1. License.

Gened date: January 2, 2015

Copyright © 1998-2015 Dave Bone

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

2. Summary of Yacco2 and Linker's Symbol Table. Basic hash table for Yacco2's symbols. It is also used by Yacco2's linker companion who deals in the terminal first sets of threads.

Yacco2's ratatouille:

- \yacco2\compiler\stbl - NT symbol table directory
- *yacco2stbl.w* - cweb generator file
- *hash_table_size* - current space allocator
- *yacco2_stbl.h* - header file
- *yacco2_stbl.cpp* - implementation
- *yacco2_stbl* - symbol table namespace
- *yacco2_stbl::stbl* - global symbol table variable
- *T_sym_report_card* - defined in Yacco2's terminal alphabet

Dependency files from other Yacco2 sub-systems:

- *yacco2.h* - basic definitions used by Yacco2
- *yacco2_err_symbols.h* - error definitions from Yacco2's grammar alphabet
- *yacco2_terminals.h* - regular terminal definitions from Yacco2's grammar alphabet

Symbol table procedures:

- *yacco2::add_sym_to_stbl*
- *yacco2::find_sym_in_stbl*
- *yacco2::get_sym_entry_by_sub*
- *yacco2::test_program* — see 'how to run the test' section

Note: *yacco2_stbl.cpp* gets compiled with Yacco2 parse library's grammars and placed into the runtime library.

3. Synopsis of table entries.

Is it a Forest or a tree?

As u the reader are probably IT types and so trees can have a different meaning, the question can be rephrased: is it a group or an individual? The symbol table takes a group approach to symbols. What does this mean? Each specific symbol (tree) is classified as either a rule, terminal, keyword, or a thread thus pigeon holeing the entry into a forest. This generalization makes parsing the grammar easier. See *rule_in_stbl* and *T_in_stbl* defined in the terminal grammar as classified examples.

The weakness in this 1 to 1 relationship is in error processing where the point of reference is not where the point of entry into the symbol table occured. To correct this weakness, the symbol table's entry has 2 parts: the symbol that created the definition and a cross reference list as referenced within the subrule's right-hand-side string of symbols. Each symbol in this xref list provides the source file coordinates for error processing purposes. The referenced symbol is entered in LILO order.

4. Special note.

Scratch pad coordinates of *rule_in_stbl* and *T_in_stbl*

When the symbol table is accessed, the current token's coordinates overrides the found table entry's original birthings of *rule_in_stbl* or *T_in_stbl*. The other entries *th_in_stbl* or *kw_in_stbl* are similarly serviced.

Using this returned generalization terminal with its scratch pad origins allows the grammar writer to create other tokens like *refered_rule* or *refered_T* containing the current coordinates. There are 2 create contexts for a symbol table entry: at definition time for all terminals and sometimes rules, and referenced time for rules of a subrule definition. All symbols in a subrule are references but i relaxed the requirement that rules must be defined before use as in the formal definition of a grammar. The waiving of this formality allows Productions to forward reference to-be-defined rules. Of course, Yacco2 must post process the declarations of rule use against their definitions to report referenced but not defined rules within the Productions.

5. Introduction to Yacco2's Symbol Table. This is Yacco2's family of symbol table routines. It uses the *Linear probing and insertion* hashing *Algorithm L* from *Vol. 3 Sorting and Searcing* P. 526 of *The Art of Computer Programming* by Donald E. Knuth. As the symbols from Yacco2 are variable in length, the *Hash procedure code* uses a scheme suggested by R. Sedgewick from his book Algorithms 2nd edition p. 233 to modulo per character.

A grammar's symbols are the nonterminals and terminals. The terminals are constant across all grammars being defined as a thread or a stand-alone unit. Usually they are brought into the grammar by Yacco2's "@" include file operator. Nonterminals (rules) are local to the grammar being defined allowing for same name usuage across multiple grammars. The only globalness to a grammar's name is its namespace and possibly its thread name. All grammar symbols are mutually exclusive to one another: there is no commonness across these enities. From a grammar's perspective, *Terminals* are classified into 4 categories: errors, raw characters, lr constant symbols, and finally the regular terminals. It is their literal string that one uses throughout the grammar's productions which makes it global in scope. From a syntax-directed code perspective, these categories are emitted using their namespace qualifiers. This decision was made to lower global namespace pollution. Hence the syntax-directed c++ code must use their namespace qualifier. From a symbol table management point of view, symbol handling is very easy as a symbol scoping stack is not required as in the Pascal language.

Using *cweb* allows me to converse with myself on various considerations which hopefully is not too verbous or symptoms of paranoia. An array of symbol entry records is used. Yes I know what is written about the merits of object oriented programming but... As a package all-inclusive-deal the class is safe — no need to remember cleanup duties like routines that are loosely coupled. But do I want to include all the other mechanisms required like the operator[] or use a type from the Standard Template Library (STL) like Vector? No I don't but I want a containment facility of all its loosely declared elements. So how to do it? Just wrap it in a namespace construct. Now the safe way is to not use a pointer mechanism requiring a destructor for cleanup but an array of symbol entry records which does not grow out the memory dynamically but takes the maximum size at startup time. No memory leakage can occur and no cleanup required. This requires a change to the symbol entry to indicate vacancy which had used the null pointer. The cost of table initialization should be done at the compiler/linker/startup time. Each occupied entry in the table records how the symbol has been used and declared with a pointer to the symbol's definition.

As this is the oracle of Yacco2's symbol table management, I want it to be safe and fast. It is the safe part that becomes intriguing. There are many ways to handle errors:

1) abort
2) throw/catch an error using c++ constructs
3) return some form of status

Abort is a drastic last-resort measure. Flow control is shutdown with a message directed to the user. Throw/catch has merit but I'm not wanting to get into it for efficiency purposes. The degree of error processing for this family of routines is single level calls: not recursive. Typically point 3 is a singular entity: an integer. When things evolve, this strategy is too simplistic when more points of evidence need reporting. So, lets look at a report card approach which must be safe and efficient. When I went to school and teachers could write or would take the time to properly evaluate the student through comments, the take home bulletin had to be signed by the parent or a proxy... Now the report card allows multiple statuses to be marked up and returned to the caller. It becomes the caller's responsibility to check for irregularities and take appropriate action: not like the old days when an unsigned report card meant go back home until a signature acknowledges receipt.

So how can efficiency be put into place? A globally defined variable accessed by the caller/called routines has no discipline. The caller and the called do their thing with the implicit understanding that each routine deposits and checks the global variable. It is this implicit coupling that makes it highly efficient in a one-to-one action/response narration. I do not like implicitness as it forces one to build up all these untold truths, constraints, pre-post conditions. By importing all components via parameters to the called routine removes misunderstanding or forgetfullness.

Now how to make the report card approach efficient? Returning a result by use of a function call is the usual way of doing things. Established between the caller / called routines is the explicit relationship that the

result is returned. Return-by-value depends on the optimization of the compiler regarding efficiency on "the copy of objects" syndrome. Return-by-address thrashs the memory heap. Both methods leaves one open to not deal with the returned result. In fact the function can be called with the returned parameter being dropped: the returned parameter is not assigned to a caller's variable though the compiler should shriek at you. This form of calling still leaves the potential of memory leaks and sloppy programming. Elimination of memory leaks can be programmed out but not the sloppy programming; you still need to deal with the report card's verdict. Well, 50% improvement is not bad.

So the only alternative is to pass the report card as a parameter: either by reference or by pointer. My past experience with threads and pass-by-reference have not been good. In fact, depending on the compiler used, the program just did not work consistently! Passing by address pointer, though it contains the potential of a bad pointer, has never failed me. So my leanings are towards the pointer approach. Therefore parameter passing makes the global variable a local scope issue for the called routine with its parametric pre-conditions forced onto the calling routine. This pro-con debate is now settled for my routines that are multi-dimensional in findings: even a singular dimension gives the mechanism to build out extensions using new axes for actions, errors, specific results and ... In my mind's eye, this concludes how I will continue developing code. The hard part is to maintain this discipline.

October 2004 - grumblings from within. Optimism leads to re-trying. We'll have again a go of passing parameters by reference. I hope the compilers are more reliable this time in a multi-threaded environment than in the past. It certainly lowers constraint checks. Here goes. You'll get a post evaluation report if it's not successful.

Now for the good stuff — the report card. It should contain the following items:

1) status indicator — okay or failure
2) action taken — aborted,not fnd,fnd,inserted
3) ↑ error terminal when an error occured
4) ↑ symbol-table-entry if found or inserted
5) subscript value used

The report card is defined in Yacco2's grammar Terminal section that acts as a dictionary of terms.

6. Namespace sections.

⟨bns 6⟩ ≡
  namespace yacco2_stbl {

7.

⟨ens 7⟩ ≡
  } /* end of namspace **yacco2_stbl** */

8.

⟨uns 8⟩ ≡
  using namespace yacco2_stbl;

This code is used in section 34.

9. Yacco2's symbol table blueprint. Output of the code.

⟨`yacco2_stbl.cpp` 9⟩ ≡
  ⟨Include header file 22⟩;
  using namespace NS_yacco2_err_symbols;
  using namespace NS_yacco2_terminals;
  using namespace yacco2;
  ⟨Accrue source code 20⟩;

10. Maintenance.

The following subsections explain what to do in various cases.

11. How to compile the symbol table. Originally the project uses Microsoft's Visual Studio c++. Now it's Unix compiled. The below notes are in a Microsoft context. U can easily substitute a Unix context. In project \yacco2\compiler\grammars, the emitted code **yacco2_stbl** . *cpp* is compiled with the grammars used by Yacco2.

12. How to print out *cweave* report.
Fire up the DOS batch command processor and set the default directory as follows:
cd \yacco2\compiler\symbol table

To produce a PDF document, type in the following at the DOS PROMPT:
pdftex yacco2stbl

To produce a postscript document, for the visually impaired like my aging self due to the PDF styled greyed section numbers within the table of contents and index, type in the following at the DOS PROMPT:
tex yacco2stbl
dvips yacco2stbl

Use Ghostscript to print out the yacco2stbl.ps file and Adobe Reader for the pdf version.

13. Salient Points established.
Key points concluded:
1) Procedure calls only used. Results are returned thru a passed Report card parameter.
2) c++ native types used for efficiency with no memory leaks.
3) Namespace eliminates loosely defined entities spread across different spectrums.
4) Report card is the scratch pad of evolutionary results used throughout all the symbol table phrases.

14. How to expand the symbol table.
Defined as a cweb macro, *hash_table_size* is the size of the table expressed as a prime number. If you need to enlarge it, use a Google search on prime number and find the Java applelet and run it. Adjust this macro definition and re-generate the project symbol table in Visual Studio which creates a new ctangle **yacco2_stbl** . *cpp* c++ file. This file then needs to be recompiled. See *How to compile* above.

15. How to run the test.

Create the following DOS batch file:

```
rem file: t1.bat
@ECHO OFF
rem testdriver batch harness
rem first parm indicated the option to test
rem 1 - cmd line only
rem 2 - cmd line then P1 lex
rem 3 - cmd line then P3 syntax
rem 4 - runs the symbol table test
echo on
cd \yacco2\testdriver\debug
rem echo Test out symbol table
testdriver 4 \yacco2\compiler \grammars\fsm_class_phrase6.dat
```

Now double click on the batch file created "t1.bat". NT will launch the batch process and execute the "testdriver" with its passed parameters. The appropriate tests are displayed on the console of the batch process.

The test run should produce the following approximate facsimile:

```
C:\yacco2\testdriver>rem file: t1.bat
C:\yacco2\testdriver> rem note: remed out lines that have been tested okay
C:\yacco2\testdriver> cd \yacco2\testdriver\debug
C:\yacco2\testdriver\Debug> rem echo test out symbol table
C:\yacco2\testdriver\Debug> testdriver 4 /yacco2/compiler/grammars/fsm_class_phrase6.dat
Prime number used: 20011
Test zero len key
zero_len_name: zero-len-symbol
1st entry test
duplicate entry test
Error duplicate entry: dup-entry-in-sym-table
Same key address test
Error same key address: same-address-as-key-in-symbol-table
Fill up balance of table
Overflow test
Error overflow: sym-table-full
Find symbol test
Not found test
Get symbol by subscript test
Error subscript out of range: -1 subscript-out-of-range
Error not found if this message appears more than once: 17482
Remember 1 spot is left vacant in the algorithm!
Error subscript out of range: 20011 subscript-out-of-range
Clean up
enter any key then press Enter: x
```

16. Terminology.

Symbol's literal — a variable length array of characters delimited by the null character "00". In c++, it's a **const char** $*$ type. It is the value used to find a symbol in Yacco2's symbol table expressed by the string of characters bracketed by the double quotes: example is `"#user-declaration"`. Go see Yacco2's terminal definitions.

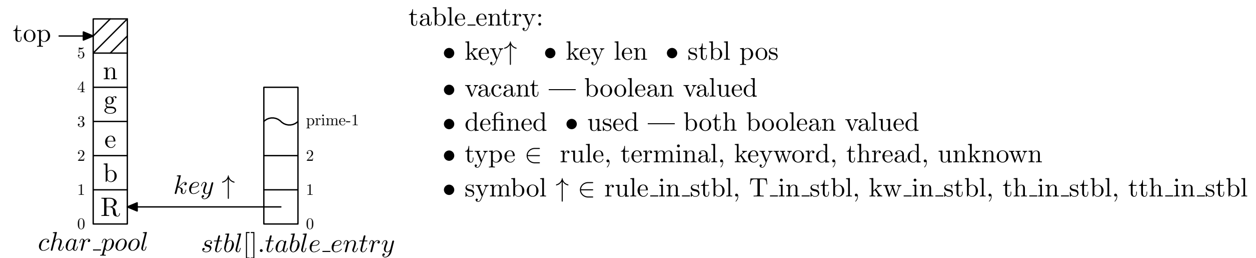
17. Facts.

Forget the tabloids and graffitti. Here is the gospel truth:

1) 1..*prime number* is the total spots in the symbol table
2) algorithm leaves 1 spot vacant.
3) *prime number* − 1 is the number of spots for insertion.
4) 0..*prime number* − 1 subscript range due to modulo calculation
5) Grammar symbols fall into lrk, err, raw character, or terminals
6) Each terminal group has its own namespace

The vacant spot only surfaces when the table is full and is pseudo-random in nature; this is due to value of the keys entered and where the hash function positions them.

18. Anatomy of symbol table.



The above figure shows the symbol table (stbl) as an array of fixed size records of table_entry having one entry in it. I use a quasi Pascal definition to describe the stbl cuz its more intuitively descriptive than its “c++” cousin. A hash function on the symbol’s literal name is used where it stores the key in its own character pool (char_pool). Access to the symbol table is thru the hash function or by subscript.

Various table entries:

rule_in_stbl:
- stbl index
- rule_def ↑
  - auto delete • auto abort
  - contains an epsilon subrule
  - rule name
  - parallel_monitor_phrase↑
  - subrules_phrase↑
  - rule_lhs_phrase↑
  - lhs directives map — <string, CAbs_lr1_sym↑>
- list of refered_rule ↑
  - referenced rule — rule_in_stbl↑

T_in_stbl:
- stbl index
- terminal_def ↑
  - auto delete • auto abort
  - terminal name
  - class name
  - directives map — <string, CAbs_lr1_sym↑>
- list of refered_T ↑
  - referenced terminal — T_in_stbl↑

th_in_stbl: Linker only
- stbl index
- thread_attributes ↑
  - transitive • monolithic
  - grammar name
  - namespace name
  - thread name
  - file name
  - enumeration value

tth_in_stbl: Linker only
- stbl index
- T_attributes ↑
  - fully qualified name
  - enumeration value

kw_in_stbl:
- stbl index
- CAbs_lr1_sym ↑

19. Create header file for symbol table environment.
Please note the appropriate include files that have been generated by other Yacco2's *cweb* sub-systems.

```
⟨ yacco2_stbl.h 19 ⟩ ≡
  ⟨ Preprocessor definitions ⟩
#ifndef yacco2_stbl_
#define yacco2_stbl_ 0
#include "yacco2.h"
#include "yacco2_err_symbols.h"
#include "yacco2_terminals.h"
#include <cstring>
  namespace yacco2_stbl {
    using namespace NS_yacco2_terminals;
    using namespace yacco2;
    extern table_entry stbl[hash_table_size];
    void hash_fnct(T_sym_tbl_report_card & Report, const char &Key);
    extern void add_sym_to_stbl
    (T_sym_tbl_report_card & Report, const char &Name
    , yacco2::CAbs_lr1_sym & Sym
    , table_entry::defined_or_used_typ Why
    , table_entry::entry_typ What);
    extern void find_sym_in_stbl(T_sym_tbl_report_card & Report, const char &Name);
    extern void test_program( );
    extern void get_sym_entry_by_sub(T_sym_tbl_report_card & Report, int Sub);
    extern char char_pool_[char_pool_size];
    extern int char_pool_idx_;
  };
#endif
```

20. Accrue source code.

```
⟨ Accrue source code 20 ⟩ ≡
  table_entry yacco2_stbl::stbl[hash_table_size] = {table_entry( )};
  char yacco2_stbl::char_pool_[char_pool_size] = { }
  ;
  int yacco2_stbl::char_pool_idx_(0);
```

See also sections 24, 25, 30, 33, and 34.
This code is used in section 9.

21. ⟨ Hashing 21 ⟩ ≡

```
  yacco2_stbl::hash_fnct(Report, Name);
```

This code is used in sections 25 and 30.

22. Include header file.

```
⟨ Include header file 22 ⟩ ≡
#include "yacco2_stbl.h"
```

This code is used in section 9.

23. Notes regarding hashing.
Due to the variable length of a symbol's name, the hash function goes across the character list whereby each character's value is added to the previous calculated hashed value shifted by some amount and then divided by the *hash_table_size* to get the remainder. Moduloing per character prevents a potential overflow.

The reason for not making the hash function external to the world is that its output is not universal. The calculated value is just the starting seed for the *add_sym_to_stbl* to find a vacant spot. Hence it is locally accessible by **yacco2_stbl**'s namespace procedures.

Constraints:

ip1: report card to contain the result

ip2: symbol's literal

Errors:

1) Zero len symbol

A potential error could occur if the symbol literal is not delimited by a "00" the null character. This could lead to an address violation. As Yacco2 creates this using the c++ string facility, this type of error can only be caused by the c++ compiler. Originally I had programmed an artificial maximun len of 512 characters but decided that this was overkill.

```
#define hash_table_size 20011   /* prime of course */
#define hash_table_full hash_table_size - 1
#define str_delimiter '\x00'
#define char_pool_size 100 * 1024
```

24. The hash procedure code: ***hash_fnct*****.** The length of key includes the delimiter. This allows the complete key to be moved into the character pool.

⟨ Accrue source code 20 ⟩ +≡

```
void yacco2_stbl::hash_fnct(T_sym_tbl_report_card &Report, const char &Key)
{
  Report.pos_ = -1;
  Report.key_len_ = strlen(&Key);
  if (Report.key_len_ ≡ 0) {
    Report.status_ = T_sym_tbl_report_card::failure;
    Report.err_entry_ = new Err_zero_len_sym;
    return;
  }
  Report.key_len_ += 1;
  const char *k = &Key;
  Report.pos_ = *k++;
  for ( ; *k ≠ 0; ++k) {   /* walk the plank characters */
    Report.pos_ = ((Report.pos_ * 257) + *k) % hash_table_size;
  }
  Report.status_ = T_sym_tbl_report_card::okay;
}
```

25. Add symbol to table routine: *add_sym_to_stbl*.
Algorithm L. Watch for the subtlety of the algorithm. It leaves one spot vacant so as to not check upfront on every entry whether the table is full. This check is done when the item is to be inserted which can produce a table overflow error. I was caught by this due to my assumptions going into the reading of the algorithm as opposed to the remarks on L4 which refines the assumptions to stop the looping due to the one spot always left empty. Again I prefer the upfront statement of expectations but this was done for teaching purposes. As C is not restriced to label length as in MIX, I use the label name to indicate intent rather than MIX's letter-number combination. The "leave one spot vacant" optimization benefits finding an entry.

Constraints:

ip1: report card to contain the result
ip2: symbol's literal
ip3: symbol object
ip4: reason for entry — defined or used
ip5: what type of symbol — terminal or rule

Errors:

1) duplicate entry
3) hash function errors
4) same key address between new symbol and item in symbol table

```
⟨ Accrue source code 20 ⟩ +≡
  void yacco2_stbl::add_sym_to_stbl
  (T_sym_tbl_report_card & Report
  , const char &Name
  , yacco2::CAbs_lr1_sym & Sym
  , table_entry::defined_or_used_typ Why
  , table_entry::entry_typ What)
  {
    static int guest_cnt(0);
  hash: ⟨ Hashing 21 ⟩;
    ⟨ Handle hash errors 26 ⟩;
  compare: ⟨ Compare 27 ⟩;
  advance: ⟨ Advance to next 28 ⟩;
  insert: ⟨ Insert 29 ⟩;
  }
```

26. Check the report card for appropriate behavior.

```
⟨ Handle hash errors 26 ⟩ ≡
  if (Report.status_ ≡ T_sym_tbl_report_card::failure) return;
  if (Report.status_ ≡ T_sym_tbl_report_card::fatal) return;
```

This code is used in sections 25 and 30.

27. Find a free spot for the symbol.

```
⟨ Compare 27 ⟩ ≡
  table_entry *te = &yacco2_stbl::stbl[Report.pos_];
  int r(0);
  if (te→vacant_ ≡ true) goto insert;
  r = strcmp(te→key_, &Name);
  if (r ≡ 0) {
    Report.status_ = T_sym_tbl_report_card::failure;
    Report.err_entry_ = new Err_dup_entry_in_sym_table;
    Report.tbl_entry_ = te;
    Report.key_len_ = 0;
    return;
  }
```

This code is used in section 25.

28. Linear probing to find spot.

```
⟨ Advance to next 28 ⟩ ≡
  −−Report.pos_;
  if (Report.pos_ < 0) Report.pos_ += hash_table_size;
  goto compare;
```

This code is used in section 25.

29. Insert the symbol into the table.

```
⟨ Insert 29 ⟩ ≡
  if (guest_cnt ≡ hash_table_full) {
    Report.status_ = T_sym_tbl_report_card::failure;
    Report.err_entry_ = new Err_sym_tbl_full;
    return;
  }
  if (char_pool_idx_ + Report.key_len_ > char_pool_size) {
    Report.status_ = T_sym_tbl_report_card::failure;
    Report.err_entry_ = new Err_sym_tbl_char_pool_full;
    return;
  }
  ++guest_cnt;
  te→pos_ = Report.pos_;
  te→vacant_ = false;
  char *key_name = &char_pool_[char_pool_idx_];
  strncpy(key_name, &Name, Report.key_len_);
  te→key_ = key_name;
  te→key_len_ = Report.key_len_;
  char_pool_idx_ += Report.key_len_;
  te→symbol_ = &Sym;
  te→type_ = What;
  if (Why ≡ table_entry::used) te→used_ = true;
  else te→defined_ = true;
  Report.status_ = T_sym_tbl_report_card::okay;
  Report.action_ = T_sym_tbl_report_card::inserted;
  Report.tbl_entry_ = te;
```

This code is used in section 25.

30. Find symbol in table routine: *find_sym_in_stbl*.
This routine's only purpose is to verification whether a symbol exists in the symbol table. It allows the programmer to be proactive rather than reactive to a symbol table problem. How so? Reactive would be when a duplicate error occured when adding a symbol to the table. As an error has been posted to the report card, now the programmer has to deal with this situation requiring cleanup if corrective action is taken that is different to the posted error. If the error is acceptable, then one can piggy back off the posted error. Having options is what it's all about.

Constraints:

ip1: report card to contain the result

ip2: symbol's literal

Errors:

1) hash function errors

```
⟨Accrue source code 20⟩ +≡
  void yacco2_stbl::find_sym_in_stbl(T_sym_tbl_report_card & Report, const char &Name)
  {
  hash: ⟨Hashing 21⟩;
    ⟨Handle hash errors 26⟩;
  compare: ⟨Find Compare 31⟩;
  advance: ⟨Find Advance to next 32⟩;
  }
```

31. Find a free spot for the symbol.

```
⟨Find Compare 31⟩ ≡
  Report.tbl_entry_ = &yacco2_stbl::stbl[Report.pos_];
  table_entry *te = Report.tbl_entry_;
  if (te→vacant_ ≡ true) {
    Report.status_ = T_sym_tbl_report_card::okay;
    Report.action_ = T_sym_tbl_report_card::not_fnd;
    return;
  }
  int r = strcmp(te→key_, &Name);
  if (r ≡ 0) {
    Report.status_ = T_sym_tbl_report_card::okay;
    Report.action_ = T_sym_tbl_report_card::fnd;
    return;
  }
```

This code is used in section 30.

32. Find Advance to next. Linear probing to find spot.

```
⟨Find Advance to next 32⟩ ≡
  −−Report.pos_;
  if (Report.pos_ < 0) Report.pos_ += hash_table_size;
  goto compare;
```

This code is used in section 30.

33. Get table entry by subscript: *get_sym_entry_by_sub*.

Constraints:

ip1: report card to contain the result

ip2: subscript where range 0..*hash_table_size* - 1

Errors:

1) subscript out of range

⟨ Accrue source code 20 ⟩ +≡

```
void yacco2_stbl::get_sym_entry_by_sub(T_sym_tbl_report_card & Report, int Sub)
{
  Report.pos_ = Sub;
  if ((Sub < 0) ∨ (Sub > hash_table_full)) {
    Report.status_ = T_sym_tbl_report_card::failure;
    Report.err_entry_ = new Err_subscript_out_of_range;
    return;
  }
  Report.tbl_entry_ = &yacco2_stbl::stbl[Sub];
  Report.status_ = T_sym_tbl_report_card::okay;
  if (Report.tbl_entry_→vacant_ ≡ false) Report.action_ = T_sym_tbl_report_card::fnd;
  else Report.action_ = T_sym_tbl_report_card::not_fnd;
}
```

34. Test program: *test_program***.**

A generator manufactures all the symbol literals up to table full. The literate routines below describe well each type of test with their appropriate range of errors.

```
⟨ Accrue source code 20 ⟩ +≡
void yacco2_stbl::test_program()
{
  ⟨ uns 8 ⟩;
  ⟨ Initial variables 35 ⟩;
  ⟨ Test 1st entry, duplicate, and same key address 36 ⟩;
  ⟨ Fill up balance of table 37 ⟩;
  ⟨ Test table overflow error 38 ⟩;
  ⟨ Test find symbol in table 39 ⟩;
  ⟨ Test get entry by subscript 40 ⟩;
  ⟨ Cleanup 41 ⟩;
}
```

35. Initial variables for test.

```
⟨ Initial variables 35 ⟩ ≡
const char *zero_len_name = "";
T_eol * a_good_sym = new T_eol;
int cnt(0);
T_sym_tbl_report_card report_card;
std::cout << "Prime␣number␣used:␣" << hash_table_size << std::endl;
std::cout << "Test␣zero␣len␣key" << std::endl;
add_sym_to_stbl(report_card, *zero_len_name, *a_good_sym, table_entry::used, table_entry::terminal);
if (report_card.status_ ≡ T_sym_tbl_report_card::failure) {
  std::cout << "Error␣zero_len_name:␣" << report_card.err_entry_→id__ << std::endl;
  delete report_card.err_entry_;
}
else {
  std::cout << "Should␣have␣been␣a␣zero_len_name␣error␣" << std::endl;
}
```

This code is used in section 34.

36. Test 1st entry, duplicate, and same key address.

⟨ Test 1st entry, duplicate, and same key address 36 ⟩ ≡

```
char buf[256];
std::string * sbuf[hash_table_size];   /* sbuf[0] symbolic vacant spot */
sbuf[1] = new string("1");
const char *f1st_dup = "1";
std::cout << "1st␣entry␣test" << std::endl;
add_sym_to_stbl(report_card, *sbuf[1]→c_str( ), *a_good_sym, table_entry::defed, table_entry::rule);
if (report_card.status_ ≡ T_sym_tbl_report_card::failure) {
  std::cout << "Error␣first␣entry␣failed:␣" << report_card.err_entry_→id__ << std::endl;
  delete report_card.err_entry_;
}
std::cout << "duplicate␣entry␣test" << std::endl;
add_sym_to_stbl(report_card, *f1st_dup, *a_good_sym, table_entry::defed, table_entry::rule);
if (report_card.status_ ≡ T_sym_tbl_report_card::failure) {
  std::cout << "Error␣duplicate␣entry:␣" << report_card.err_entry_→id__ << std::endl;
  delete report_card.err_entry_;
}
else {
  std::cout << "Should␣have␣been␣a␣duplicate␣entry␣error␣" << std::endl;
}
cout << "Same␣key␣address␣test␣" << endl;
add_sym_to_stbl(report_card, *sbuf[1]→c_str( ), *a_good_sym, table_entry::defed, table_entry::rule);
if (report_card.status_ ≡ T_sym_tbl_report_card::failure) {
  std::cout << "Error␣same␣key␣address:␣" << report_card.err_entry_→id__ << std::endl;
  delete report_card.err_entry_;
}
else {
  std::cout << "Should␣have␣been␣a␣same␣key␣address␣error␣" << std::endl;
}
```

This code is used in section 34.

37. Fill up balance of table.

⟨ Fill up balance of table 37 ⟩ ≡

```
const char *numeric_key = "%i";
std::cout << "Fill␣up␣balance␣of␣table" << std::endl;
for (cnt = 2; cnt ≤ hash_table_full; ++cnt) {
  sprintf(buf, numeric_key, cnt);
  sbuf[cnt] = new string(buf);
  add_sym_to_stbl(report_card, *sbuf[cnt]→c_str( ), *a_good_sym, table_entry::defed, table_entry::rule);
  if (report_card.status_ ≡ T_sym_tbl_report_card::failure) {
    std::cout << "Error␣entry␣failed:␣" << cnt << "␣" << report_card.err_entry_→id__ << std::endl;
    delete report_card.err_entry_;
  }
}
```

This code is used in section 34.

38. Test table overflow error.

⟨ Test table overflow error 38 ⟩ ≡

```
const char *overflow_key = "overflow";
std::cout << "Overflow test" << std::endl;
add_sym_to_stbl(report_card, *overflow_key, *a_good_sym, table_entry::defed, table_entry::rule);
if (report_card.status_ == T_sym_tbl_report_card::failure) {
  std::cout << "Error overflow: " << report_card.err_entry_->id__ << std::endl;
  delete report_card.err_entry_;
}
else {
  std::cout << "Should be overflow error" << std::endl;
}
```

This code is used in section 34.

39. Test find symbol in table. It walks all the keys just entered whose range is 1..*hash_table_full*. The last test is "not found" on the overflow key.

⟨ Test find symbol in table 39 ⟩ ≡

```
std::cout << "Find symbol test" << std::endl;
for (cnt = 1; cnt <= hash_table_full; ++cnt) {
  find_sym_in_stbl(report_card, *sbuf[cnt]->c_str());
  if (report_card.status_ == T_sym_tbl_report_card::failure) {
    std::cout << "Error Should have been found: " << cnt << " " << report_card.err_entry_->id__ <<
        std::endl;
    delete report_card.err_entry_;
  }
  else {
    if (report_card.action_ == T_sym_tbl_report_card::not_fnd) {
      std::cout << "Error Action should have been found: " << cnt << std::endl;
    }
  }
}
std::cout << "Not found test" << std::endl;
find_sym_in_stbl(report_card, *overflow_key);
if (report_card.status_ == T_sym_tbl_report_card::failure) {
  std::cout << "Should not be an error: " << cnt << " " << report_card.err_entry_->id__ << std::endl;
  delete report_card.err_entry_;
}
else {
  if (report_card.action_ == T_sym_tbl_report_card::fnd) {
    std::cout << "Error should be not found: " << std::endl;
  }
}
```

This code is used in section 34.

40. Test get entry by subscript. Walk thru -1..*hash_table_size*. This should produce 2 out-of-bound errors: one at each end of the spectrum traveled — -1 and *hash_table_full*. The range 0..*hash_table_full* should have all spots occupied except 1 vacant spot as explained in the Facts section.

⟨ Test get entry by subscript 40 ⟩ ≡

```
std::cout << "get␣symbol␣by␣subscript␣test" << std::endl;
for (cnt = −1; cnt ≤ hash_table_size; ++cnt) {
  get_sym_entry_by_sub(report_card, cnt);
  if (report_card.status_ ≡ T_sym_tbl_report_card::failure) {
    std::cout << "Error␣subscript␣out␣of␣range:␣" << cnt << "␣" << report_card.err_entry_→id__ <<
        std::endl;
    delete report_card.err_entry_;
  }
  else {
    if (report_card.action_ ≡ T_sym_tbl_report_card::not_fnd) {
      std::cout << "Error␣not␣found␣if␣this␣message␣appears␣more␣than␣once:␣" << cnt <<
          std::endl;
      std::cout << "Remember␣1␣spot␣is␣left␣vacant␣in␣the␣algorithm!" << std::endl;
    }
  }
}
```

This code is used in section 34.

41. Cleanup.

⟨ Cleanup 41 ⟩ ≡

```
std::cout << "Clean␣up" << std::endl;
delete a_good_sym;
for (cnt = 1; cnt ≤ hash_table_full; ++cnt) {
  delete sbuf[cnt];
}
```

This code is used in section 34.

42. Notes: bric-a-brac.

43. Remove key address dependency and copy key to a global *char_pool_* area.
Originally I used the passed key's address also as storage and guarded against the same key address passed by an error. Now what if one drains its info from cin into a common storage area? So I am more robust but with more overhead.

The symbol table can now be used elsewhere. The *CAbs_lr1_sym* value item would be changed to some generic or specific object. Possibly **void** $*$ with type casting or adjusted to the local requirements.
Improvement: 8 Mar. 2005

44. Index.

a_good_sym: 35, 36, 37, 38, 41.
*action_*: 29, 31, 33, 39, 40.
add_sym_to_stbl: 2, 19, 23, 25, 35, 36, 37, 38.
advance: 25, 30.
Algorithm: 5, 25.
buf: 36, 37.
c_str: 36, 37, 39.
CAbs_lr1_sym: 19, 25, 43.
*char_pool_*: 19, 20, 29, 43.
*char_pool_idx_*: 19, 20, 29.
char_pool_size: 19, 20, 23, 29.
cnt: 35, 37, 39, 40, 41.
code: 5.
compare: 25, 28, 30, 32.
compile: 14.
cout: 35, 36, 37, 38, 39, 40, 41.
cpp: 2, 11, 14.
cweave: 12.
cweb: 5, 19.
defed: 36, 37, 38.
*defined_*: 29.
defined_or_used_typ: 19, 25.
endl: 35, 36, 37, 38, 39, 40, 41.
entry_typ: 19, 25.
Err_dup_entry_in_sym_table: 27.
*err_entry_*: 24, 27, 29, 33, 35, 36, 37, 38, 39, 40.
Err_subscript_out_of_range: 33.
Err_sym_tbl_char_pool_full: 29.
Err_sym_tbl_full: 29.
Err_zero_len_sym: 24.
Facts: 17, 40.
failure: 24, 26, 27, 29, 33, 35, 36, 37, 38, 39, 40.
false: 29, 33.
fatal: 26.
find_sym_in_stbl: 2, 19, 30, 39.
fnd: 31, 33, 39.
f1st_dup: 36.
get_sym_entry_by_sub: 2, 19, 33, 40.
guest_cnt: 25, 29.
hash: 25, 30.
Hash: 5.
hash function: 17, 23.
hash_fnct: 19, 21, 24.
hash_table_full: 23, 29, 33, 37, 39, 40, 41.
hash_table_size: 2, 14, 19, 20, 23, 24, 28, 32, 33, 35, 36, 40.
How: 14.
How to compile the symbol table: 11, 14.
How to expand the symbol table: 14.
How to print —cweave— report: 12.
How to run the test: 15.
id__: 35, 36, 37, 38, 39, 40.
insert: 25, 27.
inserted: 29.
k: 24.
Key: 19, 24.
*key_*: 27, 29, 31.
*key_len_*: 24, 27, 29.
key_name: 29.
kw_in_stbl: 4.
Name: 19, 21, 25, 27, 29, 30, 31.
not_fnd: 31, 33, 39, 40.
NS_yacco2_err_symbols: 9.
NS_yacco2_terminals: 9, 19.
number: 17.
numeric_key: 37.
okay: 24, 29, 31, 33.
overflow_key: 38, 39.
*pos_*: 24, 27, 28, 29, 31, 32, 33.
prime: 17.
procedure: 5.
r: 27, 31.
refered_rule: 4.
refered_T: 4.
Report: 19, 21, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33.
report_card: 35, 36, 37, 38, 39, 40.
rule: 36, 37, 38.
rule_in_stbl: 3, 4.
Salient points : 13.
sbuf: 36, 37, 39, 41.
sprintf: 37.
*status_*: 24, 26, 27, 29, 31, 33, 35, 36, 37, 38, 39, 40.
stbl: 2, 19, 20, 27, 31, 33.
std: 35, 36, 37, 38, 39, 40, 41.
str_delimiter: 23.
strcmp: 27, 31.
string: 36, 37.
strlen: 24.
strncpy: 29.
Sub: 19, 33.
Sym: 19, 25, 29.
Symbol's literal: 16.
*symbol_*: 29.
T_eol: 35.
T_in_stbl: 3, 4.
T_sym_report_card: 2.
T_sym_tbl_report_card: 19, 24, 25, 26, 27, 29, 30, 31, 33, 35, 36, 37, 38, 39, 40.
table_entry: 19, 20, 25, 27, 29, 31, 35, 36, 37, 38.
*tbl_entry_*: 27, 29, 31, 33.
te: 27, 29, 31.
terminal: 35.

Terminals: 5.
Terminology: 16.
test_program: 2, 19, 34.
th_in_stbl: 4.
to: 14.
true: 27, 29, 31.
*type_*: 29.
used: 29, 35.
*used_*: 29.
*vacant_*: 27, 29, 31, 33.
What: 19, 25, 29.
Why: 19, 25, 29.
yacco2: 2, 9, 19, 25.
yacco2_err_symbols: 2.
yacco2_stbl: 2, 6, 7, 8, 11, 14, 19, 20, 21, 23, 24, 25, 27, 30, 31, 33, 34.
*yacco2_stbl_*: 19.
yacco2_terminals: 2.
yacco2stbl: 2.
zero_len_name: 35.

⟨ Accrue source code 20, 24, 25, 30, 33, 34 ⟩ Used in section 9.
⟨ Advance to next 28 ⟩ Used in section 25.
⟨ Cleanup 41 ⟩ Used in section 34.
⟨ Compare 27 ⟩ Used in section 25.
⟨ Fill up balance of table 37 ⟩ Used in section 34.
⟨ Find Advance to next 32 ⟩ Used in section 30.
⟨ Find Compare 31 ⟩ Used in section 30.
⟨ Handle hash errors 26 ⟩ Used in sections 25 and 30.
⟨ Hashing 21 ⟩ Used in sections 25 and 30.
⟨ Include header file 22 ⟩ Used in section 9.
⟨ Initial variables 35 ⟩ Used in section 34.
⟨ Insert 29 ⟩ Used in section 25.
⟨ Test 1st entry, duplicate, and same key address 36 ⟩ Used in section 34.
⟨ Test find symbol in table 39 ⟩ Used in section 34.
⟨ Test get entry by subscript 40 ⟩ Used in section 34.
⟨ Test table overflow error 38 ⟩ Used in section 34.
⟨ bns 6 ⟩
⟨ ens 7 ⟩
⟨ uns 8 ⟩ Used in section 34.
⟨ `yacco2_stbl.cpp` 9 ⟩
⟨ `yacco2_stbl.h` 19 ⟩

YACCO2˙STBL

	Section	Page
License	1	1
Summary of Yacco2 and Linker's Symbol Table	2	2
Synopsis of table entries	3	2
Special note	4	2
Introduction to Yacco2's Symbol Table	5	3
Namespace sections	6	4
Maintenance	10	4
Salient Points established	13	5
How to run the test	15	6
Terminology	16	6
Facts	17	7
Anatomy of symbol table	18	7
Create header file for symbol table environment	19	8
Notes regarding hashing	23	9
The hash procedure code: *hash_fnct*	24	9
Add symbol to table routine: *add_sym_to_stbl*	25	10
Find symbol in table routine: *find_sym_in_stbl*	30	12
Get table entry by subscript: *get_sym_entry_by_sub*	33	13
Test program: *test_program*	34	14
Notes: bric-a-brac	42	18
Index	44	19