ⟸ 0;
  if (¬and_order_ix_is_valid(o, root_or_node, choice)) goto TREE_IS_EXHAUSTED;
  nook ⟸ MARPA_DSTACK_PUSH(t→t_nook_stack, NOOK_Object);
  tree_or_node_try(t, root_or_id);    /* Empty stack, so cannot fail */
  OR_of_NOOK(nook) ⟸ root_or_node;
  Choice_of_NOOK(nook) ⟸ choice;
  Parent_of_NOOK(nook) ⟸ −1;
  NOOK_Cause_is_Expanded(nook) ⟸ 0;
  NOOK_is_Cause(nook) ⟸ 0;
  NOOK_Predecessor_is_Expanded(nook) ⟸ 0;
  NOOK_is_Predecessor(nook) ⟸ 0;
}
```

This code is cited in section 1062.

This code is used in section 1039.

1049. Look for a nook to iterate. If there is one, set it to the next choice. Otherwise, the tree is exhausted.

⟨ Start a new iteration of the tree 1049 ⟩ ≡

```
{
  MARPA_DEBUG1("Start␣new␣iteration␣of␣tree");
  while (1) {
    OR iteration_candidate_or_node;
    const NOOK iteration_candidate ⟸ MARPA_DSTACK_TOP(t→t_nook_stack,
        NOOK_Object);
    int choice;
    if (¬iteration_candidate) break;
    iteration_candidate_or_node ⟸ OR_of_NOOK(iteration_candidate);
    choice ⟸ Choice_of_NOOK(iteration_candidate) + 1;
    MARPA_ASSERT(choice > 0);
    if (and_order_ix_is_valid(o, iteration_candidate_or_node, choice)) {
        /* We have found a nook we can iterate. Set the new choice, dirty the child
          bits in the current working nook, and break out of the loop. */
      Choice_of_NOOK(iteration_candidate) ⟸ choice;
```

```
          NOOK_Cause_is_Expanded(iteration_candidate) ⇐ 0;
          NOOK_Predecessor_is_Expanded(iteration_candidate) ⇐ 0;
          break;
        }
        {    /* Dirty the corresponding bit in the parent, then pop the nook */
          const int parent_nook_ix ⇐ Parent_of_NOOK(iteration_candidate);
          if (parent_nook_ix ≥ 0) {
            NOOK parent_nook ⇐ NOOK_of_TREE_by_IX(t, parent_nook_ix);
            if (NOOK_is_Cause(iteration_candidate)) {
              NOOK_Cause_is_Expanded(parent_nook) ⇐ 0;
            }
            if (NOOK_is_Predecessor(iteration_candidate)) {
              NOOK_Predecessor_is_Expanded(parent_nook) ⇐ 0;
            }
          }    /* Continue with the next item on the stack */
          tree_or_node_release(t, ID_of_OR(iteration_candidate_or_node));
          MARPA_DSTACK_POP(t→t_nook_stack, NOOK_Object);
        }
      }
      if (Size_of_T(t) ≤ 0) goto TREE_IS_EXHAUSTED;
    }
```

This code is used in section 1039.

1050. Once we have the initial segment of a tree we want to interate, we "finish" it. Finishing a tree involves building it out, taking choice 0 for every or-node. (Any other choices will be encountered when we iterate.

1051. The worklist is a list of potentially "dirty" nooks, either the predecessor or cause of which may need to be expanded. It is harmless to have "clean" nooks in the worklist — the finishing code does nothing to a clean nook except pop it off the work list.

1052. We might consider further optimizing by checking every nook to see if it is "dirty" before pushing it onto the worklist, but we must make the same tests when the nook is popped of the worklist, in order to process it. So it's a question of whether the cost of a push and pop. outweighs the cost of duplicating the "dirty" bit tests.

1053. ⟨ Finish tree if possible 1053 ⟩ ≡

```
  {
    {
      const int stack_length ⇐ Size_of_T(t);
      MARPA_DEBUG2("Finishing␣tree,␣size␣=␣%ld", (long) stack_length);
      int i;
      /* Clear the worklist, then copy the entire remaining tree onto it. */
      MARPA_DSTACK_CLEAR(t→t_nook_worklist);
```

```
    for (i ⟸ 0; i < stack_length; i++) {
      *(MARPA_DSTACK_PUSH(t→t_nook_worklist, NOOKID)) ⟸ i;
    }
  }
  while (1) {
    NOOKID work_nook_id;
    NOOK work_nook;
    ANDID work_and_node_id;
    AND work_and_node;
    OR work_or_node;
    OR child_or_node ⟸ Λ;
    int choice;
    int child_is_cause ⟸ 0;
    int child_is_predecessor ⟸ 0;
    if (MARPA_DSTACK_LENGTH(t→t_nook_worklist) ≤ 0) {
      goto TREE_IS_FINISHED;
    }
    work_nook_id ⟸ *MARPA_DSTACK_TOP(t→t_nook_worklist, NOOKID);
    work_nook ⟸ NOOK_of_TREE_by_IX(t, work_nook_id);
    work_or_node ⟸ OR_of_NOOK(work_nook);
    work_and_node_id ⟸ and_order_get(o, work_or_node,
        Choice_of_NOOK(work_nook));
    MARPA_DEBUG5("Work␣node␣is␣%ld,␣OR=%ld,␣choice=%ld,␣AND=%ld\n", (long)
        work_nook_id, (long) ID_of_OR(work_or_node),
        (long) Choice_of_NOOK(work_nook), (long) work_and_node_id);
    work_and_node ⟸ ands_of_b + work_and_node_id;
    do {
      if (¬NOOK_Cause_is_Expanded(work_nook)) {
        const OR cause_or_node ⟸ Cause_OR_of_AND(work_and_node);
        if (¬OR_is_Token(cause_or_node)) {
          child_or_node ⟸ cause_or_node;
          child_is_cause ⟸ 1;
          MARPA_DEBUG3("Work␣nook␣ID␣is␣%ld,␣child␣OR␣%ld␣is␣cause", (long)
              work_nook_id, ID_of_OR(child_or_node));
          break;
        }
      }
      NOOK_Cause_is_Expanded(work_nook) ⟸ 1;
      if (¬NOOK_Predecessor_is_Expanded(work_nook)) {
        child_or_node ⟸ Predecessor_OR_of_AND(work_and_node);
        if (child_or_node) {
          child_is_predecessor ⟸ 1;
          MARPA_DEBUG3("Work␣nook␣ID␣is␣%ld,␣child␣OR␣%ld␣is␣predecessor",
              (long) work_nook_id, ID_of_OR(child_or_node));
```

```
          break;
        }
      }
      NOOK_Predecessor_is_Expanded(work_nook) ⟸ 1;
      MARPA_DSTACK_POP(t→t_nook_worklist, NOOKID);
      goto NEXT_NOOK_ON_WORKLIST;
    } while (0);
    ⟨If tree has cycle, go to NEXT_TREE 1063⟩choice ⟸ 0;
    if (¬and_order_ix_is_valid(o, child_or_node, choice)) goto NEXT_TREE;
    MARPA_DEBUG2("After␣check␣for␣valid␣and␣order␣ix,␣node=%lx", (long)
        child_or_node);
    ⟨Add new nook to tree 1064⟩;
  NEXT_NOOK_ON_WORKLIST: ;
  }
NEXT_TREE: ;
}
```

This code is used in section 1039.

1054. We check for or-node cycles here. It is necessary to demonstrate carefully that our logic eliminates all, and only, the or-nodes which lead to cycles.

1055. Lemma: Non-zero duplicate implies cycle. If the length of an or-node is non-zero and it has a duplicate in the tree, then that or-node is part of a cycle.

Proof: Let an or-node appear twice in the tree, at instances *i1* and *i2*. Since the or-node has non-zero length then its dotted rule has the form $A ::= \alpha \bullet \beta$, where

- α is a sentential form of one or more symbols which derives *terms*, and
- *terms* is a string of terminals.

Since the or-node is not zero-length, *terms* contains at least one non-null symbol, call it t. t has a fixed location in the lexical input string, call it x.

⟨Lemma: Non-zero duplicate implies cycle 1055⟩ ≡

This code is cited in sections 1056, 1057, and 1060.

This code is used in section 1060.

1056. [Continuing ⟨Lemma: Non-zero duplicate implies cycle 1055⟩] Either *i1* derives *i2* or *i2* derives *i1*. If that were not the case then t would appear at two distinct locations, both of which must be location x, which is nonsensical.

1057. [Continuing ⟨Lemma: Non-zero duplicate implies cycle 1055⟩] Assume without loss of generality that *i1* derives *i2*. The same logic which caused the derivation from *i1* to *i2*, will cause this derivation to be repeated an arbitrary number of times, causing an or-node cycle. QED for ⟨Lemma: Non-zero duplicate implies cycle 1055⟩.

1058. Lemma: Cycle implies duplicate. If an or-node is part of a cycle, then it has a duplicate in the tree.

Proof: If an or-node never produces a duplicate in the tree, by definition there is no cycle for this or-node in that tree.

⟨ Lemma: Cycle implies duplicate 1058 ⟩ ≡

This code is cited in section 1060.

This code is used in section 1060.

1059. Lemma: Cycle implies non-zero. If an or-node is part of a cycle, then the length of the or-node is non-zero.

Proof: We will show the contrapositive, that a zero-length or-node does not produce a cycle. To do this we show that a zero-length or-node is a "dead-end' in terms of derivation. An or-node derives other or-nodes with through its predecessor or its cause. A zero-length or-node has no predecessor. (In theory a predicted dotted rule can be seen as the predecessor, but predecessors are semantically inert, and derivationally dead-ends, so we do not bother with them.) The cause of a zero-length or-node must be zero length. The only zero-length causes are nulling symbols, and these are derivational dead-ends. Thus a derivation from a zero-length or-node never takes us back to an or-node. QED for ⟨ Lemma: Cycle implies non-zero 1059 ⟩.

⟨ Lemma: Cycle implies non-zero 1059 ⟩ ≡

This code is cited in sections 1059 and 1060.

This code is used in section 1060.

1060. Theorem: Non-zero and duplicate iff cycle. Theorem: The length of an or-node is non-zero and it has a duplicate in the tree, iff that or-node is part of a cycle.

Proof: This theorem follows from ⟨ Lemma: Non-zero duplicate implies cycle 1055 ⟩, ⟨ Lemma: Cycle implies duplicate 1058 ⟩ and ⟨ Lemma: Cycle implies non-zero 1059 ⟩. QED.

⟨ Theorem: Non-zero and duplicate iff cycle 1060 ⟩ ≡
  ⟨ Lemma: Non-zero duplicate implies cycle 1055 ⟩
  ⟨ Lemma: Cycle implies duplicate 1058 ⟩
  ⟨ Lemma: Cycle implies non-zero 1059 ⟩

This code is cited in sections 1061 and 1062.

This code is used in sections 1061 and 1062.

1061. Theorem: Or-node cycle elimination is consistent. Or-node cycle elimination is consistent, that is, every tree pruned because of an or-node cycle actually does contain an or-node cycle.

Proof: All pruning for or-node cycles occurs in ⟨ If tree has cycle, go to `NEXT_TREE` 1063 ⟩. This proof follows directly from that fact and ⟨ Theorem: Non-zero and duplicate iff cycle 1060 ⟩. QED.

⟨ Theorem: Or-node cycle elimination is consistent 1061 ⟩ ≡
  ⟨ Theorem: Non-zero and duplicate iff cycle 1060 ⟩

This code is used in section 1063.

1062. Theorem: Or-node cycle elimination is complete. Or-node cycle elimination is complete, that is, every tree than contains an or-node cycle is pruned.

Proof: Or-nodes are added to the trees in either ⟨ Initialize the tree iterator 1048 ⟩ or ⟨ Add new nook to tree 1064 ⟩. Only the root or-node is added in⟨ Initialize the tree iterator 1048 ⟩, and this is never part of an or-node cycle for the reasons given in ⟨ Initialize the tree iterator 1048 ⟩.

The code in ⟨ Add new nook to tree 1064 ⟩ is guarded by ⟨ If tree has cycle, go to NEXT_TREE 1063 ⟩. By ⟨ Theorem: Non-zero and duplicate iff cycle 1060 ⟩, ⟨ If tree has cycle, go to NEXT_TREE 1063 ⟩ prunes every tree containing and or-node cycle. It follow that or-node cycle elimination is complete. QED.

⟨ Theorem: Or-node cycle elimination is complete 1062 ⟩ ≡
⟨ Theorem: Non-zero and duplicate iff cycle 1060 ⟩

This code is used in section 1063.

1063. ⟨ If tree has cycle, go to NEXT_TREE 1063 ⟩ ≡

```
{
  ⟨ Theorem: Or-node cycle elimination is consistent 1061 ⟩
  ⟨ Theorem: Or-node cycle elimination is complete 1062 ⟩
  MARPA_DEBUG3("Before check for duplicate or node, node=%lx ID=%ld", (long)
      child_or_node, (long) ID_of_OR(child_or_node));  /* If the child or-node is
      not of zero length, try to claim it. Otherwise, reject the tree. */
  if (Length_of_OR(child_or_node) ∧ ¬tree_or_node_try(t, ID_of_OR(child_or_node)))
    goto NEXT_TREE;
  MARPA_DEBUG3("After check for duplicate or node, node=%lx ID=%ld", (long)
      child_or_node, (long) ID_of_OR(child_or_node));
}
```

This code is cited in sections 1061 and 1062.

This code is used in section 1053.

1064. ⟨ Add new nook to tree 1064 ⟩ ≡

```
{
  NOOKID new_nook_id ⟸ Size_of_T(t);
  NOOK new_nook ⟸ MARPA_DSTACK_PUSH(t→t_nook_stack, NOOK_Object);
  *(MARPA_DSTACK_PUSH(t→t_nook_worklist, NOOKID)) ⟸ new_nook_id;
  work_nook ⟸ NOOK_of_TREE_by_IX(t, work_nook_id);
    /* Refresh work_nook because push to dynamic stack may have moved it */
  Parent_of_NOOK(new_nook) ⟸ work_nook_id;
  Choice_of_NOOK(new_nook) ⟸ choice;
  OR_of_NOOK(new_nook) ⟸ child_or_node;
  MARPA_DEBUG5("New node is %ld, OR=%ld, choice=%ld, AND=%ld\n", (long)
      new_nook_id, (long) ID_of_OR(child_or_node), (long) choice, (long)
      and_order_get(o, child_or_node, choice));
  NOOK_Cause_is_Expanded(new_nook) ⟸ 0;
  if ((NOOK_is_Cause(new_nook) ⟸ Boolean(child_is_cause))) {
    NOOK_Cause_is_Expanded(work_nook) ⟸ 1;
```

```
    }
    NOOK_Predecessor_is_Expanded(new_nook) ⟸ 0;
    if ((NOOK_is_Predecessor(new_nook) ⟸ Boolean(child_is_predecessor))) {
      NOOK_Predecessor_is_Expanded(work_nook) ⟸ 1;
    }
  }
```

This code is cited in section 1062.

This code is used in section 1053.

1065. Accessors.

⟨ Function definitions 41 ⟩ +≡

```
  int marpa_t_parse_count(Marpa_Tree t)
  {
    return t→t_parse_count;
  }
```

1066.

```
#define Size_of_T(t) MARPA_DSTACK_LENGTH((t)→t_nook_stack)
```

⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_t_size(Marpa_Tree t)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack tree objects 1023 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    if (T_is_Exhausted(t)) {
      MARPA_ERROR(MARPA_ERR_TREE_EXHAUSTED);
      return failure_indicator;
    }
    if (T_is_Nulling(t)) return 0;
    return Size_of_T(t);
  }
```

1067. Evaluation (V, VALUE) code.

1068. This code helps compute a value for a parse tree. I say "helps" because evaluating a parse tree involves semantics, and libmarpa has only limited knowledge of the semantics. This code is really just to assist the higher level in keeping an evaluation stack.

The main reason to have evaluation logic in libmarpa at all is to hide libmarpa's internal rewrites from the semantics. If it were not for that, it would probably be just as easy to provide a parse tree to the higher level and let them decide how to evaluate it.

```
⟨ Public incomplete structures 47 ⟩ +≡
  struct marpa_value;
  typedef struct marpa_value *Marpa_Value;
```

1069.
```
⟨ Private incomplete structures 107 ⟩ +≡
  typedef struct s_value *VALUE;
```

1070. This structure tracks the top of the evaluation stack, but does **not** maintain the actual evaluation stack — that is left for the upper layers to do. It does, however, mantain a stack of the counts of symbols in the original (or "virtual") rules. This enables libmarpa to make the rewriting of the grammar invisible to the semantics.

```
#define Next_Value_Type_of_V(val) ((val)→t_next_value_type)
#define V_is_Active(val) (Next_Value_Type_of_V(val) ≠ MARPA_STEP_INACTIVE)
#define T_of_V(v) ((v)→t_tree)
```

1071.
```
⟨ VALUE structure 1071 ⟩ ≡
  struct s_value {
    struct marpa_value public;
    Marpa_Tree t_tree;
    ⟨ Widely aligned value elements 1075 ⟩
    ⟨ Int aligned value elements 1085 ⟩
    int t_token_type;
    int t_next_value_type;
    ⟨ Bit aligned value elements 1092 ⟩
  };
```

This code is used in section 1022.

1072. Public data.

```
⟨ Public structures 44 ⟩ +≡
  struct marpa_value {
    Marpa_Step_Typet_step_type;
    Marpa_Symbol_ID t_token_id;
    int t_token_value;
    Marpa_Rule_ID t_rule_id;
    int t_arg_0;
    int t_arg_n;
```

```
    int t_result;
    Marpa_Earley_Set_ID t_token_start_ys_id;
    Marpa_Earley_Set_ID t_rule_start_ys_id;
    Marpa_Earley_Set_ID t_ys_id;
  };
```

1073. The public defines use "es" instead of "ys" for Earley set.

⟨ Public defines 109 ⟩ +≡

```
#define marpa_v_step_type(v) ((v)→t_step_type)
#define marpa_v_token(v) ((v)→t_token_id)
#define marpa_v_symbol(v)marpa_v_token (v)
#define marpa_v_token_value(v) ((v)→t_token_value)
#define marpa_v_rule(v) ((v)→t_rule_id)
#define marpa_v_arg_0(v) ((v)→t_arg_0)
#define marpa_v_arg_n(v) ((v)→t_arg_n)
#define marpa_v_result(v) ((v)→t_result)
#define marpa_v_rule_start_es_id(v) ((v)→t_rule_start_ys_id)
#define marpa_v_token_start_es_id(v) ((v)→t_token_start_ys_id)
#define marpa_v_es_id(v) ((v)→t_ys_id)
```

1074. Arg_N_of_V is the current top of stack. Result_of_V is where the result of the next evaluation operation should be placed and, once that is done, will be the new top of stack. If the next evaluation operation is a stack no-op, Result_of_V immediately becomes the new top of stack.

```
#define Step_Type_of_V(val) ((val)→public.t_step_type)
#define XSYID_of_V(val) ((val)→public.t_token_id)
#define RULEID_of_V(val) ((val)→public.t_rule_id)
#define Token_Value_of_V(val) ((val)→public.t_token_value)
#define Token_Type_of_V(val) ((val)→t_token_type)
#define Arg_0_of_V(val) ((val)→public.t_arg_0)
#define Arg_N_of_V(val) ((val)→public.t_arg_n)
#define Result_of_V(val) ((val)→public.t_result)
#define Rule_Start_of_V(val) ((val)→public.t_rule_start_ys_id)
#define Token_Start_of_V(val) ((val)→public.t_token_start_ys_id)
#define YS_ID_of_V(val) ((val)→public.t_ys_id)
```

⟨ Initialize value elements 1074 ⟩ ≡

```
  XSYID_of_V(v) ⟸ −1;
  RULEID_of_V(v) ⟸ −1;
  Token_Value_of_V(v) ⟸ −1;
  Token_Type_of_V(v) ⟸ DUMMY_OR_NODE;
  Arg_0_of_V(v) ⟸ −1;
  Arg_N_of_V(v) ⟸ −1;
  Result_of_V(v) ⟸ −1;
  Rule_Start_of_V(v) ⟸ −1;
```

`Token_Start_of_V`$(v) \Longleftarrow -1$;
`YS_ID_of_V`$(v) \Longleftarrow -1$;

See also sections 1081, 1086, 1093, 1095, 1098, and 1103.

This code is used in section 1083.

1075. The obstack. An obstack with the same lifetime as the valuator.

⟨ Widely aligned value elements 1075 ⟩ ≡
struct `marpa_obstack *t_obs`;

See also sections 1080 and 1102.

This code is used in section 1071.

1076. ⟨ Destroy value obstack 1076 ⟩ ≡
`marpa_obs_free`$(v\to$`t_obs`$)$;

This code is used in section 1089.

1077. Virtual stack.

1078. A dynamic stack is used here instead of a fixed stack for two reasons. First, there are only a few stack moves per call of `marpa_v_step`. Since at least one subroutine call occurs every few virtual stack moves, virtual stack moves are not really within a tight CPU loop. Therefore shaving off the few instructions it takes to check stack size is less important than it is in other places.

1079. Second, the fixed stack, to accomodate the worst case, would have to be many times larger than what will usually be needed. My current best bound on the worst case for virtual stack size is as follows.

The virtual stack only grows once for each virtual rule. To be virtual, a rule must divide into a least two "real" or rewritten, rules, so worst case is half of all applications of real rules grow the virtual stack. The number of applications of real rules is the size of the parse tree, $|\texttt{tree}|$. So, if the fixed stack is sized per tree, it must be $|\texttt{tree}|/2+1$.

1080. I set the initial size of the dynamic stack to be $|\texttt{tree}|/1024$, with a minimum of 1024. 1024 is chosen because in some modern configurations a smaller allocation may require extra work. The purpose of the $|\texttt{tree}|/1024$ is to guarantee that this code is $O(n)$. $|\texttt{tree}|/1024$ is a fixed fraction of the worst case size, so the number of stack reallocations is $O(1)$.

#define `VStack_of_V(val)` ((`val`)$\to$`t_virtual_stack`)

⟨ Widely aligned value elements 1075 ⟩ +≡
`MARPA_DSTACK_DECLARE(t_virtual_stack)`;

1081. ⟨ Initialize value elements 1074 ⟩ +≡
`MARPA_DSTACK_SAFE(VStack_of_V`(v)`)`;

1082. ⟨ Destroy value elements 1082 ⟩ ≡

```
{
  if (_MARPA_LIKELY(MARPA_DSTACK_IS_INITIALIZED(VStack_of_V(v)) ≠ Λ)) {
    MARPA_DSTACK_DESTROY(VStack_of_V(v));
  }
}
```

This code is used in section 1089.

1083. Valuator constructor.

⟨ Function definitions 41 ⟩ +≡

```
  Marpa_Value marpa_v_new(Marpa_Tree t)
  {
    ⟨ Return Λ on failure 1228 ⟩
    ⟨ Unpack tree objects 1023 ⟩;
    ⟨ Fail if fatal error 1249 ⟩
    if (t→t_parse_count ≤ 0) {
      MARPA_ERROR(MARPA_ERR_BEFORE_FIRST_TREE);
      return Λ;
    }
    if (¬T_is_Exhausted(t)) {
      const XSYID xsy_count ⟸ XSY_Count_of_G(g);
      struct marpa_obstack *const obstack ⟸ marpa_obs_init;
      const VALUE v ⟸ marpa_obs_new(obstack, struct s_value, 1);
      v→t_obs ⟸ obstack;
      Step_Type_of_V(v) ⟸ Next_Value_Type_of_V(v) ⟸ MARPA_STEP_INITIAL;
      ⟨ Initialize value elements 1074 ⟩
      tree_pause(t);
      T_of_V(v) ⟸ t;
      if (T_is_Nulling(o)) {
        V_is_Nulling(v) ⟸ 1;
      }
      else {
        const int minimum_stack_size ⟸ (8192/sizeof (int));
        const int initial_stack_size ⟸ MAX(Size_of_TREE(t)/1024,
            minimum_stack_size);
        MARPA_DSTACK_INIT(VStack_of_V(v), int, initial_stack_size);
      }
      return (Marpa_Value) v;
    }
    MARPA_ERROR(MARPA_ERR_TREE_EXHAUSTED);
    return Λ;
  }
```

1084. Reference counting and destructors.

1085. ⟨ Int aligned value elements 1085 ⟩ ≡

```
int t_ref_count;
```

See also section 1097.

This code is used in section 1071.

1086. ⟨ Initialize value elements 1074 ⟩ +≡

```
v→t_ref_count ⟸ 1;
```

1087. Decrement the value reference count.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void value_unref(VALUE v)
{
  MARPA_ASSERT(v→t_ref_count > 0)
  v→t_ref_count --;
  if (v→t_ref_count ≤ 0) {
    value_free(v);
  }
}
void marpa_v_unref(Marpa_Value public_v)
{
  value_unref((VALUE) public_v);
}
```

1088. Increment the value reference count.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE VALUE value_ref(VALUE v)
{
  MARPA_ASSERT(v→t_ref_count > 0)v→t_ref_count ++;
  return v;
}
Marpa_Value marpa_v_ref(Marpa_Value v)
{
  return (Marpa_Value) value_ref((VALUE) v);
}
```

1089. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void value_free(VALUE v)
{
  tree_unpause(T_of_V(v));
  ⟨ Destroy value elements 1082 ⟩
  ⟨ Destroy value obstack 1076 ⟩
}
```

1090. ⟨ Unpack value objects 1090 ⟩ ≡
TREE t ⟸ T_of_V(v);
⟨ Unpack tree objects 1023 ⟩

This code is used in sections 1096, 1099, 1105, 1107, 1108, 1109, 1110, 1112, and 1115.

1091. Valuator is nulling?. Is this valuator for a nulling parse?

#define V_is_Nulling(v) ((v)→t_is_nulling)

1092. ⟨ Bit aligned value elements 1092 ⟩ ≡
BITFIELD t_is_nulling:1;

See also section 1094.

This code is used in section 1071.

1093. ⟨ Initialize value elements 1074 ⟩ +≡
V_is_Nulling(v) ⟸ 0;

1094. Trace valuator?.

#define V_is_Trace(val) ((val)→t_trace)

⟨ Bit aligned value elements 1092 ⟩ +≡
BITFIELD t_trace:1;

1095. ⟨ Initialize value elements 1074 ⟩ +≡
V_is_Trace(v) ⟸ 0;

1096. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_v_trace(Marpa_Value public_v, int flag)
{
  ⟨ Return −2 on failure 1229 ⟩
  const VALUE v ⟸ (VALUE) public_v;
  ⟨ Unpack value objects 1090 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  if (_MARPA_UNLIKELY(¬V_is_Active(v))) {
    MARPA_ERROR(MARPA_ERR_VALUATOR_INACTIVE);
    return failure_indicator;
  }
  V_is_Trace(v) ⟸ Boolean(flag);
  return 1;
}
```

1097. Nook of valuator.

#define NOOK_of_V(val) ((val)→t_nook)

⟨ Int aligned value elements 1085 ⟩ +≡
NOOKID t_nook;

1098. ⟨ Initialize value elements 1074 ⟩ +≡

```
NOOK_of_V(v) ⟸ −1;
```

1099. Returns -1 if valuator is nulling.

⟨ Function definitions 41 ⟩ +≡

```
Marpa_Nook_ID _marpa_v_nook(Marpa_Value public_v)
{
  ⟨ Return −2 on failure 1229 ⟩
  const VALUE v ⟸ (VALUE) public_v;
  ⟨ Unpack value objects 1090 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  if (_MARPA_UNLIKELY(V_is_Nulling(v))) return −1;
  if (_MARPA_UNLIKELY(¬V_is_Active(v))) {
    MARPA_ERROR(MARPA_ERR_VALUATOR_INACTIVE);
    return failure_indicator;
  }
  return NOOK_of_V(v);
}
```

1100. Symbol valued status.

1101.

```
#define XSY_is_Valued_BV_of_V(v) ((v)→t_xsy_is_valued)
```

1102.

```
#define XRL_is_Valued_BV_of_V(v) ((v)→t_xrl_is_valued)
#define Valued_Locked_BV_of_V(v) ((v)→t_valued_locked)
```

⟨ Widely aligned value elements 1075 ⟩ +≡

```
LBV t_xsy_is_valued;
LBV t_xrl_is_valued;
LBV t_valued_locked;
```

1103. ⟨ Initialize value elements 1074 ⟩ +≡

```
{
  XSY_is_Valued_BV_of_V(v) ⟸ lbv_clone(v→t_obs, Valued_BV_of_B(b), xsy_count);
  Valued_Locked_BV_of_V(v) ⟸ lbv_clone(v→t_obs, Valued_Locked_BV_of_B(b),
      xsy_count);
}
```

1104.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int symbol_is_valued(VALUE v, Marpa_Symbol_ID xsy_id)
{
  return lbv_bit_test(XSY_is_Valued_BV_of_V(v), xsy_id);
}
```

1105.

⟨ Function definitions 41 ⟩ +≡

```
int marpa_v_symbol_is_valued(Marpa_Value public_v, Marpa_Symbol_ID xsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  const VALUE v ⟸ (VALUE) public_v;
  ⟨ Unpack value objects 1090 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if xsy_id is malformed 1232 ⟩
  ⟨ Soft fail if xsy_id does not exist 1233 ⟩
  return lbv_bit_test(XSY_is_Valued_BV_of_V(v), xsy_id);
}
```

1106. The setting here overrides the value set with the grammar.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int symbol_is_valued_set(VALUE v, XSYID xsy_id, int value)
{
  ⟨ Return −2 on failure 1229 ⟩
  const int old_value ⟸ lbv_bit_test(XSY_is_Valued_BV_of_V(v), xsy_id);
  if (old_value ≡ value) {
    lbv_bit_set(Valued_Locked_BV_of_V(v), xsy_id);
    return value;
  }
  if (_MARPA_UNLIKELY(lbv_bit_test(Valued_Locked_BV_of_V(v), xsy_id))) {
    return failure_indicator;
  }
  lbv_bit_set(Valued_Locked_BV_of_V(v), xsy_id);
  if (value) {
    lbv_bit_set(XSY_is_Valued_BV_of_V(v), xsy_id);
  }
  else {
    lbv_bit_clear(XSY_is_Valued_BV_of_V(v), xsy_id);
  }
  return value;
}
```

1107. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_v_symbol_is_valued_set(Marpa_Value public_v, Marpa_Symbol_ID
        xsy_id, int value)
{
  const VALUE v ⟸ (VALUE) public_v;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack value objects 1090 ⟩
  ⟨ Fail if fatal error 1249 ⟩
```

```
    if (_MARPA_UNLIKELY(value < 0 ∨ value > 1)) {
      MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
      return failure_indicator;
    }
    ⟨Fail if xsy_id is malformed 1232⟩
    ⟨Soft fail if xsy_id does not exist 1233⟩
    return symbol_is_valued_set(v, xsy_id, value);
  }
```

1108. Force all symbols to be locked as valued. Return failure if that is not possible.

⟨Function definitions 41⟩ +≡

```
  int marpa_v_valued_force(Marpa_Value public_v)
  {
    const VALUE v ⟸ (VALUE) public_v;
    ⟨Return −2 on failure 1229⟩
    XSYID xsy_count;
    XSYID xsy_id;
    ⟨Unpack value objects 1090⟩
    ⟨Fail if fatal error 1249⟩
    xsy_count ⟸ XSY_Count_of_G(g);
    for (xsy_id ⟸ 0; xsy_id < xsy_count; xsy_id++) {
      if (_MARPA_UNLIKELY(¬lbv_bit_test(XSY_is_Valued_BV_of_V(v),
            xsy_id) ∧ lbv_bit_test(Valued_Locked_BV_of_V(v), xsy_id))) {
        return failure_indicator;
      }
      lbv_bit_set(Valued_Locked_BV_of_V(v), xsy_id);
      lbv_bit_set(XSY_is_Valued_BV_of_V(v), xsy_id);
    }
    return xsy_count;
  }
```

1109. ⟨Function definitions 41⟩ +≡

```
  int marpa_v_rule_is_valued_set(Marpa_Value public_v, Marpa_Rule_ID xrl_id, int
        value)
  {
    const VALUE v ⟸ (VALUE) public_v;
    ⟨Return −2 on failure 1229⟩
    ⟨Unpack value objects 1090⟩
    ⟨Fail if fatal error 1249⟩
    if (_MARPA_UNLIKELY(value < 0 ∨ value > 1)) {
      MARPA_ERROR(MARPA_ERR_INVALID_BOOLEAN);
      return failure_indicator;
    }
    ⟨Fail if xrl_id is malformed 1241⟩
```

```
    ⟨ Soft fail if xrl_id does not exist 1239 ⟩
    {
      const XRL xrl ⟸ XRL_by_ID(xrl_id);
      const XSYID xsy_id ⟸ LHS_ID_of_XRL(xrl);
      return symbol_is_valued_set(v, xsy_id, value);
    }
  }
```

1110. ⟨ Function definitions 41 ⟩ +≡

```
  int marpa_v_rule_is_valued(Marpa_Value public_v, Marpa_Rule_ID xrl_id)
  {
    const VALUE v ⟸ (VALUE) public_v;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack value objects 1090 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Fail if xrl_id is malformed 1241 ⟩
    ⟨ Soft fail if xrl_id does not exist 1239 ⟩
    {
      const XRL xrl ⟸ XRL_by_ID(xrl_id);
      const XSYID xsy_id ⟸ LHS_ID_of_XRL(xrl);
      return symbol_is_valued(v, xsy_id);
    }
  }
```

1111. Stepping the valuator. The value type indicates whether the value is for a semantic rule, a semantic token, etc.

⟨ Public typedefs 91 ⟩ +≡

```
  typedef int Marpa_Step_Type;
```

1112. *#define* STEP_GET_DATA MARPA_STEP_INTERNAL2

⟨ Function definitions 41 ⟩ +≡

```
  Marpa_Step_Type marpa_v_step(Marpa_Value public_v)
  {
    ⟨ Return −2 on failure 1229 ⟩
    const VALUE v ⟸ (VALUE) public_v;
    if (V_is_Nulling(v)) {
      ⟨ Unpack value objects 1090 ⟩
      ⟨ Step through a nulling valuator 1114 ⟩
      return Step_Type_of_V(v);
    }
    while (V_is_Active(v)) {
      Marpa_Step_Type current_value_type ⟸ Next_Value_Type_of_V(v);
      switch (current_value_type) {
      case MARPA_STEP_INITIAL:
```

```
        {
          XSYID xsy_count;
          ⟨ Unpack value objects 1090 ⟩
          xsy_count ⟸ XSY_Count_of_G(g);
          lbv_fill(Valued_Locked_BV_of_V(v), xsy_count);
          ⟨ Set rule-is-valued vector 1113 ⟩
        }    /* fall through */
      case STEP_GET_DATA: ⟨ Perform evaluation steps 1115 ⟩
        if (¬V_is_Active(v)) break;     /* fall through */
      case MARPA_STEP_TOKEN:
        {
          int tkn_type ⟸ Token_Type_of_V(v);
          Next_Value_Type_of_V(v) ⟸ MARPA_STEP_RULE;
          if (tkn_type ≡ NULLING_TOKEN_OR_NODE) {
            if (lbv_bit_test(XSY_is_Valued_BV_of_V(v), XSYID_of_V(v))) {
              Result_of_V(v) ⟸ Arg_N_of_V(v);
              return Step_Type_of_V(v) ⟸ MARPA_STEP_NULLING_SYMBOL;
            }
          }
          else if (tkn_type ≠ DUMMY_OR_NODE) {
            Result_of_V(v) ⟸ Arg_N_of_V(v);
            return Step_Type_of_V(v) ⟸ MARPA_STEP_TOKEN;
          }
        }    /* fall through */
      case MARPA_STEP_RULE:
        if (RULEID_of_V(v) ≥ 0) {
          Next_Value_Type_of_V(v) ⟸ MARPA_STEP_TRACE;
          Result_of_V(v) ⟸ Arg_0_of_V(v);
          return Step_Type_of_V(v) ⟸ MARPA_STEP_RULE;
        }    /* fall through */
      case MARPA_STEP_TRACE: Next_Value_Type_of_V(v) ⟸ STEP_GET_DATA;
        if (V_is_Trace(v)) {
          return Step_Type_of_V(v) ⟸ MARPA_STEP_TRACE;
        }
      }
    }
    Next_Value_Type_of_V(v) ⟸ MARPA_STEP_INACTIVE;
    return Step_Type_of_V(v) ⟸ MARPA_STEP_INACTIVE;
  }
```

1113. A rule is valued if and only if its LHS is a valued symbol. All the symbol values have been locked at this point, so we can memoize the value for the rule.

```
⟨ Set rule-is-valued vector 1113 ⟩ ≡
  {
```

```
    const LBV xsy_bv ⟸ XSY_is_Valued_BV_of_V(v);
    const XRLID xrl_count ⟸ XRL_Count_of_G(g);
    const LBV xrl_bv ⟸ lbv_obs_new0(v→t_obs, xrl_count);
    XRLID xrlid;
    XRL_is_Valued_BV_of_V(v) ⟸ xrl_bv;
    for (xrlid ⟸ 0; xrlid < xrl_count; xrlid++) {
      const XRL xrl ⟸ XRL_by_ID(xrlid);
      const XSYID lhs_xsy_id ⟸ LHS_ID_of_XRL(xrl);
      if (lbv_bit_test(xsy_bv, lhs_xsy_id)) {
        lbv_bit_set(xrl_bv, xrlid);
      }
    }
  }
```

This code is used in section 1112.

1114. We do no tracing of nulling valuators, at least at this point.

```
⟨ Step through a nulling valuator 1114 ⟩ ≡
  {
    XSYID_of_V(v) ⟸ g→t_start_xsy_id;
    Token_Start_of_V(v) ⟸ YS_ID_of_V(v) ⟸ 0;
    Result_of_V(v) ⟸ Arg_0_of_V(v) ⟸ Arg_N_of_V(v) ⟸ 0;
    Step_Type_of_V(v) ⟸ MARPA_STEP_INACTIVE;
    if (Next_Value_Type_of_V(v) ≡ MARPA_STEP_INITIAL ∧
          lbv_bit_test(XSY_is_Valued_BV_of_V(v), XSYID_of_V(v))) {
      Step_Type_of_V(v) ⟸ MARPA_STEP_NULLING_SYMBOL;
    }
    Next_Value_Type_of_V(v) ⟸ MARPA_STEP_INACTIVE;
  }
```

This code is used in section 1112.

1115. ⟨ Perform evaluation steps 1115 ⟩ ≡

```
  {
    AND and_nodes;
      /* flag to indicate whether the arguments of a rule should be popped off the stack.
         Coming into this loop that is always the case – if no rule was executed, this is a
         no-op. */
    int pop_arguments ⟸ 1;
    ⟨ Unpack value objects 1090 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    and_nodes ⟸ ANDs_of_B(B_of_O(o));
    if (NOOK_of_V(v) < 0) {
      NOOK_of_V(v) ⟸ Size_of_TREE(t);
    }
    while (1) {
```

```
OR or;
IRL nook_irl;
Token_Value_of_V(v) ⇐ −1;
RULEID_of_V(v) ⇐ −1;
NOOK_of_V(v)−−;
if (NOOK_of_V(v) < 0) {
  Next_Value_Type_of_V(v) ⇐ MARPA_STEP_INACTIVE;
  break;
}
if (pop_arguments) {
    /* Pop the arguments for the last rule execution off of the stack */
  Arg_N_of_V(v) ⇐ Arg_0_of_V(v);
  pop_arguments ⇐ 0;
}
{
  ANDID and_node_id;
  AND and_node;
  int cause_or_node_type;
  OR cause_or_node;
  const NOOK nook ⇐ NOOK_of_TREE_by_IX(t, NOOK_of_V(v));
  const int choice ⇐ Choice_of_NOOK(nook);
  or ⇐ OR_of_NOOK(nook);
  YS_ID_of_V(v) ⇐ YS_Ord_of_OR(or);
  and_node_id ⇐ and_order_get(o, or, choice);
  and_node ⇐ and_nodes + and_node_id;
  cause_or_node ⇐ Cause_OR_of_AND(and_node);
  cause_or_node_type ⇐ Type_of_OR(cause_or_node);
  switch (cause_or_node_type) {
  case VALUED_TOKEN_OR_NODE: Token_Type_of_V(v) ⇐ cause_or_node_type;
    Arg_0_of_V(v) ⇐ ++Arg_N_of_V(v);
    {
      const OR predecessor ⇐ Predecessor_OR_of_AND(and_node);
      XSYID_of_V(v) ⇐
          ID_of_XSY(Source_XSY_of_NSYID(NSYID_of_OR(cause_or_node)));
      Token_Start_of_V(v) ⇐ predecessor ? YS_Ord_of_OR(predecessor) :
          Origin_Ord_of_OR(or);
      Token_Value_of_V(v) ⇐ Value_of_OR(cause_or_node);
    }
    break;
  case NULLING_TOKEN_OR_NODE: Token_Type_of_V(v) ⇐ cause_or_node_type;
    Arg_0_of_V(v) ⇐ ++Arg_N_of_V(v);
    {
      const XSY source_xsy ⇐
          Source_XSY_of_NSYID(NSYID_of_OR(cause_or_node));
```

```
        const XSYID source_xsy_id ⟸ ID_of_XSY(source_xsy);
        if (bv_bit_test(XSY_is_Valued_BV_of_V(v), source_xsy_id)) {
          XSYID_of_V(v) ⟸ source_xsy_id;
          Token_Start_of_V(v) ⟸ YS_ID_of_V(v);
        }
        else {
          Token_Type_of_V(v) ⟸ DUMMY_OR_NODE;
            /* DUMMY_OR_NODE indicates arbitrary semantics for this token */
        }
      }
      break;
    default: Token_Type_of_V(v) ⟸ DUMMY_OR_NODE;
    }
  }
  nook_irl ⟸ IRL_of_OR(or);
  if (Position_of_OR(or) ≡ Length_of_IRL(nook_irl)) {
    int virtual_rhs ⟸ IRL_has_Virtual_RHS(nook_irl);
    int virtual_lhs ⟸ IRL_has_Virtual_LHS(nook_irl);
    int real_symbol_count;
    const MARPA_DSTACK virtual_stack ⟸ &VStack_of_V(v);
    if (virtual_lhs) {
      real_symbol_count ⟸ Real_SYM_Count_of_IRL(nook_irl);
      if (virtual_rhs) {
        *(MARPA_DSTACK_TOP(*virtual_stack, int)) += real_symbol_count;
      }
      else {
        *MARPA_DSTACK_PUSH(*virtual_stack, int) ⟸ real_symbol_count;
      }
    }
    else {
      if (virtual_rhs) {
        real_symbol_count ⟸ Real_SYM_Count_of_IRL(nook_irl);
        real_symbol_count += *MARPA_DSTACK_POP(*virtual_stack, int);
      }
      else {
        real_symbol_count ⟸ Length_of_IRL(nook_irl);
      }
      {    /* Currently all rules with a non-virtual LHS are */
          /* "semantic" rules. */
        XRLID original_rule_id ⟸ ID_of_XRL(Source_XRL_of_IRL(nook_irl));
        Arg_0_of_V(v) ⟸ Arg_N_of_V(v) − real_symbol_count + 1;
        pop_arguments ⟸ 1;
        if (lbv_bit_test(XRL_is_Valued_BV_of_V(v), original_rule_id)) {
          RULEID_of_V(v) ⟸ original_rule_id;
```

```
              Rule_Start_of_V(v) ⇐= Origin_Ord_of_OR(or);
            }
          }
        }
      }
      if (RULEID_of_V(v) ≥ 0) break;
      if (Token_Type_of_V(v) ≠ DUMMY_OR_NODE) break;
      if (V_is_Trace(v)) break;
    }
  }
```

This code is used in section 1112.

1116. Lightweight boolean vectors (LBV). These macros and functions assume that the caller remembers the boolean vector's length. They also take no precautions about trailing bits in the last word. Most operations do not need to. When and if there are such operations, it will be up to the caller to make sure that the trailing bits are correct.

```
#define lbv_wordbits (sizeof (LBW) * 8U)
#define lbv_lsb (1U)
#define lbv_msb (1U << (lbv_wordbits - 1U))
```

⟨ Private typedefs 49 ⟩ +≡

```
typedef unsigned int LBW;
typedef LBW *LBV;
```

1117. Given a number of bits, compute the size.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int lbv_bits_to_size(int bits)
{
  const LBW result = (LBW)(((unsigned int)
      bits + (lbv_wordbits - 1))/lbv_wordbits);
  return (int) result;
}
```

1118. Create an unitialized LBV on an obstack.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE Bit_Vector lbv_obs_new(struct marpa_obstack *obs, int bits)
{
  int size = lbv_bits_to_size(bits);
  LBV lbv = marpa_obs_new(obs, LBW, size);
  return lbv;
}
```

1119. Zero an LBV.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE Bit_Vector lbv_zero(Bit_Vector lbv, int bits)
{
  int size = lbv_bits_to_size(bits);
  if (size > 0) {
    LBW *addr = lbv;
    while (size--) *addr++ = 0U;
  }
  return lbv;
}
```

1120. Create a zeroed LBV on an obstack.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE Bit_Vector lbv_obs_new0(struct marpa_obstack *obs, int bits)
{
  LBV lbv ⟸ lbv_obs_new(obs, bits);
  return lbv_zero(lbv, bits);
}
```

1121. Basic LBV operations.

```
#define lbv_w(lbv, bit) ((lbv) + ((bit)/lbv_wordbits))
#define lbv_b(bit) (lbv_lsb ≪ ((bit) % bv_wordbits))
#define lbv_bit_set(lbv, bit) (*lbv_w((lbv), (LBW)(bit)) |= lbv_b((LBW)(bit)))
#define lbv_bit_clear(lbv, bit)
        (*lbv_w((lbv), ((LBW)(bit))) &= ∼lbv_b((LBW)(bit)))
#define lbv_bit_test(lbv, bit)
        ((*lbv_w((lbv), ((LBW)(bit))) & lbv_b((LBW)(bit))) ≠ 0U)
```

1122. Clone an LBV onto an obstack.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE LBV lbv_clone(struct marpa_obstack *obs, LBV old_lbv, int bits)
{
  int size ⟸ lbv_bits_to_size(bits);
  const LBV new_lbv ⟸ marpa_obs_new(obs, LBW, size);
  if (size > 0) {
    LBW *from_addr ⟸ old_lbv;
    LBW *to_addr ⟸ new_lbv;
    while (size−−) *to_addr++ ⟸ *from_addr++;
  }
  return new_lbv;
}
```

1123. Fill an LBV with ones. No special provision is made for trailing bits.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE LBV lbv_fill(LBV lbv, int bits)
{
  int size ⟸ lbv_bits_to_size(bits);
  if (size > 0) {
    LBW *to_addr ⟸ lbv;
    while (size−−) *to_addr++ ⟸ ∼((LBW) 0);
  }
  return lbv;
}
```

1124. Boolean vectors. Marpa's boolean vectors are adapted from Steffen Beyer's Bit-Vector package on CPAN. This is a combined Perl package and C library for handling boolean vectors. Someone seeking a general boolean vector package should look at Steffen's instead. `libmarpa`'s boolean vectors are tightly tied in with its own needs and environment.

⟨ Private typedefs 49 ⟩ +≡
```
typedef LBW Bit_Vector_Word;
typedef Bit_Vector_Word *Bit_Vector;
```

1125. Some defines and constants

```
#define BV_BITS(bv) *(bv - 3)
#define BV_SIZE(bv) *(bv - 2)
#define BV_MASK(bv) *(bv - 1)
```

⟨ Global constant variables 40 ⟩ +≡
```
static const unsigned int bv_wordbits ⟸ lbv_wordbits;
static const unsigned int bv_modmask ⟸ lbv_wordbits − 1U;
static const unsigned int bv_hiddenwords ⟸ 3;
static const unsigned int bv_lsb ⟸ lbv_lsb;
static const unsigned int bv_msb ⟸ lbv_msb;
```

1126. Given a number of bits, compute the size.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE unsigned int bv_bits_to_size(int bits)
{
  return ((LBW) bits + bv_modmask)/bv_wordbits;
}
```

1127. Given a number of bits, compute the unused-bit mask.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE unsigned int bv_bits_to_unused_mask(int bits)
{
  LBW mask ⟸ (LBW) bits & bv_modmask;
  if (mask) mask ⟸ (LBW) ~(~0UL ≪ mask);
  else mask ⟸ (LBW) ~0UL;
  return (mask);
}
```

1128. Create a boolean vector.

1129. Always start with an all-zero vector. Note this code is a bit tricky — the pointer returned is to the data. This is offset from the `malloc`'d space, by `bv_hiddenwords`.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE Bit_Vector bv_create(int bits)
{
```

```
    LBW size ⟸ bv_bits_to_size(bits);
    LBW bytes ⟸ (size + (LBW) bv_hiddenwords) * (LBW) sizeof(Bit_Vector_Word);
    LBW *addr ⟸ (Bit_Vector) my_malloc0((size_t) bytes);
    *addr++ ⟸ (LBW) bits;
    *addr++ ⟸ size;
    *addr++ ⟸ bv_bits_to_unused_mask(bits);
    return addr;
  }
```

1130. Create a boolean vector on an obstack.

1131. Always start with an all-zero vector. Note this code is a bit tricky — the pointer returned is to the data. This is offset from the `malloc`'d space, by `bv_hiddenwords`.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE Bit_Vector bv_obs_create(struct marpa_obstack *obs, int bits)
  {
    LBW size ⟸ bv_bits_to_size(bits);
    LBW bytes ⟸ (size + (LBW) bv_hiddenwords) * (LBW) sizeof(Bit_Vector_Word);
    LBW *addr ⟸ (Bit_Vector) marpa__obs_alloc(obs, (size_t) bytes,
        ALIGNOF(LBW));
    *addr++ ⟸ (LBW) bits;
    *addr++ ⟸ size;
    *addr++ ⟸ bv_bits_to_unused_mask(bits);
    if (size > 0) {
      Bit_Vector bv ⟸ addr;
      while (size--) *bv++ ⟸ 0U;
    }
    return addr;
  }
```

1132. Shadow a boolean vector. Create another vector the same size as the original, but with all bits unset.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE Bit_Vector bv_shadow(Bit_Vector bv)
  {
    return bv_create((int) BV_BITS(bv));
  }
  PRIVATE Bit_Vector bv_obs_shadow(struct marpa_obstack *obs, Bit_Vector bv)
  {
    return bv_obs_create(obs, (int) BV_BITS(bv));
  }
```

1133. Clone a boolean vector. Given a boolean vector, creates a new vector which is an exact duplicate. This call allocates a new vector, which must be `free`'d.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE Bit_Vector bv_copy(Bit_Vector bv_to, Bit_Vector bv_from)
{
  LBW *p_to ⇐ bv_to;
  const LBW bits ⇐ BV_BITS(bv_to);
  if (bits > 0) {
    LBW count ⇐ BV_SIZE(bv_to);
    while (count--) *p_to++ ⇐ *bv_from++;
  }
  return (bv_to);
}
```

1134. Clone a boolean vector. Given a boolean vector, creates a new vector which is an exact duplicate. This call allocates a new vector, which must be `free`'d.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE Bit_Vector bv_clone(Bit_Vector bv)
{
  return bv_copy(bv_shadow(bv), bv);
}
PRIVATE Bit_Vector bv_obs_clone(struct marpa_obstack *obs, Bit_Vector bv)
{
  return bv_copy(bv_obs_shadow(obs, bv), bv);
}
```

1135. Free a boolean vector.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE void bv_free(Bit_Vector vector)
{
  if (_MARPA_LIKELY(vector ≠ Λ)) {
    vector −= bv_hiddenwords;
    my_free(vector);
  }
}
```

1136. Fill a boolean vector.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE void bv_fill(Bit_Vector bv)
{
  LBW size ⇐ BV_SIZE(bv);
  if (size ≤ 0) return;
  while (size--) *bv++ ⇐ ~0U;
  --bv;
```

```
    *bv &= BV_MASK(bv);
  }
```

1137. Clear a boolean vector.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void bv_clear(Bit_Vector bv)
  {
    LBW size ⇐ BV_SIZE(bv);
    if (size ≤ 0) return;
    while (size--) *bv++ ⇐ 0U;
  }
```

1138. This function "overclears" — it clears "too many bits". It clears a prefix of the boolean vector faster than an interval clear, at the expense of often clearing more bits than were requested. In some situations clearing the extra bits is OK.

1139. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void bv_over_clear(Bit_Vector bv, int raw_bit)
  {
    const LBW bit ⇐ (LBW) raw_bit;
    LBW length ⇐ bit/bv_wordbits + 1;
    while (length--) *bv++ ⇐ 0U;
  }
```

1140. Set a boolean vector bit.

1141. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void bv_bit_set(Bit_Vector vector, int raw_bit)
  {
    const LBW bit ⇐ (LBW) raw_bit;
    *(vector + (bit/bv_wordbits)) |= (bv_lsb ≪ (bit % bv_wordbits));
  }
```

1142. Clear a boolean vector bit.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void bv_bit_clear(Bit_Vector vector, int raw_bit)
  {
    const LBW bit ⇐ (LBW) raw_bit;
    *(vector + (bit/bv_wordbits)) &= ~(bv_lsb ≪ (bit % bv_wordbits));
  }
```

1143. Test a boolean vector bit.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int bv_bit_test(Bit_Vector vector, int raw_bit)
{
    const LBW bit ⟸ (LBW) raw_bit;
    return (*(vector + (bit/bv_wordbits)) & (bv_lsb ≪ (bit % bv_wordbits))) ≠ 0U;
}
```

1144. Test and set a boolean vector bit. Ensure that a bit is set. Return its previous value to the call, so that the return value is 1 if the call had no effect, zero otherwise.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int bv_bit_test_then_set(Bit_Vector vector, int raw_bit)
{
    const LBW bit ⟸ (LBW) raw_bit;
    Bit_Vector addr ⟸ vector + (bit/bv_wordbits);
    LBW mask ⟸ bv_lsb ≪ (bit % bv_wordbits);
    if ((*addr & mask) ≠ 0U) return 1;
    *addr |= mask;
    return 0;
}
```

1145. Test a boolean vector for all zeroes.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int bv_is_empty(Bit_Vector addr)
{
    LBW size ⟸ BV_SIZE(addr);
    int r ⟸ 1;
    if (size > 0) {
        *(addr + size − 1) &= BV_MASK(addr);
        while (r ∧ (size−− > 0)) r ⟸ (*addr++ ≡ 0);
    }
    return (r);
}
```

1146. Bitwise-negate a boolean vector.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void bv_not(Bit_Vector X, Bit_Vector Y)
{
    LBW size ⟸ BV_SIZE(X);
    LBW mask ⟸ BV_MASK(X);
    while (size−− > 0) *X++ ⟸ ∼*Y++;
    *(−−X) &= mask;
}
```

1147. Bitwise-and a boolean vector.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void bv_and(Bit_Vector X, Bit_Vector Y, Bit_Vector Z)
{
  LBW size ⇐ BV_SIZE(X);
  LBW mask ⇐ BV_MASK(X);
  while (size-- > 0) *X++ ⇐ *Y++ & *Z++;
  *(--X) &= mask;
}
```

1148. Bitwise-or a boolean vector.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void bv_or(Bit_Vector X, Bit_Vector Y, Bit_Vector Z)
{
  LBW size ⇐ BV_SIZE(X);
  LBW mask ⇐ BV_MASK(X);
  while (size-- > 0) *X++ ⇐ *Y++ | *Z++;
  *(--X) &= mask;
}
```

1149. Bitwise-or-assign a boolean vector.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void bv_or_assign(Bit_Vector X, Bit_Vector Y)
{
  LBW size ⇐ BV_SIZE(X);
  LBW mask ⇐ BV_MASK(X);
  while (size-- > 0) *X++ |= *Y++;
  *(--X) &= mask;
}
```

1150. Scan a boolean vector.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE_NOT_INLINE int bv_scan(Bit_Vector bv, int raw_start, int *raw_min, int
        *raw_max)
{
  LBW start ⇐ (LBW) raw_start;
  LBW min;
  LBW max;
  LBW size ⇐ BV_SIZE(bv);
  LBW mask ⇐ BV_MASK(bv);
  LBW offset;
  LBW bitmask;
  LBW value;
  int empty;
```

```
if (size ≡ 0) return 0;
if (start ≥ BV_BITS(bv)) return 0;
min ⟸ start;
max ⟸ start;
offset ⟸ start/bv_wordbits;
∗(bv + size − 1) &= mask;
bv += offset;
size −= offset;
bitmask ⟸ (LBW) 1 ≪ (start & bv_modmask);
mask ⟸ ∼(bitmask | (bitmask − (LBW) 1));
value ⟸ ∗bv++;
if ((value & bitmask) ≡ 0) {
  value &= mask;
  if (value ≡ 0) {
    offset++;
    empty ⟸ 1;
    while (empty ∧ (−−size > 0)) {
      if ((value ⟸ ∗bv++)) empty ⟸ 0;
      else offset++;
    }
    if (empty) {
      ∗raw_min ⟸ (int) min;
      ∗raw_max ⟸ (int) max;
      return 0;
    }
  }
  start ⟸ offset ∗ bv_wordbits;
  bitmask ⟸ bv_lsb;
  mask ⟸ value;
  while (¬(mask & bv_lsb)) {
    bitmask ≪= 1;
    mask ≫= 1;
    start++;
  }
  mask ⟸ ∼(bitmask | (bitmask − 1));
  min ⟸ start;
  max ⟸ start;
}
value ⟸ ∼value;
value &= mask;
if (value ≡ 0) {
  offset++;
  empty ⟸ 1;
  while (empty ∧ (−−size > 0)) {
    if ((value ⟸ ∼∗bv++)) empty ⟸ 0;
```

```
      else offset++;
    }
    if (empty) value ⟸ bv_lsb;
  }
  start ⟸ offset * bv_wordbits;
  while (¬(value & bv_lsb)) {
    value ≫= 1;
    start++;
  }
  max ⟸ --start;
  *raw_min ⟸ (int) min;
  *raw_max ⟸ (int) max;
  return 1;
}
```

1151. Count the bits in a boolean vector.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int bv_count(Bit_Vector v)
{
  int start, min, max;
  int count ⟸ 0;
  for (start ⟸ 0; bv_scan(v, start, &min, &max); start ⟸ max + 2) {
    count += max − min + 1;
  }
  return count;
}
```

1152. The RHS closure of a vector. Despite the fact that they are actually tied closely to their use in `libmarpa`, most of the logic of boolean vectors has a "pure math" appearance. This routine has a direct connection with the grammar.

Several properties of symbols that need to be determined have the property that, if all the symbols on the RHS of any rule have that property, so does its LHS symbol.

1153. The RHS closure looks a lot like the transitive closure, but there are several major differences. The biggest difference is that the RHS closure deals with properties and takes a **vector** to another vector; the transitive closure is for a relation and takes a transition **matrix** to another transition matrix.

1154. There are two properties of the RHS closure to note. First, any symbol in a set is in the RHS closure of that set.

1155. Second, the RHS closure is vacuously true. For any RHS closure property, every symbol which is on the LHS of an empty rule has that property. This means the RHS closure operation can only be used for properties which can meaningfully be regarded as vacuously true. In `libmarpa`, two important symbol properties are RHS closure properties: the property of being productive, and the property of being nullable.

1156. Produce the RHS closure of a vector. This routine takes a symbol vector and a grammar, and turns the original vector into the RHS closure of that vector. The original vector is destroyed.

```
⟨ Function definitions 41 ⟩ +≡
  PRIVATE void rhs_closure(GRAMMAR g, Bit_Vector bv, XRLID
        **xrl_list_x_rh_sym)
  {
    int min, max, start ⇐ 0;
    Marpa_Symbol_ID *end_of_stack ⇐ Λ;
      /* Create a work stack. */
    FSTACK_DECLARE(stack, XSYID)
    FSTACK_INIT(stack, XSYID, XSY_Count_of_G(g));
      /* bv is initialized to a set of symbols known to have the closure property. For
         example, for nullables, it is initialized to symbols on the LHS of an empty rule.
         We initialize the work stack with the set of symbols we know to have the closure
         property. */
    while (bv_scan(bv, start, &min, &max)) {
      XSYID xsy_id;
      for (xsy_id ⇐ min; xsy_id ≤ max; xsy_id++) {
        *(FSTACK_PUSH(stack)) ⇐ xsy_id;
      }
      start ⇐ max + 2;
    }

  while ((end_of_stack ⇐ FSTACK_POP(stack))) {      /* For as long as there is a
          symbol on the work stack. xsy_id is the symbol we're working on. */
      const XSYID xsy_id ⇐ *end_of_stack;
      XRLID *p_xrl ⇐ xrl_list_x_rh_sym[xsy_id];
      const XRLID *p_one_past_rules ⇐ xrl_list_x_rh_sym[xsy_id + 1];
      for ( ; p_xrl < p_one_past_rules; p_xrl++) {      /* For every rule with xsy_id
            on its RHS. rule is the rule we are currently working on. */
        const XRLID rule_id ⇐ *p_xrl;
        const XRL rule ⇐ XRL_by_ID(rule_id);
        int rule_length;
        int rh_ix;
        const XSYID lhs_id ⇐ LHS_ID_of_XRL(rule);
        const int is_sequence ⇐ XRL_is_Sequence(rule);
          /* If the LHS is already marked as having the closure property, skip ahead to
            the next rule. */
        if (bv_bit_test(bv, lhs_id)) goto NEXT_RULE;
        rule_length ⇐ Length_of_XRL(rule);
```

```
/* If any symbol on the RHS of rule does not have the closure property, we will
      be be justified in saying that it's LHS has the closure property – skip to
      the next rule. This works for the present allowed sequence rules – These
      currently always allow rules of length 1, which do not necessarily have a
      separator, so that they may be treated like BNF rules of length 1. */
  for (rh_ix ⟸ 0; rh_ix < rule_length; rh_ix++) {
    if (¬bv_bit_test(bv, RHS_ID_of_XRL(rule, rh_ix))) goto NEXT_RULE;
  }
/* If this is a sequence rule with a minimum greater than two, we must also
        check if the separator has the closure property. As of this writing, rules
        of minimum size greater than 1 are not allowed, so that this code is
        untested. */
  if (is_sequence ∧ Minimum_of_XRL(rule) ≥ 2) {
    XSYID separator_id ⟸ Separator_of_XRL(rule);
    if (separator_id ≥ 0) {
      if (¬bv_bit_test(bv, separator_id)) goto NEXT_RULE;
    }
  }
/* If I am here, we know that the the LHS symbol has the closure property, but is
        not marked as such. Mark it, and push it on the work stack. */
  bv_bit_set(bv, lhs_id);
  *(FSTACK_PUSH(stack)) ⟸ lhs_id;
NEXT_RULE: ;
  }
}
FSTACK_DESTROY(stack);
}
```

1157. Boolean matrixes. Marpa's boolean matrixes are implemented differently from the matrixes in Steffen Beyer's Bit-Vector package on CPAN, but like Beyer's matrixes are build on that package. Beyer's matrixes are a single boolean vector which special routines index by row and column. Marpa's matrixes are arrays of vectors.

Since there are "hidden words" before the data in each vectors, Marpa must repeat these for each row of a vector. Consequences:

- Marpa matrixes use a few extra bytes per row of space.
- Marpa's matrix pointers cannot be used as vectors.
- Marpa's rows **can** be used as vectors.
- Marpa's matrix pointers point to the beginning of the allocated space. *Bit_Vector* pointers use trickery and include "hidden words" before the pointer.

1158. Note that *typedef*'s for *Bit_Matrix* and *Bit_Vector* are identical.

1159. ⟨ Private structures 48 ⟩ +≡

```
struct s_bit_matrix {
  int t_row_count;
  Bit_Vector_Word t_row_data[1];
};
typedef struct s_bit_matrix *Bit_Matrix;
typedef struct s_bit_matrix Bit_Matrix_Object;
```

1160. Create a boolean matrix.

1161. Here the pointer returned is the actual start of the `malloc`'d space. This is **not** the case with vectors, whose pointer is offset for the "hidden words".

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE Bit_Matrix matrix_buffer_create(void *buffer, int rows, int columns)
{
  int row;
  const LBW bv_data_words ⟸ bv_bits_to_size(columns);
  const LBW bv_mask ⟸ bv_bits_to_unused_mask(columns);
  Bit_Matrix matrix_addr ⟸ buffer;

  matrix_addr→t_row_count ⟸ rows;
  for (row ⟸ 0; row < rows; row++) {
    const LBW row_start ⟸ (LBW) row * (bv_data_words + bv_hiddenwords);
    LBW *p_current_word ⟸ matrix_addr→t_row_data + row_start;
    LBW data_word_counter ⟸ bv_data_words;

    *p_current_word++ ⟸ (LBW) columns;
    *p_current_word++ ⟸ bv_data_words;
    *p_current_word++ ⟸ bv_mask;
    while (data_word_counter--) *p_current_word++ ⟸ 0;
  }
  return matrix_addr;
}
```

1162. Size a boolean matrix in bytes.

1163. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE size_t matrix_sizeof(int rows, int columns)
{
  const LBW bv_data_words ⟸ bv_bits_to_size(columns);
  const LBW row_bytes ⟸ (LBW)(bv_data_words + bv_hiddenwords) * (LBW)
      sizeof(Bit_Vector_Word);
  return offsetof(struct s_bit_matrix, t_row_data) + ((size_t) rows) * row_bytes;
}
```

1164. Create a boolean matrix on an obstack.

1165. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE Bit_Matrix matrix_obs_create(struct marpa_obstack *obs, int rows, int
        columns)
{    /* Needs to be aligned as a Bit_Matrix_Object */
  Bit_Matrix matrix_addr ⟸ marpa__obs_alloc(obs, matrix_sizeof(rows,
      columns), ALIGNOF(Bit_Matrix_Object));
  return matrix_buffer_create(matrix_addr, rows, columns);
}
```

1166. Clear a boolean matrix.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void matrix_clear(Bit_Matrix matrix)
{
  Bit_Vector row;
  int row_ix;
  const int row_count ⟸ matrix→t_row_count;
  Bit_Vector row0 ⟸ matrix→t_row_data + bv_hiddenwords;
  LBW words_per_row ⟸ BV_SIZE(row0) + bv_hiddenwords;
  row_ix ⟸ 0;
  row ⟸ row0;
  while (row_ix < row_count) {
    bv_clear(row);
    row_ix++;
    row += words_per_row;
  }
}
```

1167. Find the number of columns in a boolean matrix. The column count returned is for the first row. It is assumed that all rows have the same number of columns. Note that, in this implementation, the matrix has no idea internally of how many rows it has.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int matrix_columns(Bit_Matrix matrix)
{
  Bit_Vector row0 ⟸ matrix→t_row_data + bv_hiddenwords;
  return (int) BV_BITS(row0);
}
```

1168. Find a row of a boolean matrix. Here's where the slight extra overhead of repeating identical "hidden word" data for each row of a matrix pays off. This simply returns a pointer into the matrix. This is adequate if the data is not changed. If it is changed, the vector should be cloned. There is a bit of arithmetic, to deal with the hidden words offset.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE Bit_Vector matrix_row(Bit_Matrix matrix, int row)
{
  Bit_Vector row0 ⟸ matrix→t_row_data + bv_hiddenwords;
  LBW words_per_row ⟸ BV_SIZE(row0) + bv_hiddenwords;
  return row0 + (LBW) row * words_per_row;
}
```

1169. Set a boolean matrix bit.

1170. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void matrix_bit_set(Bit_Matrix matrix, int row, int column)
{
  Bit_Vector vector ⟸ matrix_row(matrix, row);
  bv_bit_set(vector, column);
}
```

1171. Clear a boolean matrix bit.

1172. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void matrix_bit_clear(Bit_Matrix matrix, int row, int column)
{
  Bit_Vector vector ⟸ matrix_row(matrix, row);
  bv_bit_clear(vector, column);
}
```

1173. Test a boolean matrix bit.

1174. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE int matrix_bit_test(Bit_Matrix matrix, int row, int column)
{
  Bit_Vector vector ⟸ matrix_row(matrix, row);
  return bv_bit_test(vector, column);
}
```

1175. Produce the transitive closure of a boolean matrix. This routine takes a matrix representing a relation and produces a matrix that represents the transitive closure of the relation. The matrix is assumed to be square. The input matrix will be destroyed.

Its uses Warshall's algorithm, which is $O(n^3)$ where the matrix is nxn.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE_NOT_INLINE void transitive_closure(Bit_Matrix matrix)
{
  int size ⟸ matrix_columns(matrix);
  int outer_row;
  for (outer_row ⟸ 0; outer_row < size; outer_row++) {
    Bit_Vector outer_row_v ⟸ matrix_row(matrix, outer_row);
    int column;
    for (column ⟸ 0; column < size; column++) {
      Bit_Vector inner_row_v ⟸ matrix_row(matrix, column);
      if (bv_bit_test(inner_row_v, outer_row)) {
        bv_or_assign(inner_row_v, outer_row_v);
      }
    }
  }
}
```

1176. Efficient stacks and queues.

1177. The interface for these macros is somewhat hackish, in that the user often must be aware of the implementation of the macros. Arguably, using these macros is not all that easier than hand-writing each instance. But the most important goal was safety – by writing this stuff once I have a greater assurance that it is tested and bug-free. Another important goal was that there be no compromise on efficiency, when compared to hand-written code.

1178. Fixed size stacks. `libmarpa` uses stacks and worklists extensively. Often a reasonable maximum size is known when they are set up, in which case they can be made very fast.

```
#define FSTACK_DECLARE(stack,type) struct {
    int t_count;
    type *t_base;
  } stack;
#define FSTACK_CLEAR(stack) ((stack).t_count ⇐ 0)
#define FSTACK_INIT(stack,type,n)
  (FSTACK_CLEAR(stack),((stack).t_base ⇐ marpa_new(type,n)))
#define FSTACK_SAFE(stack) ((stack).t_base ⇐ Λ)
#define FSTACK_BASE(stack,type) ( ( type * ) (stack).t_base )
#define FSTACK_INDEX(this,type,ix) (FSTACK_BASE((this),type) + (ix))
#define FSTACK_TOP(this,type)
  (FSTACK_LENGTH(this) ≤ 0 ? Λ : FSTACK_INDEX((this),type,
      FSTACK_LENGTH(this) − 1))
#define FSTACK_LENGTH(stack) ((stack).t_count)
#define FSTACK_PUSH(stack) ((stack).t_base + stack.t_count++)
#define FSTACK_POP(stack)
  ((stack).t_count ≤ 0 ? Λ : (stack).t_base + (−−(stack).t_count))
#define FSTACK_IS_INITIALIZED(stack) ((stack).t_base)
#define FSTACK_DESTROY(stack) (my_free((stack).t_base))
```

1179. Dynamic queues. This is simply a dynamic stack extended with a second index. These is no destructor at this point, because so far all uses of this let another container "steal" the data from this one. When one exists, it will simply call the dynamic stack destructor. Instead I define a destructor for the "thief" container to use when it needs to free the data.

```
#define DQUEUE_DECLARE(this) struct s_dqueue this
#define DQUEUE_INIT(this,type,initial_size) ((this.t_current ⇐ 0),
      MARPA_DSTACK_INIT(this.t_stack,type,initial_size))
#define DQUEUE_PUSH(this,type) MARPA_DSTACK_PUSH(this.t_stack,type)
#define DQUEUE_POP(this,type) MARPA_DSTACK_POP(this.t_stack,type)
#define DQUEUE_NEXT(this,type)
  (this.t_current ≥ MARPA_DSTACK_LENGTH(this.t_stack) ? Λ :
      (MARPA_DSTACK_BASE(this.t_stack,type)) + this.t_current++)
```

```
#define DQUEUE_BASE(this, type) MARPA_DSTACK_BASE(this.t_stack, type)
#define DQUEUE_END(this) MARPA_DSTACK_LENGTH(this.t_stack)
#define STOLEN_DQUEUE_DATA_FREE(data) MARPA_STOLEN_DSTACK_DATA_FREE(data)
⟨ Private incomplete structures 107 ⟩ +≡
  struct s_dqueue;
  typedef struct s_dqueue *DQUEUE;
```

1180. ⟨ Private structures 48 ⟩ +≡

```
  struct s_dqueue {
    int t_current;
    struct marpa_dstack_s t_stack;
  };
```

1181. Counted integer lists (CIL). As a structure, almost not worth bothering with, if it were not for its use in CILAR's. The first *int* is a count, and purists might insist on a struct instead of an array. A struct would reflect the logical structure more accurately. But would it make the actual code less readable, not more, which I believe has to be the object.

#define `Count_of_CIL(cil)` `(cil[0])`
#define `Item_of_CIL(cil, ix)` `(cil[1 + (ix)])`
#define `Sizeof_CIL(ix)` (*sizeof* (*int*) ∗ (1 + (`ix`)))

1182. ⟨ Private typedefs 49 ⟩ +≡
typedef int ∗*CIL*;

1183. Counted integer list arena (CILAR). These implement an especially efficient memory allocation scheme. Libmarpa needs many copies of integer lists, where the integers are symbol ID's, rule ID's, etc. The same ones are used again and again. The CILAR allows them to be allocated once and reused.

The CILAR is a software implementation of memory which is both random-access and content-addressable. Content-addressability saves space – when the contents are identical they can be reused. The content-addressability is implemented in software (as an AVL). While lookup is not slow the intention is that the content-addressability will used infrequently – once created or found the CIL will be memoized for random-access through a pointer.

1184. An obstack for the actual data, and a tree for the lookups.

```
⟨Private utility structures 1184⟩ ≡
  struct s_cil_arena {
    struct marpa_obstack *t_obs;
    MARPA_AVL_TREE t_avl;
    MARPA_DSTACK_DECLARE(t_buffer);
  };
  typedef struct s_cil_arena CILAR_Object;
```

This code is used in section 1381.

1185. ⟨Private incomplete structures 107⟩ +≡

```
  struct s_cil_arena;
```

1186. ⟨Private typedefs 49⟩ +≡

```
  typedef struct s_cil_arena *CILAR;
```

1187. To Do: The initial capacity of the CILAR dstack is absurdly small, in order to test the logic during development. Once things settle, MARPA_DSTACK_INIT should be changed to MARPA_DSTACK_INIT2.

```
#define CAPACITY_OF_CILAR(cilar) (CAPACITY_OF_DSTACK(cilar→t_buffer) − 1)
⟨Function definitions 41⟩ +≡
  PRIVATE void cilar_init(const CILAR cilar)
  {
    cilar→t_obs ⟸ marpa_obs_init;
    cilar→t_avl ⟸ _marpa_avl_create(cil_cmp, Λ);
    MARPA_DSTACK_INIT(cilar→t_buffer, int, 2);
    *MARPA_DSTACK_INDEX(cilar→t_buffer, int, 0) ⟸ 0;
  }
```

1188. To Do: The initial capacity of the CILAR dstack is absurdly small, in order to test the logic during development. Once things settle, MARPA_DSTACK_INIT should be changed to MARPA_DSTACK_INIT2.

```
⟨Function definitions 41⟩ +≡
  PRIVATE void cilar_buffer_reinit(const CILAR cilar)
```

```
{
  MARPA_DSTACK_DESTROY(cilar→t_buffer);
  MARPA_DSTACK_INIT(cilar→t_buffer, int, 2);
  *MARPA_DSTACK_INDEX(cilar→t_buffer, int, 0) ⟸ 0;
}
```

1189. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void cilar_destroy(const CILAR cilar)
{
  _marpa_avl_destroy(cilar→t_avl);
  marpa_obs_free(cilar→t_obs);
  MARPA_DSTACK_DESTROY((cilar→t_buffer));
}
```

1190. Return the empty CIL from a CILAR.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE CIL cil_empty(CILAR cilar)
{
  CIL cil ⟸ MARPA_DSTACK_BASE(cilar→t_buffer, int);
    /* We assume there is enough room */
  Count_of_CIL(cil) ⟸ 0;
  return cil_buffer_add(cilar);
}
```

1191. Return a singleton CIL from a CILAR.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE CIL cil_singleton(CILAR cilar, int element)
{
  CIL cil ⟸ MARPA_DSTACK_BASE(cilar→t_buffer, int);
  Count_of_CIL(cil) ⟸ 1;
  Item_of_CIL(cil, 0) ⟸ element;
    /* We assume there is enough room in the CIL buffer for a singleton */
  return cil_buffer_add(cilar);
}
```

1192. Add the CIL in the buffer to the CILAR. This method is optimized for the case where the CIL is alread in the CIL, in which case this method finds the current entry.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE CIL cil_buffer_add(CILAR cilar)
{
  CIL cil_in_buffer ⟸ MARPA_DSTACK_BASE(cilar→t_buffer, int);
  CIL found_cil ⟸ _marpa_avl_find(cilar→t_avl, cil_in_buffer);
  if (¬found_cil) {
    int i;
```

```
    const int cil_size_in_ints ⟸ Count_of_CIL(cil_in_buffer) + 1;
    found_cil ⟸ marpa_obs_new(cilar→t_obs, int, cil_size_in_ints);
    for (i ⟸ 0; i < cil_size_in_ints; i++) {
        /* Assumes that the CIL's are int * */
      found_cil[i] ⟸ cil_in_buffer[i];
    }
    _marpa_avl_insert(cilar→t_avl, found_cil);
  }
  return found_cil;
}
```

1193. Add a CIL taken from a bit vector to the CILAR. This method is optimized for the case where the CIL is already in the CIL, in which case this method finds the current entry. The CILAR buffer is used, so its current contents will be destroyed.

⟨Function definitions 41⟩ +≡

```
PRIVATE CIL cil_bv_add(CILAR cilar, Bit_Vector bv)
{
  int min, max, start ⟸ 0;
  cil_buffer_clear(cilar);
  for (start ⟸ 0; bv_scan(bv, start, &min, &max); start ⟸ max + 2) {
    int new_item;
    for (new_item ⟸ min; new_item ≤ max; new_item++) {
      cil_buffer_push(cilar, new_item);
    }
  }
  return cil_buffer_add(cilar);
}
```

1194. Clear the CILAR buffer.

⟨Function definitions 41⟩ +≡

```
PRIVATE void cil_buffer_clear(CILAR cilar)
{
  const MARPA_DSTACK dstack ⟸ &cilar→t_buffer;
  MARPA_DSTACK_CLEAR(*dstack);
    /* Has same effect as Count_of_CIL(cil_in_buffer) ⟸ 0, except that it sets the
       MARPA_DSTACK up properly */
  *MARPA_DSTACK_PUSH(*dstack, int) ⟸ 0;
}
```

1195. Push an *int* onto the end of the CILAR buffer. It is up to the caller to ensure the buffer is sorted when and if added to the CILAR.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE CIL cil_buffer_push(CILAR cilar, int new_item)
{
  CIL cil_in_buffer;
  MARPA_DSTACK dstack ⟸ &cilar→t_buffer;
  *MARPA_DSTACK_PUSH(*dstack, int) ⟸ new_item;
    /* Note that the buffer CIL might have been moved by the MARPA_DSTACK_PUSH */
  cil_in_buffer ⟸ MARPA_DSTACK_BASE(*dstack, int);
  Count_of_CIL(cil_in_buffer)++;
  return cil_in_buffer;
}
```

1196. Make sure that the CIL buffer is large enough to hold `element_count` elements.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE CIL cil_buffer_reserve(CILAR cilar, int element_count)
{
  const int desired_dstack_capacity ⟸ element_count + 1;
    /* One extra for the count word */
  const int old_dstack_capacity ⟸ MARPA_DSTACK_CAPACITY(cilar→t_buffer);
  if (old_dstack_capacity < desired_dstack_capacity) {
    const int target_capacity ⟸ MAX(old_dstack_capacity * 2,
        desired_dstack_capacity);
    MARPA_DSTACK_RESIZE(&(cilar→t_buffer), int, target_capacity);
  }
  return MARPA_DSTACK_BASE(cilar→t_buffer, int);
}
```

1197. Merge two CIL's into a new one. Not used at this point. This method trades unneeded obstack block allocations for CPU speed.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE CIL cil_merge(CILAR cilar, CIL cil1, CIL cil2)
{
  const int cil1_count ⟸ Count_of_CIL(cil1);
  const int cil2_count ⟸ Count_of_CIL(cil2);
  CIL new_cil ⟸ cil_buffer_reserve(cilar, cil1_count + cil2_count);
  int new_cil_ix ⟸ 0;
  int cil1_ix ⟸ 0;
  int cil2_ix ⟸ 0;
  while (cil1_ix < cil1_count ∧ cil2_ix < cil2_count) {
    const int item1 ⟸ Item_of_CIL(cil1, cil1_ix);
    const int item2 ⟸ Item_of_CIL(cil2, cil2_ix);
```

```
      if (item1 < item2) {
        Item_of_CIL(new_cil, new_cil_ix) ⇐ item1;
        cil1_ix++;
        new_cil_ix++;
        continue;
      }
      if (item2 < item1) {
        Item_of_CIL(new_cil, new_cil_ix) ⇐ item2;
        cil2_ix++;
        new_cil_ix++;
        continue;
      }
      Item_of_CIL(new_cil, new_cil_ix) ⇐ item1;
      cil1_ix++;
      cil2_ix++;
      new_cil_ix++;
    }
    while (cil1_ix < cil1_count) {
      const int item1 ⇐ Item_of_CIL(cil1, cil1_ix);
      Item_of_CIL(new_cil, new_cil_ix) ⇐ item1;
      cil1_ix++;
      new_cil_ix++;
    }
    while (cil2_ix < cil2_count) {
      const int item2 ⇐ Item_of_CIL(cil2, cil2_ix);
      Item_of_CIL(new_cil, new_cil_ix) ⇐ item2;
      cil2_ix++;
      new_cil_ix++;
    }
    Count_of_CIL(new_cil) ⇐ new_cil_ix;
    return cil_buffer_add(cilar);
  }
```

1198. Merge *int* `new_element` into an a CIL already in the CILAR. Optimized for the case where the CIL already includes `new_element`, in which case it returns Λ.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE CIL cil_merge_one(CILAR cilar, CIL cil, int new_element)
  {
    const int cil_count ⇐ Count_of_CIL(cil);
    CIL new_cil ⇐ cil_buffer_reserve(cilar, cil_count + 1);
    int new_cil_ix ⇐ 0;
    int cil_ix ⇐ 0;
    while (cil_ix < cil_count) {
      const int cil_item ⇐ Item_of_CIL(cil, cil_ix);
```

```
      if (cil_item ≡ new_element) {    /* new_element is already in cil, so we just
            return cil. It is OK to abandon the CIL in progress */
        return Λ;
      }
      if (cil_item > new_element) break;
      Item_of_CIL(new_cil, new_cil_ix) ⟸ cil_item;
      cil_ix++;
      new_cil_ix++;
    }
    Item_of_CIL(new_cil, new_cil_ix) ⟸ new_element;
    new_cil_ix++;
    while (cil_ix < cil_count) {
      const int cil_item ⟸ Item_of_CIL(cil, cil_ix);
      Item_of_CIL(new_cil, new_cil_ix) ⟸ cil_item;
      cil_ix++;
      new_cil_ix++;
    }
    Count_of_CIL(new_cil) ⟸ new_cil_ix;
    return cil_buffer_add(cilar);
  }
```

1199. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE_NOT_INLINE int cil_cmp(const void *ap, const void *bp, void
        *param UNUSED)
  {
    int ix;
    CIL cil1 ⟸ (CIL) ap;
    CIL cil2 ⟸ (CIL) bp;
    int count1 ⟸ Count_of_CIL(cil1);
    int count2 ⟸ Count_of_CIL(cil2);
    if (count1 ≠ count2) {
      return count1 > count2 ? 1 : −1;
    }
    for (ix ⟸ 0; ix < count1; ix++) {
      const int item1 ⟸ Item_of_CIL(cil1, ix);
      const int item2 ⟸ Item_of_CIL(cil2, ix);
      if (item1 ≡ item2) continue;
      return item1 > item2 ? 1 : −1;
    }
    return 0;
  }
```

1200. Per-Earley-set list (PSL) code. There are several cases where Marpa needs to look up a triple $\langle s, s', k\rangle$, where s and s' are earlemes, and $0 < k < n$, where n is a reasonably small constant, such as the number of AHM's. Earley items, or-nodes and and-nodes are examples.

1201. Lookup for Earley items needs to be $O(1)$ to justify Marpa's time complexity claims. Setup of the parse bocage for evaluation is not parsing in the strict sense, but makes sense to have it meet the same time complexity claims.

1202. To obtain $O(1)$, Marpa uses a special data structure, the Per-Earley-Set List. The Per-Earley-Set Lists rely on the following being true:

- It can be arranged so that only one s' is being considered at a time, so that we are in fact looking up a duple $\langle s, k\rangle$.
- In all cases of interest we will have pointers available that take us directly to all of the Earley sets involved, so that lookup of the data for an Earley set is $O(1)$.
- The value of k is always less than a constant. Therefore any reasonable algorithm for the search and insertion of k is $O(1)$.

1203. The idea is that each Earley set has a list of values for all the keys k. We arrange to consider only one Earley set s at a time. A pointer takes us to the Earley set s' in $O(1)$ time. Each Earley set has a list of values indexed by k. Since this list is of a size less than a constant, search and insertion in it is $O(1)$. Thus each search and insertion for the triple $\langle s, s', k\rangle$ takes $O(1)$ time.

1204. In understanding how the PSL's are used, it is important to keep in mind that the PSL's are kept in Earley sets as a convenience, and that the semantic relation of the Earley set to the data structure being tracked by the PSL is not important in the choice of where the PSL goes. All data structures tracked by PSL's belong semantically more to the Earley set of their dot earleme than any other, but for the time complexity hack to work, that must be held constand while another Earley set is the one which varies. In the case of Earley items and or-nodes, the varying Earley set is the origin. In the case of and-nodes, the origin Earley set is also held constant, and the Earley set of the middle earleme is the variable.

1205. The PSL's are kept in a linked list. Each contains `Size_of_PSL` *void* ∗'s. `t_owner` is the address of the location that "owns" this PSL. That location will be NULL'ed when deallocating.

⟨ Private incomplete structures 107 ⟩ +≡
    struct s_per_earley_set_list;
    typedef struct s_per_earley_set_list ∗*PSL*;

1206. *#define* `Sizeof_PSL(psar)` (*sizeof* (`PSL_Object`) + ((*size_t*) `psar→t_psl_length` − 1) ∗ *sizeof* (*void* ∗))
#define `PSL_Datum(psl, i)` ((`psl`)→`t_data`[(*i*)])

⟨ Private structures 48 ⟩ +≡

```
struct s_per_earley_set_list {
  PSL t_prev;
  PSL t_next;
  PSL *t_owner;
  void *t_data[1];
};
typedef struct s_per_earley_set_list PSL_Object;
```

1207. The per-Earley-set lists are allcated from per-Earley-set arenas.

⟨ Private incomplete structures 107 ⟩ +≡

```
struct s_per_earley_set_arena;
typedef struct s_per_earley_set_arena *PSAR;
```

1208. The "dot" PSAR is to track earley items whose origin or current earleme is at the "dot" location, that is, the current Earley set. The "predict" PSAR is to track earley items for predictions at locations other than the current earleme. The "predict" PSAR is used for predictions which result from scanned items. Since they are predictions, their current Earley set and origin are at the same earleme. This earleme will be somewhere after the current earleme.

⟨ Private structures 48 ⟩ +≡

```
struct s_per_earley_set_arena {
  int t_psl_length;
  PSL t_first_psl;
  PSL t_first_free_psl;
};
typedef struct s_per_earley_set_arena PSAR_Object;
```

1209. #define Dot_PSAR_of_R(r) (&(r)→t_dot_psar_object)

⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
PSAR_Object t_dot_psar_object;
```

1210. ⟨ Initialize dot PSAR 1210 ⟩ ≡

```
{
  if (G_is_Trivial(g)) {
    psar_safe(Dot_PSAR_of_R(r));
  }
  else {
    psar_init(Dot_PSAR_of_R(r), AHM_Count_of_G(g));
  }
}
```

This code is used in section 551.

1211. ⟨ Destroy recognizer elements 561 ⟩ +≡

```
psar_destroy(Dot_PSAR_of_R(r));
```

1212. Create a "safe" PSAR. A "safe" data structure is not considered initialized, and will need to be initialized before use. But the destructor may "safely" be called on it.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE void psar_safe(const PSAR psar)
{
  psar→t_psl_length ⟸ 0;
  psar→t_first_psl ⟸ psar→t_first_free_psl ⟸ Λ;
}
```

1213. ⟨ Function definitions 41 ⟩ +≡
```
PRIVATE void psar_init(const PSAR psar, int length)
{
  psar→t_psl_length ⟸ length;
  psar→t_first_psl ⟸ psar→t_first_free_psl ⟸ psl_new(psar);
}
```

1214. ⟨ Function definitions 41 ⟩ +≡
```
PRIVATE void psar_destroy(const PSAR psar)
{
  PSL psl ⟸ psar→t_first_psl;
  while (psl) {
    PSL next_psl ⟸ psl→t_next;
    PSL *owner ⟸ psl→t_owner;
    if (owner) *owner ⟸ Λ;
    my_free(psl);
    psl ⟸ next_psl;
  }
}
```

1215. ⟨ Function definitions 41 ⟩ +≡
```
PRIVATE PSL psl_new(const PSAR psar)
{
  int i;
  PSL new_psl ⟸ my_malloc(Sizeof_PSL(psar));
  new_psl→t_next ⟸ Λ;
  new_psl→t_prev ⟸ Λ;
  new_psl→t_owner ⟸ Λ;
  for (i ⟸ 0; i < psar→t_psl_length; i++) {
    PSL_Datum(new_psl, i) ⟸ Λ;
  }
  return new_psl;
}
```

1216. To Do: This is temporary data and perhaps should be keep track of on a per-phase obstack.

```
#define Dot_PSL_of_YS(ys) ((ys)→t_dot_psl)
⟨ Widely aligned Earley set elements 632 ⟩ +≡
  PSL t_dot_psl;
```

1217. ⟨ Initialize Earley set 638 ⟩ +≡

```
  {
    set→t_dot_psl ⟸ Λ;
  }
```

1218. A PSAR reset nulls out the data in the PSL's. It is a moderately expensive operation, usually avoided by having the logic check for "stale" data. But when the PSAR is needed for a a different type of PSL data, one which will require different stale-detection logic, the old PSL data need to be nulled.

```
⟨ Function definitions 41 ⟩ +≡
  PRIVATE void psar_reset(const PSAR psar)
  {
    PSL psl ⟸ psar→t_first_psl;
    while (psl ∧ psl→t_owner) {
      int i;
      for (i ⟸ 0; i < psar→t_psl_length; i++) {
        PSL_Datum(psl, i) ⟸ Λ;
      }
      psl ⟸ psl→t_next;
    }
    psar_dealloc(psar);
  }
```

1219. A PSAR dealloc removes an owner's claim to the all of its PSLs, and puts them back on the free list. It does **not** null out the stale PSL items.

1220. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void psar_dealloc(const PSAR psar)
  {
    PSL psl ⟸ psar→t_first_psl;
    while (psl) {
      PSL *owner ⟸ psl→t_owner;
      if (¬owner) break;
      (*owner) ⟸ Λ;
      psl→t_owner ⟸ Λ;
      psl ⟸ psl→t_next;
    }
    psar→t_first_free_psl ⟸ psar→t_first_psl;
  }
```

1221. This function "claims" a PSL. The address of the claimed PSL and the PSAR from which to claim it are arguments. The caller must ensure that there is not a PSL already at the claiming address.

1222. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void psl_claim(PSL *const psl_owner, const PSAR psar)
{
  PSL new_psl ⟸ psl_alloc(psar);
  (*psl_owner) ⟸ new_psl;
  new_psl→t_owner ⟸ psl_owner;
}
```

1223. ⟨ Function definitions 41 ⟩ +≡

```
PRIVATE PSL psl_claim_by_es(PSAR or_psar, struct s_bocage_setup_per_ys
        *per_ys_data, YSID ysid)
{
  PSL *psl_owner ⟸ &(per_ys_data[ysid].t_or_psl);
  if (¬*psl_owner) psl_claim(psl_owner, or_psar);
  return *psl_owner;
}
```

1224. This function "allocates" a PSL. It gets a free PSL from the PSAR. There must always be at least one free PSL in a PSAR. This function replaces the allocated PSL with a new free PSL when necessary.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE PSL psl_alloc(const PSAR psar)
{
  PSL free_psl ⟸ psar→t_first_free_psl;
  PSL next_psl ⟸ free_psl→t_next;
  if (¬next_psl) {
    next_psl ⟸ free_psl→t_next ⟸ psl_new(psar);
    next_psl→t_prev ⟸ free_psl;
  }
  psar→t_first_free_psl ⟸ next_psl;
  return free_psl;
}
```

1225. Obstacks. `libmarpa` uses the system malloc, either directly or indirectly. Indirect use comes via obstacks. Obstacks are more efficient, but limit the ability to resize memory, and to control the lifetime of the memory.

1226. Marpa makes extensive use of its own implementation of obstacks. Marpa's obstacks are based on ideas that originate with GNU's obstacks. Much of the memory allocated in `libmarpa` is

- In individual allocations less than 4K, often considerable less.
- Once created, are kept for the entire life of the either the grammar or the recognizer.
- Once created, is never resized. For these, obstacks are perfect. `libmarpa`'s grammar has an obstacks. Small allocations needed for the lifetime of the grammar are allocated on these as the grammar object is built. All these allocations are are conveniently and quickly deallocated when the grammar's obstack is destroyed along with its parent grammar.

1227. External failure reports. Most of `libmarpa`'s external functions return failure under one or more circumstances — for example, they may have been called incorrectly. Many of the external routines share failure logic in common. I found it convenient to gather much of this logic here. All the logic in this section expects `failure_indication` to be set in the scope in which it is used. All failures treated in this section are hard failures.

1228. Routines returning pointers typically use Λ as both the soft and hard failure indicator.

⟨ Return Λ on failure 1228 ⟩ ≡

```
void *const failure_indicator ⟸ Λ;
```

This code is used in sections 551, 653, 942, 977, 1025, and 1083.

1229. Routines returning integer value use −2 as the general failure indicator.

⟨ Return −2 on failure 1229 ⟩ ≡

```
const int failure_indicator ⟸ −2;
```

This code is used in sections 63, 74, 80, 81, 94, 95, 99, 102, 118, 119, 149, 152, 153, 163, 164, 165, 168, 171, 174, 177, 181, 182, 185, 188, 189, 190, 193, 194, 195, 198, 199, 200, 207, 211, 226, 229, 232, 235, 240, 243, 248, 249, 252, 261, 262, 270, 272, 273, 278, 279, 282, 283, 290, 293, 298, 302, 306, 309, 312, 316, 319, 322, 324, 333, 335, 337, 343, 346, 352, 355, 358, 361, 364, 368, 412, 478, 479, 481, 483, 543, 544, 545, 567, 582, 583, 586, 588, 590, 592, 604, 605, 612, 639, 640, 641, 642, 710, 737, 802, 821, 822, 832, 833, 836, 837, 955, 959, 970, 987, 991, 994, 995, 999, 1008, 1039, 1066, 1096, 1099, 1105, 1106, 1107, 1108, 1109, 1110, 1112, 1262, 1263, 1264, 1266, 1271, 1273, 1276, 1278, 1279, 1280, 1283, 1285, 1286, 1287, 1292, 1295, 1297, 1300, 1302, 1305, 1308, 1309, 1311, 1313, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1329, 1330, 1332, 1334, 1335, 1336, 1337, 1338, 1339, 1342, 1343, 1344, 1345, 1346, 1347, 1348, and 1355.

1230. Grammar failures. g is assumed to be the value of the relevant grammar, when one is required.

⟨ Fail if precomputed 1230 ⟩ ≡

```
if (_MARPA_UNLIKELY(G_is_Precomputed(g))) {
  MARPA_ERROR(MARPA_ERR_PRECOMPUTED);
  return failure_indicator;
}
```

This code is used in sections 81, 95, 153, 182, 189, 190, 194, 195, 199, 200, 261, 262, 279, 283, 368, 543, and 545.

1231. ⟨ Fail if not precomputed 1231 ⟩ ≡

```
if (_MARPA_UNLIKELY(¬G_is_Precomputed(g))) {
  MARPA_ERROR(MARPA_ERR_NOT_PRECOMPUTED);
  return failure_indicator;
}
```

This code is used in sections 168, 174, 177, 185, 229, 232, 235, 306, 309, 312, 316, 319, 333, 335, 337, 343, 346, 352, 355, 358, 412, 478, 479, 481, 483, and 551.

1232. ⟨ Fail if `xsy_id` is malformed 1232 ⟩ ≡

```
if (_MARPA_UNLIKELY(XSYID_is_Malformed(xsy_id))) {
  MARPA_ERROR(MARPA_ERR_INVALID_SYMBOL_ID);
  return failure_indicator;
}
```

This code is used in sections 81, 149, 152, 153, 164, 165, 168, 171, 174, 177, 181, 182, 185, 188, 189, 190, 193, 194, 195, 198, 199, 200, 207, 211, 583, 586, 588, 590, 592, 1105, and 1107.

1233. Fail with −1 for well-formed, but non-existent symbol ID.

⟨ Soft fail if `xsy_id` does not exist 1233 ⟩ ≡

```
if (_MARPA_UNLIKELY(¬XSYID_of_G_Exists(xsy_id))) {
  MARPA_ERROR(MARPA_ERR_NO_SUCH_SYMBOL_ID);
  return −1;
}
```

This code is used in sections 81, 149, 164, 165, 168, 171, 174, 177, 181, 182, 185, 188, 189, 190, 193, 194, 195, 198, 199, 200, 207, 211, 586, 588, 590, 592, 1105, and 1107.

1234. ⟨ Fail if `xsy_id` does not exist 1234 ⟩ ≡

```
if (_MARPA_UNLIKELY(¬XSYID_of_G_Exists(xsy_id))) {
  MARPA_ERROR(MARPA_ERR_NO_SUCH_SYMBOL_ID);
  return failure_indicator;
}
```

This code is used in sections 152, 153, and 583.

1235. ⟨ Fail if `nsy_id` is invalid 1235 ⟩ ≡

```
if (_MARPA_UNLIKELY(¬nsy_is_valid(g, nsy_id))) {
  MARPA_ERROR(MARPA_ERR_INVALID_NSYID);
  return failure_indicator;
}
```

This code is used in sections 229, 232, 235, 240, 243, 248, 249, and 252.

1236. ⟨ Fail if `nsy_id` is malformed 1236 ⟩ ≡

```
if (_MARPA_UNLIKELY(NSYID_is_Malformed(nsy_id))) {
  MARPA_ERROR(MARPA_ERR_INVALID_SYMBOL_ID);
  return failure_indicator;
}
```

This code is used in section 1283.

1237. Fail with −1 for well-formed, but non-existent symbol ID.

⟨ Soft fail if `nsy_id` does not exist 1237 ⟩ ≡

```
if (_MARPA_UNLIKELY(¬NSYID_of_G_Exists(nsy_id))) {
  MARPA_ERROR(MARPA_ERR_NO_SUCH_SYMBOL_ID);
  return −1;
}
```

This code is used in section 1283.

1238. ⟨ Fail if `irl_id` is invalid 1238 ⟩ ≡

```
if (_MARPA_UNLIKELY(¬IRLID_of_G_is_Valid(irl_id))) {
  MARPA_ERROR(MARPA_ERR_INVALID_IRLID);
  return failure_indicator;
}
```

This code is used in sections 324, 333, 335, 337, 343, 346, 352, 355, 358, 361, 364, and 412.

1239. For well-formed, but non-existent rule ids, sometimes we want hard failures, and sometimes soft (−1).

⟨ Soft fail if xrl_id does not exist 1239 ⟩ ≡
```
if (_MARPA_UNLIKELY(¬XRLID_of_G_Exists(xrl_id))) {
  MARPA_ERROR(MARPA_ERR_NO_SUCH_RULE_ID);
  return −1;
}
```
This code is used in sections 270, 272, 273, 282, 283, 298, 302, 306, 309, 312, 316, 319, 322, 545, 1109, and 1110.

1240. ⟨ Fail if xrl_id does not exist 1240 ⟩ ≡
```
if (_MARPA_UNLIKELY(¬XRLID_of_G_Exists(xrl_id))) {
  MARPA_ERROR(MARPA_ERR_NO_SUCH_RULE_ID);
  return failure_indicator;
}
```
This code is used in sections 278, 279, 290, and 293.

1241.

⟨ Fail if xrl_id is malformed 1241 ⟩ ≡
```
if (_MARPA_UNLIKELY(XRLID_is_Malformed(xrl_id))) {
  MARPA_ERROR(MARPA_ERR_INVALID_RULE_ID);
  return failure_indicator;
}
```
This code is used in sections 270, 272, 273, 278, 279, 282, 283, 290, 293, 298, 302, 306, 309, 312, 316, 319, 322, 545, 1109, and 1110.

1242. ⟨ Fail if zwaid does not exist 1242 ⟩ ≡
```
if (_MARPA_UNLIKELY(¬ZWAID_of_G_Exists(zwaid))) {
  MARPA_ERROR(MARPA_ERR_NO_SUCH_ASSERTION_ID);
  return failure_indicator;
}
```
This code is used in sections 545, 821, and 822.

1243.

⟨ Fail if zwaid is malformed 1243 ⟩ ≡
```
if (_MARPA_UNLIKELY(ZWAID_is_Malformed(zwaid))) {
  MARPA_ERROR(MARPA_ERR_INVALID_ASSERTION_ID);
  return failure_indicator;
}
```
This code is used in sections 545, 821, and 822.

1244. "AIMID" in the error code name is a legacy of a previous implementation. The name of the error code must be kept the same for backward compatibility.

⟨ Fail if item_id is invalid 1244 ⟩ ≡
```
if (_MARPA_UNLIKELY(¬ahm_is_valid(g, item_id))) {
  MARPA_ERROR(MARPA_ERR_INVALID_AIMID);
```

```
    return failure_indicator;
  }
```

This code is used in sections 479, 481, and 483.

1245. Recognizer failures. r is assumed to be the value of the relevant recognizer, when one is required.

⟨ Fail if recognizer started 1245 ⟩ ≡

```
  if (_MARPA_UNLIKELY(Input_Phase_of_R(r) ≠ R_BEFORE_INPUT)) {
    MARPA_ERROR(MARPA_ERR_RECCE_STARTED);
    return failure_indicator;
  }
```

This code is used in sections 605 and 710.

1246. ⟨ Fail if recognizer not started 1246 ⟩ ≡

```
  if (_MARPA_UNLIKELY(Input_Phase_of_R(r) ≡ R_BEFORE_INPUT)) {
    MARPA_ERROR(MARPA_ERR_RECCE_NOT_STARTED);
    return failure_indicator;
  }
```

This code is used in sections 582, 583, 639, 640, 832, 833, 836, 837, 942, 1248, 1264, and 1266.

1247. ⟨ Fail if recognizer not accepting input 1247 ⟩ ≡

```
  if (_MARPA_UNLIKELY(Input_Phase_of_R(r) ≠ R_DURING_INPUT)) {
    MARPA_ERROR(MARPA_ERR_RECCE_NOT_ACCEPTING_INPUT);
    return failure_indicator;
  }
  if (_MARPA_UNLIKELY(¬R_is_Consistent(r))) {
    MARPA_ERROR(MARPA_ERR_RECCE_IS_INCONSISTENT);
    return failure_indicator;
  }
```

This code is used in sections 737 and 802.

1248. ⟨ Fail if not trace-safe 1248 ⟩ ≡

  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Fail if recognizer not started 1246 ⟩

This code is used in sections 641, 642, 1262, 1263, 1271, 1273, 1276, 1278, 1279, 1280, 1283, 1285, 1286, 1287, 1292, 1295, 1297, 1300, 1302, 1305, 1308, 1309, 1311, and 1313.

1249. It is expected the first test, for mismatched headers, will be optimized completely out if the versions numbers are consistent.

⟨ Fail if fatal error 1249 ⟩ ≡

```
  if (HEADER_VERSION_MISMATCH) {
    MARPA_ERROR(MARPA_ERR_HEADERS_DO_NOT_MATCH);
    return failure_indicator;
  }
  if (_MARPA_UNLIKELY(¬IS_G_OK(g))) {
    MARPA_ERROR(g→t_error);
```

```
    return failure_indicator;
  }
```

This code is used in sections 63, 74, 80, 81, 94, 95, 99, 102, 119, 149, 152, 153, 168, 171, 174, 177, 181, 182, 185, 188, 189, 190, 193, 194, 195, 198, 199, 200, 226, 229, 232, 235, 261, 262, 270, 272, 273, 278, 279, 282, 283, 290, 293, 298, 302, 306, 309, 312, 316, 319, 333, 335, 337, 368, 543, 544, 545, 567, 582, 583, 586, 588, 590, 592, 604, 605, 612, 639, 640, 821, 822, 832, 833, 837, 942, 955, 959, 970, 977, 987, 991, 994, 995, 999, 1008, 1025, 1039, 1066, 1083, 1096, 1099, 1105, 1107, 1108, 1109, 1110, 1115, 1248, 1264, 1266, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1329, 1330, 1332, and 1341.

1250. The central error routine for the recognizer. There are two flags which control its behavior. One flag makes a error recognizer-fatal. When there is a recognizer-fatal error, all subsequent invocations of external functions for that recognizer object will fail. It is a design goal of libmarpa to leave as much discretion about error handling to the higher layers as possible. Because of this, even the most severe errors are not necessarily made recognizer-fatal. `libmarpa` makes an error recognizer-fatal only when the integrity of the recognizer object is so thorougly compromised that `libmarpa`'s external functions cannot proceed without risking internal memory errors, such as bus errors and segment violations. "Recognizer-fatal" status is thus, not a means of dictating to the higher layers that a `libmarpa` condition must be application-fatal, but a way of preventing a recognizer error from becoming application-fatal without the application's consent.

```
#define FATAL_FLAG (#1U)
```

1251. Several convenience macros are provided. These are easier and less error-prone than specifying the flags. Not being error-prone is important since there are many calls to `r_error` in the code.

```
#define MARPA_DEV_ERROR(message)
        (set_error(g, MARPA_ERR_DEVELOPMENT, (message), 0U))
#define MARPA_INTERNAL_ERROR(message)
        (set_error(g, MARPA_ERR_INTERNAL, (message), 0U))
#define MARPA_ERROR(code)  (set_error(g, (code), Λ, 0U))
#define MARPA_FATAL(code)  (set_error(g, (code), Λ, FATAL_FLAG))
```

1252. Not inlined. `r_error` occurs in the code quite often, but `r_error` should actually be invoked only in exceptional circumstances. In this case space clearly is much more important than speed.

⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE_NOT_INLINE void set_error(GRAMMAR g, Marpa_Error_Code
          code, const char *message, unsigned int flags)
  {
    g→t_error ⇐ code;
    g→t_error_string ⇐ message;
    if (flags & FATAL_FLAG) g→t_is_ok ⇐ 0;
  }
```

1253. If this is called when Libmarpa is in a "not OK" state, it means very bad things are happening – possibly memory overwrites. So we do not attempt much. We return, leaving the error code as is, unless it is MARPA_ERR_NONE. Since this would be completely misleading, we take a chance and try to change it to MARPA_ERR_I_AM_NOT_OK.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE Marpa_Error_Code clear_error(GRAMMAR g)
{
  if (¬IS_G_OK(g)) {
    if (g→t_error ≡ MARPA_ERR_NONE) g→t_error ⟸ MARPA_ERR_I_AM_NOT_OK;
    return g→t_error;
  }
  g→t_error ⟸ MARPA_ERR_NONE;
  g→t_error_string ⟸ Λ;
  return MARPA_ERR_NONE;
}
```

1254. Messages and logging. There are a few cases in which it is not appropriate to rely on the upper layers for error messages. These cases include serious internal problems, memory allocation failures, and debugging.

1255. Memory allocation.

1256. Most of the memory allocation logic is in other documents. Here is its potentially public interface, the configurable failure handler. By default, a memory allocation failure inside the Marpa library is a fatal error.

1257. The default handler can be changed, but this is not documented for two reasons. First, it is not tested. Second, What else an application can do is not at all clear. Nearly universal practice is to treat memory allocation errors as irrecoverable and fatal. These functions all return *void* ∗ in order to avoid compiler warnings about void returns.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE_NOT_INLINE void *marpa__default_out_of_memory(void)
{
  abort();
  return Λ;    /* to prevent warnings on some compilers */
}
void *(*const marpa__out_of_memory)(void) ⟸ marpa__default_out_of_memory;
```

1258. ⟨ Debugging variable declarations 1258 ⟩ ≡
```
extern void *(*const marpa__out_of_memory)(void);
```
See also section 1364.

This code is used in sections 1384 and 1387.

1259. ⟨ Public typedefs 91 ⟩ +≡
```
typedef const char *Marpa_Message_ID;
```

1260. Trace functions.

The "trace " functions were designed for just that – use in tracing and diagnostics. They were not designed for use in production – they lack some of the efficiency and coverage needed. For the recognizer's trace functions, this intent is, in Kollos, to replace them with the "looker" functions.

Many of the trace functions use a "trace Earley set" which is tracked on a per-recognizer basis. The "trace Earley set" is tracked separately from the current Earley set for the parse. The two may coincide, but should not be confused.

⟨ Widely aligned recognizer elements 558 ⟩ +≡
```
struct s_earley_set *t_trace_earley_set;
```

1261. ⟨ Initialize recognizer elements 554 ⟩ +≡
```
r→t_trace_earley_set ⟸ Λ;
```

1262. ⟨ Function definitions 41 ⟩ +≡
```
Marpa_Earley_Set_ID _marpa_r_trace_earley_set(Marpa_Recognizer r)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  YS trace_earley_set ⟸ r→t_trace_earley_set;
  ⟨ Fail if not trace-safe 1248 ⟩
  if (¬trace_earley_set) {
    MARPA_ERROR(MARPA_ERR_NO_TRACE_YS);
    return failure_indicator;
  }
  return Ord_of_YS(trace_earley_set);
}
```

1263. ⟨ Function definitions 41 ⟩ +≡
```
Marpa_Earley_Set_ID marpa_r_latest_earley_set(Marpa_Recognizer r)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if not trace-safe 1248 ⟩
  if (G_is_Trivial(g)) return 0;
  return Ord_of_YS(Latest_YS_of_R(r));
}
```

1264. ⟨ Function definitions 41 ⟩ +≡
```
Marpa_Earleme marpa_r_earleme(Marpa_Recognizer r, Marpa_Earley_Set_ID set_id)
{
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Return −2 on failure 1229 ⟩
  YS earley_set;
```

```
    ⟨ Fail if recognizer not started 1246 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    if (set_id < 0) {
      MARPA_ERROR(MARPA_ERR_INVALID_LOCATION);
      return failure_indicator;
    }
    r_update_earley_sets(r);
    if (¬YS_Ord_is_Valid(r, set_id)) {
      MARPA_ERROR(MARPA_ERR_NO_EARLEY_SET_AT_LOCATION);
      return failure_indicator;
    }
    earley_set ⟸ YS_of_R_by_Ord(r, set_id);
    return Earleme_of_YS(earley_set);
  }
```

1265. Note that this trace function returns the earley set size of the **current earley set**. It includes rejected *YIM*'s.

1266. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_r_earley_set_size(Marpa_Recognizer r, Marpa_Earley_Set_ID set_id)
  {
    ⟨ Return −2 on failure 1229 ⟩
    YS earley_set;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if recognizer not started 1246 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    r_update_earley_sets(r);
    if (¬YS_Ord_is_Valid(r, set_id)) {
      MARPA_ERROR(MARPA_ERR_INVALID_LOCATION);
      return failure_indicator;
    }
    earley_set ⟸ YS_of_R_by_Ord(r, set_id);
    return YIM_Count_of_YS(earley_set);
  }
```

1267. Many of the trace functions use a “trace Earley item” which is tracked on a per-recognizer basis.

⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
  YIM t_trace_earley_item;
```

1268. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
  r→t_trace_earley_item ⟸ Λ;
```

1269. This function sets the trace Earley set to the one indicated by the ID of the argument. On success, the earleme of the new trace Earley set is returned.

1270. Various other trace data depends on the Earley set, and must be consistent with it. This function clears all such data, unless it is called while the recognizer is in a trace-unsafe state (initial, fatal, etc.) or unless the the Earley set requested by the argument is already the trace Earley set. On failure because the ID is for a non-existent Earley set which does not exist, −1 is returned. The upper levels may choose to treat this as a soft failure. This may be treated as a soft failure by the upper levels. On failure because the ID is illegal (less than zero) or for other failures, −2 is returned. The upper levels may choose to treat these as hard failures.

1271. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Earleme _marpa_r_earley_set_trace(Marpa_Recognizer r, Marpa_Earley_Set_ID
        set_id){ YS earley_set;
    const int es_does_not_exist ⟸ −1; ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    if (r→t_trace_earley_set ∧ Ord_of_YS(r→t_trace_earley_set) ≡ set_id) {
        /* If the set is already the current earley set, return successfully without
           resetting any of the dependant data */
      return Earleme_of_YS(r→t_trace_earley_set);
    }
    ⟨ Clear trace Earley set dependent data 1272 ⟩
    if (set_id < 0) {
      MARPA_ERROR(MARPA_ERR_INVALID_LOCATION);
      return failure_indicator;
    }
    r_update_earley_sets(r);
    if (set_id ≥ MARPA_DSTACK_LENGTH(r→t_earley_set_stack)) {
      return es_does_not_exist;
    }
    earley_set ⟸ YS_of_R_by_Ord(r, set_id);
    r→t_trace_earley_set ⟸ earley_set;
    return Earleme_of_YS(earley_set); }
```

1272. ⟨ Clear trace Earley set dependent data 1272 ⟩ ≡

```
  {
    r→t_trace_earley_set ⟸ Λ;
    trace_earley_item_clear(r);
    ⟨ Clear trace postdot item data 1284 ⟩
  }
```

This code is used in sections 1271 and 1273.

1273. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_AHM_ID _marpa_r_earley_item_trace(Marpa_Recognizer r,
        Marpa_Earley_Item_ID item_id)
  {
```

```
    const int yim_does_not_exist ⇐ −1;
    ⟨ Return −2 on failure 1229 ⟩
    YS trace_earley_set;
    YIM earley_item;
    YIM *earley_items;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    trace_earley_set ⇐ r→t_trace_earley_set;
    if (¬trace_earley_set) {
      ⟨ Clear trace Earley set dependent data 1272 ⟩
      MARPA_ERROR(MARPA_ERR_NO_TRACE_YS);
      return failure_indicator;
    }
    trace_earley_item_clear(r);
    if (item_id < 0) {
      MARPA_ERROR(MARPA_ERR_YIM_ID_INVALID);
      return failure_indicator;
    }
    if (item_id ≥ YIM_Count_of_YS(trace_earley_set)) {
      return yim_does_not_exist;
    }
    earley_items ⇐ YIMs_of_YS(trace_earley_set);
    earley_item ⇐ earley_items[item_id];
    r→t_trace_earley_item ⇐ earley_item;
    return AHMID_of_YIM(earley_item);
  }
```

1274. Clear all the data elements specifically for the trace Earley item. The difference between this code and `trace_earley_item_clear` is that `trace_earley_item_clear` also clears the source link.

```
⟨ Clear trace Earley item data 1274 ⟩ ≡
  r→t_trace_earley_item ⇐ Λ;
```

This code is used in sections 1275, 1276, and 1285.

1275. ⟨ Function definitions 41 ⟩ +≡

```
  PRIVATE void trace_earley_item_clear(RECCE r)
  {
    ⟨ Clear trace Earley item data 1274 ⟩
    trace_source_link_clear(r);
  }
```

1276. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Earley_Set_ID _marpa_r_earley_item_origin(Marpa_Recognizer r)
{
  ⟨ Return −2 on failure 1229 ⟩
  YIM item ⟸ r→t_trace_earley_item;
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if not trace-safe 1248 ⟩
  if (¬item) {
    ⟨ Clear trace Earley item data 1274 ⟩
    MARPA_ERROR(MARPA_ERR_NO_TRACE_YIM);
    return failure_indicator;
  }
  return Origin_Ord_of_YIM(item);
}
```

1277. Leo item (LIM) trace functions. The functions in this section are all accessors. The trace Leo item is selected by setting the trace postdot item to a Leo item.

1278. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Symbol_ID _marpa_r_leo_predecessor_symbol(Marpa_Recognizer r)
{
  const Marpa_Symbol_ID no_predecessor ⟸ −1;
  ⟨ Return −2 on failure 1229 ⟩
  PIM postdot_item ⟸ r→t_trace_postdot_item;
  LIM predecessor_leo_item;
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if not trace-safe 1248 ⟩
  if (¬postdot_item) {
    MARPA_ERROR(MARPA_ERR_NO_TRACE_PIM);
    return failure_indicator;
  }
  if (YIM_of_PIM(postdot_item)) {
    MARPA_ERROR(MARPA_ERR_PIM_IS_NOT_LIM);
    return failure_indicator;
  }
  predecessor_leo_item ⟸ Predecessor_LIM_of_LIM(LIM_of_PIM(postdot_item));
  if (¬predecessor_leo_item) return no_predecessor;
  return Postdot_NSYID_of_LIM(predecessor_leo_item);
}
```

1279. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Earley_Set_ID _marpa_r_leo_base_origin(Marpa_Recognizer r)
{
  const JEARLEME pim_is_not_a_leo_item ⇐ −1;
  ⟨ Return −2 on failure 1229 ⟩
  PIM postdot_item ⇐ r→t_trace_postdot_item;
  ⟨ Unpack recognizer objects 560 ⟩
  YIM base_earley_item;
  ⟨ Fail if not trace-safe 1248 ⟩
  if (¬postdot_item) {
    MARPA_ERROR(MARPA_ERR_NO_TRACE_PIM);
    return failure_indicator;
  }
  if (YIM_of_PIM(postdot_item)) return pim_is_not_a_leo_item;
  base_earley_item ⇐ Trailhead_YIM_of_LIM(LIM_of_PIM(postdot_item));
  return Origin_Ord_of_YIM(base_earley_item);
}
```

1280. Actually return AHM ID, not the obsolete AHFA ID.

⟨ Function definitions 41 ⟩ +≡

```
Marpa_AHM_ID _marpa_r_leo_base_state(Marpa_Recognizer r)
{
  const JEARLEME pim_is_not_a_leo_item ⇐ −1;
  ⟨ Return −2 on failure 1229 ⟩
  PIM postdot_item ⇐ r→t_trace_postdot_item;
  YIM base_earley_item;
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if not trace-safe 1248 ⟩
  if (¬postdot_item) {
    MARPA_ERROR(MARPA_ERR_NO_TRACE_PIM);
    return failure_indicator;
  }
  if (YIM_of_PIM(postdot_item)) return pim_is_not_a_leo_item;
  base_earley_item ⇐ Trailhead_YIM_of_LIM(LIM_of_PIM(postdot_item));
  return AHMID_of_YIM(base_earley_item);
}
```

1281. PIM Trace functions. Many of the trace functions use a "trace postdot item". This is tracked on a per-recognizer basis.

⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
PIM *t_trace_pim_nsy_p;
PIM t_trace_postdot_item;
```

1282. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
r→t_trace_pim_nsy_p ⇐ Λ;
r→t_trace_postdot_item ⇐ Λ;
```

1283. marpa_r_postdot_symbol_trace takes a recognizer and an internal symbol ID as an argument. (Note untested previous versions used an external symbol ID, which was inconsistent with the rest of the interface.)

marpa_r_postdot_symbol_trace sets the trace postdot item to the first postdot item for the symbol ID. If there is no postdot item for that symbol ID, it returns −1. On failure for other reasons, it returns −2 and clears the trace postdot item.

⟨ Function definitions 41 ⟩ +≡

```
Marpa_Symbol_ID _marpa_r_postdot_symbol_trace(Marpa_Recognizer
        r, Marpa_Symbol_ID nsy_id)
{
  ⟨ Return −2 on failure 1229 ⟩
  YS current_ys ⇐ r→t_trace_earley_set;
  PIM *pim_nsy_p;
  PIM pim;
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Clear trace postdot item data 1284 ⟩
  ⟨ Fail if not trace-safe 1248 ⟩
  ⟨ Fail if nsy_id is malformed 1236 ⟩
  ⟨ Soft fail if nsy_id does not exist 1237 ⟩
  if (¬current_ys) {
    MARPA_ERROR(MARPA_ERR_NO_TRACE_YS);
    return failure_indicator;
  }
  pim_nsy_p ⇐ PIM_NSY_P_of_YS_by_NSYID(current_ys, nsy_id);
  pim ⇐ *pim_nsy_p;
  if (¬pim) return −1;
  r→t_trace_pim_nsy_p ⇐ pim_nsy_p;
  r→t_trace_postdot_item ⇐ pim;
  return nsy_id;
}
```

1284. ⟨ Clear trace postdot item data 1284 ⟩ ≡

```
r→t_trace_pim_nsy_p ⇐ Λ;
r→t_trace_postdot_item ⇐ Λ;
```

This code is used in sections 1272, 1283, 1285, and 1286.

1285. Set trace postdot item to the first in the trace Earley set, and return its postdot symbol ID. If the trace Earley set has no postdot items, return -1 and clear the trace postdot item. On other failures, return -2 and clear the trace postdot item.

⟨ Function definitions 41 ⟩ +≡

```
Marpa_Symbol_ID _marpa_r_first_postdot_item_trace(Marpa_Recognizer r)
```

```
{
  ⟨ Return −2 on failure 1229 ⟩
  YS current_earley_set ⇐ r→t_trace_earley_set;
  PIM pim;
  ⟨ Unpack recognizer objects 560 ⟩
  PIM *pim_nsy_p;
  ⟨ Clear trace postdot item data 1284 ⟩
  ⟨ Fail if not trace-safe 1248 ⟩
  if (¬current_earley_set) {
    ⟨ Clear trace Earley item data 1274 ⟩
    MARPA_ERROR(MARPA_ERR_NO_TRACE_YS);
    return failure_indicator;
  }
  if (current_earley_set→t_postdot_sym_count ≤ 0) return −1;
  pim_nsy_p ⇐ current_earley_set→t_postdot_ary + 0;
  pim ⇐ pim_nsy_p[0];
  r→t_trace_pim_nsy_p ⇐ pim_nsy_p;
  r→t_trace_postdot_item ⇐ pim;
  return Postdot_NSYID_of_PIM(pim);
}
```

1286. Set the trace postdot item to the one after the current trace postdot item, and return its postdot symbol ID. If the current trace postdot item is the last, return -1 and clear the trace postdot item. On other failures, return -2 and clear the trace postdot item.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_Symbol_ID _marpa_r_next_postdot_item_trace(Marpa_Recognizer r)
  {
    const XSYID no_more_postdot_symbols ⇐ −1;
    ⟨ Return −2 on failure 1229 ⟩
    YS current_set ⇐ r→t_trace_earley_set;
    PIM pim;
    PIM *pim_nsy_p;
    ⟨ Unpack recognizer objects 560 ⟩
    pim_nsy_p ⇐ r→t_trace_pim_nsy_p;
    pim ⇐ r→t_trace_postdot_item;
    ⟨ Clear trace postdot item data 1284 ⟩
    if (¬pim_nsy_p ∨ ¬pim) {
      MARPA_ERROR(MARPA_ERR_NO_TRACE_PIM);
      return failure_indicator;
    }
    ⟨ Fail if not trace-safe 1248 ⟩
    if (¬current_set) {
      MARPA_ERROR(MARPA_ERR_NO_TRACE_YS);
```

```
      return failure_indicator;
    }
    pim ⇐ Next_PIM_of_PIM(pim);
    if (¬pim) {
        /* If no next postdot item for this symbol, then look at next symbol */
      pim_nsy_p++;
      if (pim_nsy_p − current_set→t_postdot_ary ≥
            current_set→t_postdot_sym_count) {
        return no_more_postdot_symbols;
      }
      pim ⇐ *pim_nsy_p;
    }
    r→t_trace_pim_nsy_p ⇐ pim_nsy_p;
    r→t_trace_postdot_item ⇐ pim;
    return Postdot_NSYID_of_PIM(pim);
  }
```

1287. ⟨ Function definitions 41 ⟩ +≡

```
  Marpa_Symbol_ID _marpa_r_postdot_item_symbol(Marpa_Recognizer r)
  {
    ⟨ Return −2 on failure 1229 ⟩
    PIM postdot_item ⇐ r→t_trace_postdot_item;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    if (¬postdot_item) {
      MARPA_ERROR(MARPA_ERR_NO_TRACE_PIM);
      return failure_indicator;
    }
    return Postdot_NSYID_of_PIM(postdot_item);
  }
```

1288. Link trace functions. Many trace functions track a "trace source link". There is only one of these, shared among all types of source link. It is reported as an error if a trace function is called when it is inconsistent with the type of the current trace source link.

⟨ Widely aligned recognizer elements 558 ⟩ +≡

```
  SRCL t_trace_source_link;
```

1289. ⟨ Bit aligned recognizer elements 562 ⟩ +≡

```
  BITFIELD t_trace_source_type:3;
```

1290. ⟨ Initialize recognizer elements 554 ⟩ +≡

```
  r→t_trace_source_link ⇐ Λ;
  r→t_trace_source_type ⇐ NO_SOURCE;
```

1291. Trace first token link.

1292. Set the trace source link to a token link, if there is one, otherwise clear the trace source link. Returns the symbol ID if there was a token source link, −1 if there was none, and −2 on some other kind of failure.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_Symbol_ID _marpa_r_first_token_link_trace(Marpa_Recognizer r)
  {
    ⟨ Return −2 on failure 1229 ⟩
    SRCL source_link;
    unsigned int source_type;
    YIM item ⟸ r→t_trace_earley_item;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    ⟨ Set item, failing if necessary 1306 ⟩
    source_type ⟸ Source_Type_of_YIM(item);
    switch (source_type) {
    case SOURCE_IS_TOKEN: r→t_trace_source_type ⟸ SOURCE_IS_TOKEN;
      source_link ⟸ SRCL_of_YIM(item);
      r→t_trace_source_link ⟸ source_link;
      return NSYID_of_SRCL(source_link);
    case SOURCE_IS_AMBIGUOUS:
      {
        source_link ⟸ LV_First_Token_SRCL_of_YIM(item);
        if (source_link) {
          r→t_trace_source_type ⟸ SOURCE_IS_TOKEN;
          r→t_trace_source_link ⟸ source_link;
          return NSYID_of_SRCL(source_link);
        }
      }
    }
    trace_source_link_clear(r);
    return −1;
  }
```

1293. Trace next token link.

1294. Set the trace source link to the next token link, if there is one. Otherwise clear the trace source link.

1295. Returns the symbol ID if there is a next token source link, −1 if there was none, and −2 on some other kind of failure.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_Symbol_ID _marpa_r_next_token_link_trace(Marpa_Recognizer r)
  {
```

```
    ⟨ Return −2 on failure 1229 ⟩
    SRCL source_link;
    YIM item;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    ⟨ Set item, failing if necessary 1306 ⟩
    if (r→t_trace_source_type ≠ SOURCE_IS_TOKEN) {
      trace_source_link_clear(r);
      MARPA_ERROR(MARPA_ERR_NOT_TRACING_TOKEN_LINKS);
      return failure_indicator;
    }
    source_link ⟸ Next_SRCL_of_SRCL(r→t_trace_source_link);
    if (¬source_link) {
      trace_source_link_clear(r);
      return −1;
    }
    r→t_trace_source_link ⟸ source_link;
    return NSYID_of_SRCL(source_link);
  }
```

1296. Trace first completion link.

1297. Set the trace source link to a completion link, if there is one, otherwise clear the completion source link. Returns the AHM ID (not the obsolete AHFA state ID) of the cause if there was a completion source link, −1 if there was none, and −2 on some other kind of failure.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_Symbol_ID _marpa_r_first_completion_link_trace(Marpa_Recognizer r)
  {
    ⟨ Return −2 on failure 1229 ⟩
    SRCL source_link;
    unsigned int source_type;
    YIM item ⟸ r→t_trace_earley_item;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    ⟨ Set item, failing if necessary 1306 ⟩
    switch ((source_type ⟸ Source_Type_of_YIM(item))) {
    case SOURCE_IS_COMPLETION:
      r→t_trace_source_type ⟸ SOURCE_IS_COMPLETION;
      source_link ⟸ SRCL_of_YIM(item);
      r→t_trace_source_link ⟸ source_link;
      return Cause_AHMID_of_SRCL(source_link);
    case SOURCE_IS_AMBIGUOUS:
      {
```

```
        source_link ⇐ LV_First_Completion_SRCL_of_YIM(item);
        if (source_link) {
          r→t_trace_source_type ⇐ SOURCE_IS_COMPLETION;
          r→t_trace_source_link ⇐ source_link;
          return Cause_AHMID_of_SRCL(source_link);
        }
      }
    }
    trace_source_link_clear(r);
    return −1;
  }
```

1298. Trace next completion link.

1299. Set the trace source link to the next completion link, if there is one. Otherwise clear the trace source link.

1300. Returns the cause AHM ID if there is a next completion source link, −1 if there was none, and −2 on some other kind of failure.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_Symbol_ID _marpa_r_next_completion_link_trace(Marpa_Recognizer r)
  {
    ⟨ Return −2 on failure 1229 ⟩
    SRCL source_link;
    YIM item;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    ⟨ Set item, failing if necessary 1306 ⟩
    if (r→t_trace_source_type ≠ SOURCE_IS_COMPLETION) {
      trace_source_link_clear(r);
      MARPA_ERROR(MARPA_ERR_NOT_TRACING_COMPLETION_LINKS);
      return failure_indicator;
    }
    source_link ⇐ Next_SRCL_of_SRCL(r→t_trace_source_link);
    if (¬source_link) {
      trace_source_link_clear(r);
      return −1;
    }
    r→t_trace_source_link ⇐ source_link;
    return Cause_AHMID_of_SRCL(source_link);
  }
```

1301. Trace first Leo link.

1302. Set the trace source link to a Leo link, if there is one, otherwise clear the Leo source link. Returns the AHM ID (not the obsolete AHFA state ID) of the cause if there was a Leo source link, −1 if there was none, and −2 on some other kind of failure.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_Symbol_ID _marpa_r_first_leo_link_trace(Marpa_Recognizer r)
  {
    ⟨ Return −2 on failure 1229 ⟩
    SRCL source_link;
    YIM item ⟸ r→t_trace_earley_item;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    ⟨ Set item, failing if necessary 1306 ⟩
    source_link ⟸ First_Leo_SRCL_of_YIM(item);
    if (source_link) {
      r→t_trace_source_type ⟸ SOURCE_IS_LEO;
      r→t_trace_source_link ⟸ source_link;
      return Cause_AHMID_of_SRCL(source_link);
    }
    trace_source_link_clear(r);
    return −1;
  }
```

1303. Trace next Leo link.

1304. Set the trace source link to the next Leo link, if there is one. Otherwise clear the trace source link.

1305. Returns the AHM ID if there is a next Leo source link, −1 if there was none, and −2 on some other kind of failure.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_Symbol_ID _marpa_r_next_leo_link_trace(Marpa_Recognizer r){ ⟨ Return −2
              on failure 1229 ⟩
        SRCL source_link;
        YIM item;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    ⟨ Set item, failing if necessary 1306 ⟩
    if (r→t_trace_source_type ≠ SOURCE_IS_LEO) {
      trace_source_link_clear(r);
      MARPA_ERROR(MARPA_ERR_NOT_TRACING_LEO_LINKS);
      return failure_indicator;
    }
    source_link ⟸ Next_SRCL_of_SRCL(r→t_trace_source_link);
    if (¬source_link) {
```

```
    trace_source_link_clear(r);
    return −1;
  }
  r→t_trace_source_link ⇐ source_link;
  return Cause_AHMID_of_SRCL(source_link); }
```

1306. ⟨ Set item, failing if necessary 1306 ⟩ ≡

```
item ⇐ r→t_trace_earley_item;
if (¬item) {
  trace_source_link_clear(r);
  MARPA_ERROR(MARPA_ERR_NO_TRACE_YIM);
  return failure_indicator;
}
```

This code is used in sections 1292, 1295, 1297, 1300, 1302, and 1305.

1307. Clear trace source link.

⟨ Function definitions 41 ⟩ +≡

```
PRIVATE void trace_source_link_clear(RECCE r)
{
  r→t_trace_source_link ⇐ Λ;
  r→t_trace_source_type ⇐ NO_SOURCE;
}
```

1308. Return the predecessor AHM ID. Returns the predecessor AHM ID, or -1 if there is no predecessor. If the recognizer is not trace-safe, if there is no trace source link, if the trace source link is a Leo source, or if there is some other failure, −2 is returned.

⟨ Function definitions 41 ⟩ +≡

```
AHMID _marpa_r_source_predecessor_state(Marpa_Recognizer r){ ⟨ Return −2 on
          failure 1229 ⟩
      unsigned int source_type;
      SRCL source_link;
  ⟨ Unpack recognizer objects 560 ⟩
  ⟨ Fail if not trace-safe 1248 ⟩
  source_type ⇐ r→t_trace_source_type; ⟨ Set source link, failing if
      necessary 1314 ⟩
  switch (source_type) {
  case SOURCE_IS_TOKEN: case SOURCE_IS_COMPLETION:
    {
      YIM predecessor ⇐ Predecessor_of_SRCL(source_link);
      if (¬predecessor) return −1;
      return AHMID_of_YIM(predecessor);
    }
  }
  MARPA_ERROR(invalid_source_type_code(source_type));
  return failure_indicator; }
```

1309. Return the token. Returns the token. The symbol id is the return value, and the value is written to `*value_p`, if it is non-null. If the recognizer is not trace-safe, there is no trace source link, if the trace source link is not a token source, or there is some other failure, −2 is returned.

There is no function to return just the token value for two reasons. First, since token value can be anything an additional return value is needed to indicate errors, which means the symbol ID comes at essentially zero cost. Second, whenever the token value is wanted, the symbol ID is almost always wanted as well.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_Symbol_ID _marpa_r_source_token(Marpa_Recognizer r, int *value_p)
  {
    ⟨ Return −2 on failure 1229 ⟩
    unsigned int source_type;
    SRCL source_link;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    source_type ⇐ r→t_trace_source_type;
    ⟨ Set source link, failing if necessary 1314 ⟩
    if (source_type ≡ SOURCE_IS_TOKEN) {
      if (value_p) *value_p ⇐ Value_of_SRCL(source_link);
      return NSYID_of_SRCL(source_link);
    }
    MARPA_ERROR(invalid_source_type_code(source_type));
    return failure_indicator;
  }
```

1310. Return the Leo transition symbol. The Leo transition symbol is defined only for sources with a Leo predecessor. The transition from a predecessor to the Earley item containing a source will always be over exactly one symbol. In the case of a Leo source, this symbol will be the Leo transition symbol.

1311. Returns the symbol ID of the Leo transition symbol. If the recognizer is not trace-safe, if there is no trace source link, if the trace source link is not a Leo source, or there is some other failure, −2 is returned.

```
⟨ Function definitions 41 ⟩ +≡
  Marpa_Symbol_ID _marpa_r_source_leo_transition_symbol(Marpa_Recognizer r){
            ⟨ Return −2 on failure 1229 ⟩
        unsigned int source_type;
        SRCL source_link;
    ⟨ Unpack recognizer objects 560 ⟩
    ⟨ Fail if not trace-safe 1248 ⟩
    source_type ⇐ r→t_trace_source_type; ⟨ Set source link, failing if
        necessary 1314 ⟩
    switch (source_type) {
```

```
case SOURCE_IS_LEO: return Leo_Transition_NSYID_of_SRCL(source_link);
}
MARPA_ERROR(invalid_source_type_code(source_type));
return failure_indicator; }
```

1312. Return the middle Earley set ordinal. Every source has the following defined:

- An origin (or start ordinal).
- An end ordinal (the current set).
- A "middle ordinal". An Earley item can be thought of as covering a "span" from its origin to the current set. For each source, this span is divided into two pieces at the middle ordinal.

1313. Informally, the middle ordinal can be thought of as dividing the span between the predecessor and either the source's cause or its token. If the source has no predecessor, the middle ordinal is the same as the origin. If there is a predecessor, the middle ordinal is the current set of the predecessor. If there is a cause, the middle ordinal is always the same as the origin of the cause. If there is a token, the middle ordinal is always where the token starts. On failure, such as there being no source link, −2 is returned.

```
⟨Function definitions 41⟩ +≡
  Marpa_Earley_Set_ID _marpa_r_source_middle(Marpa_Recognizer r){ ⟨Return −2 on
        failure 1229⟩
      YIM predecessor_yim ⇐ Λ;
      unsigned int source_type;
      SRCL source_link;
    ⟨Unpack recognizer objects 560⟩
    ⟨Fail if not trace-safe 1248⟩
    source_type ⇐ r→t_trace_source_type; ⟨Set source link, failing if
        necessary 1314⟩
    switch (source_type) {
    case SOURCE_IS_LEO:
      {
        LIM predecessor ⇐ LIM_of_SRCL(source_link);
        if (predecessor)
          predecessor_yim ⇐ Trailhead_YIM_of_LIM(predecessor);
        break;
      }
    case SOURCE_IS_TOKEN: case SOURCE_IS_COMPLETION:
      {
        predecessor_yim ⇐ Predecessor_of_SRCL(source_link);
        break;
      }
    default: MARPA_ERROR(invalid_source_type_code(source_type));
      return failure_indicator;
```

```
    }
    if (predecessor_yim) return YS_Ord_of_YIM(predecessor_yim);
    return Origin_Ord_of_YIM(r→t_trace_earley_item); }
```

1314. ⟨ Set source link, failing if necessary 1314 ⟩ ≡

```
  source_link ⇐ r→t_trace_source_link;
  if (¬source_link) {
    MARPA_ERROR(MARPA_ERR_NO_TRACE_SRCL);
    return failure_indicator;
  }
```

This code is used in sections 1308, 1309, 1311, and 1313.

1315. Or-node trace functions.

1316. This is common logic in the or-node trace functions. In the case of a nulling bocage, the or count of the bocage is zero, so that any `or_node_id` is either a soft or a hard error, depending on whether it is non-negative or negative.

⟨ Check `or_node_id` 1316 ⟩ ≡

```
  {
    if (_MARPA_UNLIKELY(or_node_id ≥ OR_Count_of_B(b))) {
      return −1;
    }
    if (_MARPA_UNLIKELY(or_node_id < 0)) {
      MARPA_ERROR(MARPA_ERR_ORID_NEGATIVE);
      return failure_indicator;
    }
  }
```

This code is used in sections 1008, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1329, and 1330.

1317. ⟨ Set `or_node` or fail 1317 ⟩ ≡

```
  {
    if (_MARPA_UNLIKELY(¬ORs_of_B(b))) {
      MARPA_ERROR(MARPA_ERR_NO_OR_NODES);
      return failure_indicator;
    }
    or_node ⇐ OR_of_B_by_ID(b, or_node_id);
  }
```

This code is used in sections 1008, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1329, and 1330.

1318. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_b_or_node_set(Marpa_Bocage b, Marpa_Or_Node_ID or_node_id)
  {
    OR or_node;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack bocage objects 939 ⟩
```

```
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Check or_node_id 1316 ⟩
    ⟨ Set or_node or fail 1317 ⟩
    return YS_Ord_of_OR(or_node);
  }
```

1319. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_b_or_node_origin(Marpa_Bocage b, Marpa_Or_Node_ID or_node_id)
  {
    OR or_node;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack bocage objects 939 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Check or_node_id 1316 ⟩
    ⟨ Set or_node or fail 1317 ⟩
    return Origin_Ord_of_OR(or_node);
  }
```

1320. ⟨ Function definitions 41 ⟩ +≡

```
  Marpa_IRL_ID _marpa_b_or_node_irl(Marpa_Bocage b, Marpa_Or_Node_ID
          or_node_id)
  {
    OR or_node;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack bocage objects 939 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Check or_node_id 1316 ⟩
    ⟨ Set or_node or fail 1317 ⟩
    return IRLID_of_OR(or_node);
  }
```

1321. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_b_or_node_position(Marpa_Bocage b, Marpa_Or_Node_ID or_node_id)
  {
    OR or_node;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack bocage objects 939 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Check or_node_id 1316 ⟩
    ⟨ Set or_node or fail 1317 ⟩
    return Position_of_OR(or_node);
  }
```

1322. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_b_or_node_is_whole(Marpa_Bocage b, Marpa_Or_Node_ID or_node_id)
{
  OR or_node;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Check or_node_id 1316 ⟩
  ⟨ Set or_node or fail 1317 ⟩
  return Position_of_OR(or_node) ≥ Length_of_IRL(IRL_of_OR(or_node)) ? 1 : 0;
}
```

1323. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_b_or_node_is_semantic(Marpa_Bocage b, Marpa_Or_Node_ID or_node_id)
{
  OR or_node;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Check or_node_id 1316 ⟩
  ⟨ Set or_node or fail 1317 ⟩
  return ¬IRL_has_Virtual_LHS(IRL_of_OR(or_node));
}
```

1324. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_b_or_node_first_and(Marpa_Bocage b, Marpa_Or_Node_ID or_node_id)
{
  OR or_node;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Check or_node_id 1316 ⟩
  ⟨ Set or_node or fail 1317 ⟩
  return First_ANDID_of_OR(or_node);
}
```

1325. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_b_or_node_last_and(Marpa_Bocage b, Marpa_Or_Node_ID or_node_id)
{
  OR or_node;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Check or_node_id 1316 ⟩
```

```
    ⟨ Set or_node or fail 1317 ⟩
    return First_ANDID_of_OR(or_node) + AND_Count_of_OR(or_node) − 1;
  }
```

1326. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_b_or_node_and_count(Marpa_Bocage b, Marpa_Or_Node_ID or_node_id)
  {
    OR or_node;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack bocage objects 939 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Check or_node_id 1316 ⟩
    ⟨ Set or_node or fail 1317 ⟩
    return AND_Count_of_OR(or_node);
  }
```

1327. Ordering trace functions.

1328. This is common logic in the ordering trace functions. In the case of a nulling ordering, the or count of the ordering is zero, so that any `or_node_id` is either a soft or a hard error, depending on whether it is non-negative or negative.

1329. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_o_or_node_and_node_count(Marpa_Order o, Marpa_Or_Node_ID
          or_node_id)
  {
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack order objects 984 ⟩
    ⟨ Fail if fatal error 1249 ⟩
    ⟨ Check or_node_id 1316 ⟩
    if (¬O_is_Default(o)) {
      ANDID **const and_node_orderings ⟸ o→t_and_node_orderings;
      ANDID *ordering ⟸ and_node_orderings[or_node_id];
      if (ordering) return ordering[0];
    }
    {
      OR or_node;
      ⟨ Set or_node or fail 1317 ⟩
      return AND_Count_of_OR(or_node);
    }
  }
```

1330. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_o_or_node_and_node_id_by_ix(Marpa_Order o, Marpa_Or_Node_ID
        or_node_id, int ix)
{
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack order objects 984 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  ⟨ Check or_node_id 1316 ⟩
  if (¬O_is_Default(o)) {
    ANDID **const and_node_orderings ⟸ o→t_and_node_orderings;
    ANDID *ordering ⟸ and_node_orderings[or_node_id];
    if (ordering) return ordering[1 + ix];
  }
  {
    OR or_node;
    ⟨ Set or_node or fail 1317 ⟩
    return First_ANDID_of_OR(or_node) + ix;
  }
}
```

1331. And-node trace functions.

1332. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_b_and_node_count(Marpa_Bocage b)
{
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Fail if fatal error 1249 ⟩
  return AND_Count_of_B(b);
}
```

1333. ⟨ Check bocage and_node_id; set and_node 1333 ⟩ ≡

```
{
  if (and_node_id ≥ AND_Count_of_B(b)) {
    return −1;
  }
  if (and_node_id < 0) {
    MARPA_ERROR(MARPA_ERR_ANDID_NEGATIVE);
    return failure_indicator;
  }
  {
    AND and_nodes ⟸ ANDs_of_B(b);
    if (¬and_nodes) {
      MARPA_ERROR(MARPA_ERR_NO_AND_NODES);
      return failure_indicator;
```

```
      }
      and_node ⟸ and_nodes + and_node_id;
    }
  }
```

This code is used in sections 1334, 1335, 1336, 1337, 1338, and 1339.

1334. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_b_and_node_parent(Marpa_Bocage b, Marpa_And_Node_ID and_node_id)
  {
    AND and_node;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack bocage objects 939 ⟩
    ⟨ Check bocage and_node_id; set and_node 1333 ⟩
    return ID_of_OR(OR_of_AND(and_node));
  }
```

1335. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_b_and_node_predecessor(Marpa_Bocage b, Marpa_And_Node_ID
          and_node_id)
  {
    AND and_node;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack bocage objects 939 ⟩
    ⟨ Check bocage and_node_id; set and_node 1333 ⟩
    {
      const OR predecessor_or ⟸ Predecessor_OR_of_AND(and_node);
      const ORID predecessor_or_id ⟸ predecessor_or ?
          ID_of_OR(predecessor_or) : −1;
      return predecessor_or_id;
    }
  }
```

1336. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_b_and_node_cause(Marpa_Bocage b, Marpa_And_Node_ID and_node_id)
  {
    AND and_node;
    ⟨ Return −2 on failure 1229 ⟩
    ⟨ Unpack bocage objects 939 ⟩
    ⟨ Check bocage and_node_id; set and_node 1333 ⟩
    {
      const OR cause_or ⟸ Cause_OR_of_AND(and_node);
      const ORID cause_or_id ⟸ OR_is_Token(cause_or) ? −1 : ID_of_OR(cause_or);
      return cause_or_id;
    }
  }
```

1337. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_b_and_node_symbol(Marpa_Bocage b, Marpa_And_Node_ID and_node_id)
{
  AND and_node;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Check bocage and_node_id; set and_node 1333 ⟩
  {
    const OR cause_or ⟸ Cause_OR_of_AND(and_node);
    const XSYID symbol_id ⟸ OR_is_Token(cause_or) ? NSYID_of_OR(cause_or) :
        −1;
    return symbol_id;
  }
}
```

1338. ⟨ Function definitions 41 ⟩ +≡

```
Marpa_Symbol_ID _marpa_b_and_node_token(Marpa_Bocage b, Marpa_And_Node_ID
        and_node_id, int *value_p)
{
  AND and_node;
  OR cause_or;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Check bocage and_node_id; set and_node 1333 ⟩
  cause_or ⟸ Cause_OR_of_AND(and_node);
  if (¬OR_is_Token(cause_or)) return −1;
  if (value_p) *value_p ⟸ Value_of_OR(cause_or);
  return NSYID_of_OR(cause_or);
}
```

1339. The "middle" earley set of the and-node. It is most simply defined as equivalent to the start of the cause, but the cause can be token, and in that case the simpler definition is not helpful. Instead, the end of the predecessor is used, if there is one. If there is no predecessor, the origin of the parent or-node will always be the same as "middle" of the or-node.

⟨ Function definitions 41 ⟩ +≡

```
Marpa_Earley_Set_ID _marpa_b_and_node_middle(Marpa_Bocage b, Marpa_And_Node_ID
        and_node_id)
{
  AND and_node;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack bocage objects 939 ⟩
  ⟨ Check bocage and_node_id; set and_node 1333 ⟩
  {
```

```
    const OR predecessor_or ⟸ Predecessor_OR_of_AND(and_node);
    if (predecessor_or) {
      return YS_Ord_of_OR(predecessor_or);
    }
  }
  return Origin_Ord_of_OR(OR_of_AND(and_node));
}
```

1340. Nook trace functions.

1341. This is common logic in the *NOOK* trace functions.

⟨ Check *r* and `nook_id`; set `nook` 1341 ⟩ ≡

```
{
  NOOK base_nook;
  ⟨ Fail if fatal error 1249 ⟩
  if (T_is_Exhausted(t)) {
    MARPA_ERROR(MARPA_ERR_BOCAGE_ITERATION_EXHAUSTED);
    return failure_indicator;
  }
  if (nook_id < 0) {
    MARPA_ERROR(MARPA_ERR_NOOKID_NEGATIVE);
    return failure_indicator;
  }
  if (nook_id ≥ Size_of_T(t)) {
    return −1;
  }
  base_nook ⟸ MARPA_DSTACK_BASE(t→t_nook_stack, NOOK_Object);
  nook ⟸ base_nook + nook_id;
}
```

This code is used in sections 1342, 1343, 1344, 1345, 1346, 1347, and 1348.

1342. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_t_nook_or_node(Marpa_Tree t, int nook_id)
{
  NOOK nook;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack tree objects 1023 ⟩
  ⟨ Check r and nook_id; set nook 1341 ⟩
  return ID_of_OR(OR_of_NOOK(nook));
}
```

1343. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_t_nook_choice(Marpa_Tree t, int nook_id)
{
  NOOK nook;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack tree objects 1023 ⟩
  ⟨ Check r and nook_id; set nook 1341 ⟩
  return Choice_of_NOOK(nook);
}
```

1344. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_t_nook_parent(Marpa_Tree t, int nook_id)
{
  NOOK nook;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack tree objects 1023 ⟩
  ⟨ Check r and nook_id; set nook 1341 ⟩
  return Parent_of_NOOK(nook);
}
```

1345. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_t_nook_cause_is_ready(Marpa_Tree t, int nook_id)
{
  NOOK nook;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack tree objects 1023 ⟩
  ⟨ Check r and nook_id; set nook 1341 ⟩
  return NOOK_Cause_is_Expanded(nook);
}
```

1346. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_t_nook_predecessor_is_ready(Marpa_Tree t, int nook_id)
{
  NOOK nook;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack tree objects 1023 ⟩
  ⟨ Check r and nook_id; set nook 1341 ⟩
  return NOOK_Predecessor_is_Expanded(nook);
}
```

1347. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_t_nook_is_cause(Marpa_Tree t, int nook_id)
{
  NOOK nook;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack tree objects 1023 ⟩
  ⟨ Check r and nook_id; set nook 1341 ⟩
  return NOOK_is_Cause(nook);
}
```

1348. ⟨ Function definitions 41 ⟩ +≡

```
int _marpa_t_nook_is_predecessor(Marpa_Tree t, int nook_id)
{
  NOOK nook;
  ⟨ Return −2 on failure 1229 ⟩
  ⟨ Unpack tree objects 1023 ⟩
  ⟨ Check r and nook_id; set nook 1341 ⟩
  return NOOK_is_Predecessor(nook);
}
```

1349. Looker functions.
The functions are intended as a run-time and production-quality way of examining the Earley tables. For the recognizer data, in Kollos, they will replace the "trace" functions.
Lookers are internal. Many Libmarpa internal calls currently do some checking of arguments. Libmarpa methods, including at least one of the looker methods, will do checking for the user. Callers of looker methods are required to ensure all necessary argument checking is done.
All looker function calls are mutators. In addition, the lookers have public accessor macros. Looker data can be safely accessed only via a looker accessor or the return value of a looker mutator. After any call to a looker function, only a specified set of accessors are valid. This is because the lookers mutators reuse data fields.

⟨ Public structures 44 ⟩ +≡
```
struct s_marpa_yim_look {
    Marpa_Rule_ID t_yim_look_rule_id;
    int t_yim_look_dot;
    Marpa_Earley_Set_ID t_yim_look_origin_id;
    Marpa_IRL_ID t_yim_look_irl_id;
    int t_yim_look_irl_dot;
};
typedef struct s_marpa_yim_look Marpa_Earley_Item_Look;
```

1350. These accessors are valid for `marpa_r_look_yim`.

⟨ Public defines 109 ⟩ +≡
```
#define marpa_eim_look_rule_id(l) ((l)→t_yim_look_rule_id)
#define marpa_eim_look_dot(l) ((l)→t_yim_look_dot)
#define marpa_eim_look_origin(l) ((l)→t_yim_look_origin_id)
#define marpa_eim_look_irl_id(l) ((l)→t_yim_look_irl_id)
#define marpa_eim_look_irl_dot(l) ((l)→t_yim_look_irl_dot)
```

1351. The YIM looker returns data specific to a YIM. It is also necessary before the use of any other looker accessor or mutator, to initializes the looker's Earley set and Earley item.

⟨ Function definitions 41 ⟩ +≡
```
PRIVATE int look_yim(Marpa_Earley_Item_Look *look, YS earley_set,
        Marpa_Earley_Item_ID eim_id)
{
    int xrl_position;
    int raw_xrl_position;
    YIM *earley_items ⇐ YIMs_of_YS(earley_set);
    YIM earley_item ⇐ earley_items[eim_id];
    AHM ahm ⇐ AHM_of_YIM(earley_item);
    XRL xrl ⇐ XRL_of_AHM(ahm);
    if (xrl) {
      marpa_eim_look_rule_id(look) ⇐ ID_of_XRL(xrl);
```

```
      xrl_position ⟸ XRL_Position_of_AHM(ahm);
      raw_xrl_position ⟸ Raw_XRL_Position_of_AHM(ahm);
    }
    else {
      marpa_eim_look_rule_id(look) ⟸ −1;
      raw_xrl_position ⟸ xrl_position ⟸ −1;
    }
    marpa_eim_look_dot(look) ⟸ xrl_position;
    marpa_eim_look_origin(look) ⟸ Origin_Ord_of_YIM(earley_item);
    marpa_eim_look_irl_id(look) ⟸ IRLID_of_AHM(ahm);
    marpa_eim_look_irl_dot(look) ⟸ Position_of_AHM(ahm);
    return raw_xrl_position;
  }
```

1352. This is the external wrapper of the YIM looker. Caller must ensure that its arguments are checked.

⟨ Public function prototypes 411 ⟩ +≡

```
  int _marpa_r_look_yim(Marpa_Recognizer r, Marpa_Earley_Item_Look
      *look, Marpa_Earley_Set_ID es_id, Marpa_Earley_Item_ID eim_id);
```

1353. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_r_look_yim(Marpa_Recognizer r, Marpa_Earley_Item_Look
        *look, Marpa_Earley_Set_ID es_id, Marpa_Earley_Item_ID eim_id)
  {
    const YS earley_set ⟸ YS_of_R_by_Ord(r, es_id);
    return look_yim(look, earley_set, eim_id);
  }
```

1354. This function is convenient for checking looker arguments. Returns 1 if all are OK, 0 if no such Earley item, -1 if no such Earley set. If Earley item or Earley set are malformed, or on other hard failure, returns -2.

⟨ Public function prototypes 411 ⟩ +≡

```
  int _marpa_r_yim_check(Marpa_Recognizer r, Marpa_Earley_Set_ID es_id,
      Marpa_Earley_Item_ID eim_id);
```

1355. ⟨ Function definitions 41 ⟩ +≡

```
  int _marpa_r_yim_check(Marpa_Recognizer r, Marpa_Earley_Set_ID es_id,
        Marpa_Earley_Item_ID eim_id){ YS earley_set;
      ⟨ Unpack recognizer objects 560 ⟩
      ⟨ Return −2 on failure 1229 ⟩
      if (es_id < 0) {
        MARPA_ERROR(MARPA_ERR_INVALID_LOCATION);
        return failure_indicator;
      }
      if (eim_id < 0) {
```

```
    MARPA_ERROR(MARPA_ERR_YIM_ID_INVALID);
    return failure_indicator;
  }
  r_update_earley_sets(r);
  earley_set ⟸ YS_of_R_by_Ord(r, es_id);
  if (es_id ≥ MARPA_DSTACK_LENGTH(r→t_earley_set_stack)) {
    MARPA_ERROR(MARPA_ERR_INVALID_LOCATION);
    return −1;
  }
  if (eim_id ≥ YIM_Count_of_YS(earley_set)) {
    return 0;
  }
  return 1; }
```

1356. Basic PIM Looker functions.

1357. The only PIM looker functions at the moment are “basic”. They return data only for PIMs chains which do not contain a LIM. For efficiency, they use the fact that the LIMs come first in a PIM chain.

1358. The structure for looking at PIM data. Eventually there will be a lot of fields for LIM data. `t_pim_eim_id` is −1 if PIM is a LIM, otherwise it is the ordinal of the EIM.

⟨ Public structures 44 ⟩ +≡

```
struct s_marpa_pim_look {
  _Marpa_PIM t_pim_look_current;
  Marpa_Earley_Item_ID t_pim_look_eim_id;
};
typedef struct s_marpa_pim_look Marpa_Postdot_Item_Look;
```

1359. These accessors are valid for `marpa_r_look_pim_eim_first` and `marpa_r_look_pim_eim_next`.

⟨ Public defines 109 ⟩ +≡

```
#define marpa_pim_look_eim(l) ((l)→t_pim_look_eim_id)
```

1360. Return the first Earley Item ID from a PIM chain. Caller must ensure that its arguments are checked.

On success, returns the Earley item index, and sets up the field in the `look` structure. If there is no PIM chain for `es_id` and `nsy_id`, returns -1. If this PIM chain contains a LIM, returns -1.

⟨ Public function prototypes 411 ⟩ +≡

```
int _marpa_r_look_pim_eim_first(Marpa_Recognizer r, Marpa_Postdot_Item_Look
    *look, Marpa_Earley_Set_ID es_id, Marpa_Symbol_ID nsy_id);
```

1361. This function is prototyped here rather than the internal.texi file.

⟨ Function definitions 41 ⟩ +≡

```
int _marpa_r_look_pim_eim_first(Marpa_Recognizer r, Marpa_Postdot_Item_Look
        *look, Marpa_Earley_Set_ID es_id, Marpa_Symbol_ID nsy_id)
{
  int earley_item_ix ⟸ −1;
  const YS earley_set ⟸ YS_of_R_by_Ord(r, es_id);
  YIM earley_item ⟸ Λ;
  PIM pim ⟸ First_PIM_of_YS_by_NSYID(earley_set, nsy_id);
  if (¬pim) return −1;
  earley_item ⟸ YIM_of_PIM(pim);
  if (¬earley_item) return −1;
  look→t_pim_look_current ⟸ pim;
  earley_item_ix ⟸ Ord_of_YIM(earley_item);
  marpa_pim_look_eim(look) ⟸ earley_item_ix;
  return earley_item_ix;
}
```

1362. Return the data for the next PIM from a PIM chain. Caller must ensure that its arguments are checked. `look` must have been initialized by a previous call to `_marpa_r_look_pim_eim_first`.

On success, returns the Earley item index, and sets up the field in the `look` structure. If there is no next PIM, returns -1. `_marpa_r_look_pim_eim_first` should soft fail if there is a LIM in this PIM chain but, just in case, `_marpa_r_look_pim_eim_next` soft fails and returns -1 if this PIM chain contains a LIM.

⟨ Public function prototypes 411 ⟩ +≡

```
int _marpa_r_look_pim_eim_next(Marpa_Postdot_Item_Look *look);
```

1363. This function is prototyped here rather than the internal.texi file.

⟨ Function definitions 41 ⟩ +≡

```
int _marpa_r_look_pim_eim_next(Marpa_Postdot_Item_Look *look)
{
  int earley_item_ix ⟸ −1;
  YIM earley_item ⟸ Λ;
  PIM pim ⟸ Next_PIM_of_PIM(look→t_pim_look_current);
  if (¬pim) return −1;
  earley_item ⟸ YIM_of_PIM(pim);
  if (¬earley_item) return −1;
  look→t_pim_look_current ⟸ pim;
  earley_item_ix ⟸ Ord_of_YIM(earley_item);
  marpa_pim_look_eim(look) ⟸ earley_item_ix;
  return earley_item_ix;
}
```

1364. Debugging functions. Much of the debugging logic is in other documents. Here is the public interface, which allows resetting the debug handler and the debug level, as well as functions which are targeted at debugging the data structures describes in this document.

⟨ Debugging variable declarations 1258 ⟩ +≡

```
extern int marpa__default_debug_handler(const char *format, ...);
extern int(*marpa__debug_handler)(const char *, ...);
extern int marpa__debug_level;
```

1365. ⟨ Function definitions 41 ⟩ +≡

```
void marpa_debug_handler_set(int(*debug_handler)(const char *, ...))
{
  marpa__debug_handler ⟸ debug_handler;
}
```

1366. ⟨ Function definitions 41 ⟩ +≡

```
int marpa_debug_level_set(int new_level)
{
  const int old_level ⟸ marpa__debug_level;
  marpa__debug_level ⟸ new_level;
  return old_level;
}
```

1367. For thread-safety, these are for debugging only. Even in debugging, while not actually initialized constants, they are intended to be set very early and left unchanged.

1368. ⟨ Global debugging variables 1368 ⟩ ≡

```
int(*marpa__debug_handler)(const char *, ...) ⟸ marpa__default_debug_handler;
int marpa__debug_level ⟸ 0;
```

This code is used in section 1384.

1369. Earley item tag. A function to print a descriptive tag for an Earley item.

⟨ Debug function prototypes 1369 ⟩ ≡

```
static const char *yim_tag_safe(char *buffer, GRAMMAR g, YIM yim) UNUSED;
static const char *yim_tag(GRAMMAR g, YIM yim) UNUSED;
```

See also sections 1371, 1373, and 1375.

This code is used in section 1384.

1370. It is passed a buffer to keep it thread-safe.

⟨ Debug function definitions 1370 ⟩ ≡

```
static const char *yim_tag_safe(char *buffer, GRAMMAR g, YIM yim)
{
  if (¬yim) return "NULL";
  sprintf(buffer, "S%d@%d-%d", AHMID_of_YIM(yim), Origin_Earleme_of_YIM(yim),
      Earleme_of_YIM(yim));
```

```
    return buffer;
  }
  static char DEBUG_yim_tag_buffer[1000];
  static const char *yim_tag(GRAMMAR g, YIM yim)
  {
    return yim_tag_safe(DEBUG_yim_tag_buffer, g, yim);
  }
```

See also sections 1372, 1374, and 1376.

This code is used in section 1384.

1371. Leo item tag. A function to print a descriptive tag for an Leo item.

⟨ Debug function prototypes 1369 ⟩ +≡

```
  static char *lim_tag_safe(char *buffer, LIM lim) UNUSED;
  static char *lim_tag(LIM lim) UNUSED;
```

1372. This function is passed a buffer to keep it thread-safe. be made thread-safe.

⟨ Debug function definitions 1370 ⟩ +≡

```
  static char *lim_tag_safe(char *buffer, LIM lim)
  {
    sprintf(buffer, "L%d@%d", Postdot_NSYID_of_LIM(lim), Earleme_of_LIM(lim));
    return buffer;
  }
  static char DEBUG_lim_tag_buffer[1000];
  static char *lim_tag(LIM lim)
  {
    return lim_tag_safe(DEBUG_lim_tag_buffer, lim);
  }
```

1373. Or-node tag. Functions to print a descriptive tag for an or-node item. One is thread-safe, the other is more convenient but not thread-safe.

⟨ Debug function prototypes 1369 ⟩ +≡

```
  static const char *or_tag_safe(char *buffer, OR or) UNUSED;
  static const char *or_tag(OR or) UNUSED;
```

1374. It is passed a buffer to keep it thread-safe.

⟨ Debug function definitions 1370 ⟩ +≡

```
  static const char *or_tag_safe(char *buffer, OR or)
  {
    if (¬or) return "NULL";
    if (OR_is_Token(or)) return "TOKEN";
    if (Type_of_OR(or) ≡ DUMMY_OR_NODE) return "DUMMY";
    sprintf(buffer, "R%d:%d@%d-%d", IRLID_of_OR(or), Position_of_OR(or),
        Origin_Ord_of_OR(or), YS_Ord_of_OR(or));
    return buffer;
```

```
}
static char DEBUG_or_tag_buffer[1000];
static const char *or_tag(OR or)
{
  return or_tag_safe(DEBUG_or_tag_buffer, or);
}
```

1375. AHM tag. Functions to print a descriptive tag for an AHM. One is passed a buffer to keep it thread-safe. The other uses a global buffer, which is not thread-safe, but convenient when debugging in a non-threaded environment.

⟨ Debug function prototypes 1369 ⟩ +≡

```
static const char *ahm_tag_safe(char *buffer, AHM ahm) UNUSED;
static const char *ahm_tag(AHM ahm) UNUSED;
```

1376. ⟨ Debug function definitions 1370 ⟩ +≡

```
static const char *ahm_tag_safe(char *buffer, AHM ahm)
{
  if (¬ahm) return "NULL";
  const int ahm_position ⟸ Position_of_AHM(ahm);
  if (ahm_position ≥ 0) {
    sprintf(buffer, "R%d@%d", IRLID_of_AHM(ahm), Position_of_AHM(ahm));
  }
  else {
    sprintf(buffer, "R%d@end", IRLID_of_AHM(ahm));
  }
  return buffer;
}
static char DEBUG_ahm_tag_buffer[1000];
static const char *ahm_tag(AHM ahm)
{
  return ahm_tag_safe(DEBUG_ahm_tag_buffer, ahm);
}
```

1377. File layout.

1378. The output files are **not** source files, but I add the license to them anyway, as close to the top as possible.

1379. Also, it is helpful to someone first trying to orient herself, if built source files contain a comment to that effect and a warning not that they are not intended to be edited directly. So I add such a comment.

1380. `marpa`.*c* **layout.**

1381. ⟨ `marpa.c.p10` 1381 ⟩ ≡
#include "config.h"
#ifndef MARPA_DEBUG
#define MARPA_DEBUG 0
#endif
#include "marpa.h"
#include "marpa_ami.h"
  ⟨ Preprocessor definitions ⟩
#include "marpa_obs.h"
#include "marpa_avl.h"
  ⟨ Private incomplete structures 107 ⟩
  ⟨ Private typedefs 49 ⟩
  ⟨ Private utility structures 1184 ⟩
  ⟨ Private structures 48 ⟩
  ⟨ Private unions 669 ⟩

See also sections 1382 and 1383.

1382. To preserve thread-safety, global variables are either constants, or used strictly for debugging.

⟨ `marpa.c.p10` 1381 ⟩ +≡
  ⟨ Global constant variables 40 ⟩

1383. ⟨ `marpa.c.p10` 1381 ⟩ +≡
  ⟨ Recognizer structure 550 ⟩
  ⟨ Source object structure 680 ⟩
  ⟨ Earley item structure 651 ⟩
  ⟨ Bocage structure 937 ⟩

1384. ⟨ `marpa.c.p50` 1384 ⟩ ≡
  ⟨ Debugging variable declarations 1258 ⟩
#if MARPA_DEBUG
  ⟨ Debug function prototypes 1369 ⟩
  ⟨ Debug function definitions 1370 ⟩
#endif
  ⟨ Global debugging variables 1368 ⟩
  ⟨ Function definitions 41 ⟩

1385. Public header file.

1386. Our portion of the public header file.

1387. ⟨ `marpa.h.p50` 1387 ⟩ ≡
extern const int `marpa_major_version`;
extern const int `marpa_minor_version`;
extern const int `marpa_micro_version`;
⟨ Public defines 109 ⟩
⟨ Public incomplete structures 47 ⟩
⟨ Public typedefs 91 ⟩
⟨ Public structures 44 ⟩
⟨ Debugging variable declarations 1258 ⟩
⟨ Public function prototypes 411 ⟩

1388. Index.

Marpa: the program

June 15, 2022 at 15:21