HYACC User Manual

Created on 3/12/07. Last modified on 4/7/09.

Version 0.95

Department of Information and Computer Science, University of Hawaii
Please send all bug report and comments to chenx@hawaii.edu

This documentation introduces Hyacc and its usage.

1. Overview

1.1 Background

Many people have used Yacc. It is a LALR(1) compiler generator, often used with the lexical analyzer Lex to create compilers. There are many variations of Yacc, such as Bison and Berkeley yacc.

Hyacc is similar to Yacc in that it is a compiler generator. It is different from yacc in that it is a LR(0)/LALR(1)/LR(1) compiler generator. Hyacc can accept not only LR(0) and LALR(1) grammars, but also all LR(1) grammars. This is more powerful than Yacc.

Hyacc is pronounced as “HiYacc”, means Hawaii Yacc.

Specifically, based on the original LR(1) algorithm [Knuth], the practical general method [Pager77] is used to combine (weakly) compatible states to reduce the state space and increase the performance. Based on this, another optimization of unit production elimination [Pager77b] and its extension are also used in the hope of further reduce the size of the state space.

Besides, Hyacc also implemented LR(1) parser generation based on the lane-tracing algorithm [Pager77c][Pager73]. Hyacc also supports LALR(1) (based on lane-tracing) and LR(0).

Hyacc tries to be backward compatible with Yacc and Bison in the format of input file and command line switches.

Hyacc was developed under Cygwin, and has been tested in Fedora Core 4.0, Solaris and Suse 10.3. It is ANSI C compliant, and can be easily ported to other platforms.

1.2 Feature list

Current features:

1) Implements the original Knuth LR(1) algorithm [Knuth].
2) Implements the practical general method (weak compatibility) [Pager77]. This is a LR(1) algorithm.
3) Removes unit productions [Pager77b].
4) Removes repeated states after removing unit productions.
5) Implements the lane-tracing algorithm [Pager77c][Pager73]. This is a LR(1) algorithm.
6) Supports LALR(1) based on the lane-tracing algorithm phase 1.
7) Supports LR(0).
8) Allows empty productions.
9) Allows mid-production actions.
10) Allows these directives: %token, %left, %right, %expect, %start, %prec.
11) In case of ambiguous grammars, uses precedence and associativity to resolve conflicts. When unavoidable conflicts happen, in case of shift/reduce conflicts the default action is to use shift, in case of reduce/reduce conflicts the default is to use the production that appears first in a grammar.
12) Is compatible to yacc and bison in the ways of input file format, ambiguous grammar handling, error handling and output file format.
13) Works together with Lex. Or the users can provide the yylex() function by themselves.
14) If specified, can generate a graphviz input file for the parsing machine.
15) If specified, the generated compiler can record the parsing steps in a file.
16) Is ANSI C compliant.
17) Rich information in debug output.

What's not working so far and to be implemented:

1) Hyacc is not reentrant.
2) Hyacc does not support these Yacc directives: %nonassoc, %union, %type.
3) The optimization of removing unit productions can possibly lead to shift/shift conflicts in case of grammars that are ambiguous or not LR(1), and thus should not be applied in such situation.

1.3 A little note on the license

All the source files of Hyacc comes under the GPL license. The only exception is the file hyaccpar, which comes under the BSD license and is the skeleton parser driver of hyacc output. This should guarantee that Hyacc itself is protected by GPL, but the parser generators created by Hyacc can be used in both open source and proprietary software. This addresses the problem that Richard Stallman discussed in "Conditions for Using Bison" of his Bison 1.23 manual and Bison 1.24 manual.

2. Compilation and installation.

2.1 For Unix/Linux/Cygwin users

These files are included in the package for Unix/Linux/Cygwin users:

lane_tracing.h – The header file for lane-tracing functions.
mrt.h – The header file for multi-rooted tree.
stack_config.h – The header file for configuration stack.
y.h – The header file.
get_options.c – Gets command line switches.
gen_compiler.c – Generates compiler from the parsing machine created in y.c.
gen_graphviz.c – Generates graphviz input file for the parsing machine.
get_yacc_grammar.c – Parses input grammar file and feeds the result to y.c.
hyacc_path.c – Gives path information of hyaccpar and hyaccmanpage.
inst.c – Recreates hyacc_path.c at compilation time.
lr0.c – Contains functions for the LR(0) algorithm.
lane_tracing.c – Contains functions for the lane-tracing algorithm.
mrt.c – Contains functions for multi-rooted tree used in unit production elimination.
queue.c – A circular, expandable queue of integer.
stack_config.c – Contains functions for configuration stack.
state_hash_table.c – A hash table that makes searching states easy.
symbol_table.c – A hash table that stores information of grammar symbols.
upe.c – contains functions for unit production elimination.
version.c – Gives version information of Hyacc.
y.c – Creates LR(1) parsing machine, and applies the three optimizations.
hyaccpar – The LR(0)/LALR(1)/LR(1) compiler parse engine.
hyaccmanpage – The man page file.
hyacc.1 – Used to create the man page file.
makefile – The make file.

The makefile file is the utility used for compilation.

The options provided by the makefile are:

1) If compile the first time, type “make release” to compile the source code. This will take the INSTALL_PATH and feed it to inst.c, which recreate hyacc_path.c using the path information. Then it does the compilation to create the executable file hyacc.
2) If NOT compile the first time, type “make” to compile the source code, this is different from “make release” in that it does not create a new hyacc_path.c file.

3) Type “make debug” will do the same thing as “make release”, but also use the –g switch, so the user can use gdb debugger to debug the hyacc executable if anything goes wrong.
4) Type “make clean” will remove the hyacc executable file.
5) Type “make install” will do all the work of “make release”, and then copy executable hyacc, hyaccpar and hyaccmanpage to the destination directory.
6) Type “make uninstall” will tell the user what files to remove. The user needs to follow the instruction and manually remove the files hyacc, hyaccpar and hyaccmanpage from the installation folder.
7) Type “make dist” will create a distribution package in the format of hyacc_mm-dd-yy.tar.gz.

A typical process of compile and install Hyacc is:

If the user wants to install to the current directory:

1) Type “make release” will do all the work.

If the user wants to install to a different directory:

1) Modify the INSTALL_PATH macro at the top of the makefile. This tells where the user wants to install Hyacc. By default, this is the current directory. If you want the files to be installed to another location, make sure you have the permission to copy files there.
2) Type “make install”. That’s all.

2.2 For Windows users

For the files included in the package for windows users, the only difference from the package for Unix/Linux/Cygwin users is that now use the file hyacc_dos_path.c instead of hyacc_path.c, and there is no inst.c.

In hyacc_dos_path.c, if USE_CUR_DIR is defined as 1 (this is the default value), then the compiled binary uses current directory to locate resource files (hyaccpar and hyaccmanpage). If USE_CUR_DIR is defined as 0, then it uses C:\windows as the default installation path. The user should change this file if want to install to different location.

The user should have a C compiler in windows, like Microsoft Visual Studio 6.0. Using Microsoft Visual Studio 6.0 as an example, these steps are required:

1) Unzip the package.
2) Create an empty win32 console application from File → New.

3) Switch to File View, add all the *.c files to “Source Files”, and add *.h to “Header Files”.
4) From Build → Set Active Configuration, choose “Release”.
5) Build hyacc.exe using the Build menu or use the short-cut key F7.
6) Now the hyacc.exe is successfully built.
7) In hyacc_path_dos.c, by default USE_CUR_DIR is defined as 1. So hyacc uses the current directory to locate resource files (hyaccpar and hyaccmanpage). Otherwise, if USE_CUR_DIR is defined as 0, then C:\windows is the default installation path, and the user should copy hyacc.exe, hyaccpar and hyaccmanpage to C:\windows. This finishes the installation. Since C:\windows is on the system search path of windows, the user now can use the command “hyacc” anywhere in a dos window.
8) If the user chooses to install to a different location, he can modify hyacc_dos_path.c to tell hyacc.exe where to look for hyaccpar and hyaccmanpage files. He can use hyacc in the folder in which it resides. If he wants to use hyacc anywhere in the system, he needs to go to “My Computer → Properties → Advanced → Environment variables” (in windows XP) and add hyacc’s installation path to the PATH variable.

3. Usage

Type “hyacc –h” will show basic help message.
Type “hyacc –m” will show the man page file.

3.1 Input file

The input file format is compatible with Yacc and Bison. The user can check any Yacc/Bison manual or search on the Internet for the input file format.

The input file by default ends with suffix “.y”, but other suffix are allowed too.

$accept, $end and $placeholder are reserved words used by Hyacc. The user should NOT use these variables.

The user can use “/*” and “*/” to quote comments in each section of the input file.

Basically there are three sections, separated by “%%” directives.

Note: all “%%” and “%…” directives should start from the first column of the line.

3.1.1 Declaration section

The first section is declaration section. Between "%{" and "%}" is the declarations used by the output compiler file.

Then these Yacc/Bison-compatible directives declare terminal tokens used by Hyacc:

%token – declares a terminal token to be returned by Lex or user-defined yylex() function.

%start – declares the start symbol of the entire grammar. Hyacc adds an extra rule "$start → start_symbol" as the first rule. The result grammar is called the augmented grammar of the input grammar. The start symbol is a non-terminal.

%left – this declares a terminal token, as well as its precedence and associativity (left).

%right – this declares a terminal token, as well as its precedence and associativity (right).

All tokens declared by %left and %right have precedence defined by the relative location of the declaration. Tokens declared by the same %left or %right declaration have the same precedence. For tokens declared by different %left or %right, those appear later have high precedence. For example

%left '+' '-'
%left '*' '/'

Then '+' and '-' have the same precedence, so are '*' and '/'. But '*' and '/' have higher precedence than '+' and '-'.

These Yacc directives are not supported yet: %nonassoc, %union, %type, %pure_parser. There are more directives used by Bison, those are not supported and ignored by Hyacc at this time. In summary, only %start, %token, %left, %right, %expect and %prec are supported by Hyacc so far.

3.1.2 Rules section

The second section is the rules section, where the user specifies all the grammars. %prec directive can be used in this section at the end of a rule to specify the context-dependent precedence and associativity of the rule. For example: A → - C %prec UNARY declares this rule's precedence and associativity to be that of token UNARY, which should be declared in the declaration section.

In the rules section the user can put semantic actions of the rules at the end of each rule, quoted by "{" and "}". $$ indicates the value of the current rule. $n (n = 1, 2, 3…) indicates the value of the nth term on the RHS. For example:

A : B '+' C { $$ = $1 + $2; } /* $1 is the value of B, $2 is the value of C. */
  ;

Mid-action are not supported yet. Mid-actions are semantic actions in the middle of the RHS of a rule. For example:

A : B { $$ = $1; } '+' C
  ;

3.1.3 Code section

The third section is the code section, where the user puts all the other code he wants to go in the compiler file.

3.1.4 Example

An example hyacc input file is:

```
/* See http://www.gnu.org/software/bison/manual/html_mono/bison.html */
/* Infix notation calculator.  */

%{
       #define YYSTYPE double
       #include <math.h>
       #include <stdio.h>
       #include <stdlib.h>
       #include <ctype.h>
       int yylex (void);
       void yyerror (char const *);
       char * cursor = "#";
%}

/* Bison declarations.  */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation */

%% /* Grammar rules and actions follow.  */

     input:    /* empty */
             | input line
```

```
;

line:     '\n'          { printf ("\n%s ", cursor);   }
        | exp '\n'      { printf ("\t%.10g\n%s ", $1, cursor); }
        | error '\n'    { yyerrok;                    }
;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
%%

/* The lexical analyzer returns a double floating point
   number on the stack and the token NUM, or the numeric code
   of the character read if not a number.  It skips all blanks
   and tabs, and returns 0 for end-of-input.  */

#include <ctype.h>

int
yylex (void)
{
  int c;

  /* Skip white space.  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* Process numbers.  */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      //ungetc (c);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* Return end-of-input.  */
  if (c == EOF)
    return 0;
  /* Return a single char.  */
  return c;
}

int
main (void)
{
  printf("%s ", cursor);
```

```
  return yyparse ();
}

#include <stdio.h>

/* Called by yyparse on error.  */
void
yyerror (char const *s)
{
  fprintf (stderr, "%s\n%s", s, cursor);
}
```

Save this file as “example.y”, type command “hyacc example.y”. This will generate y.tab.c. Note that we have yylex() defined in “example.y” so we don’t need Lex in this case. Now type “gcc y.tab.c –o example” will create the compiler file “example”. Type “./example”, you can enter mathematical expressions like “-1.2 + 3” and the compiler will give results.

3.2 command line switches

Most options may be given in one of two forms: either a dash followed by a single letter, or two dashes followed by a long option name. Single letter switches are supported for all the options. Long name switches are supported for some options.

```
-b fileprefix
--file-prefix==fileprefix
       Specify a prefix to use for all hyacc output  file  names.   The
       names are chosen as if the input file were named fileprefix.c.


-c
       Use  this  switch  to  not  generate  parser  files (y.tab.c and
       y.tab.h).  This is useful when the user only wants to use the -v
       and -D switches to parse the grammar and check the y.output file
       about the grammar's information.

       -c generally is used with -v, -D and -C.


-C
       For the unit production removal optimization (when -O2 or -O3 is
       used), if a unit production rule has semantic action, when it is
       removed the semantic action won't be preserved,  so  the  output
       compiler will miss some code.

       To  solve this problem, by default HYACC adds a placeholder non-
       terminal to unit production rules with actions, so they won't be
       removed. E.g., from
       program : expression              {printf("answer = %d\n", $1);}
               ;
       to
       program : expression $PlaceHolder {printf("answer = %d\n", $1);}
               ;
       $PlaceHolder : /* empty */
                    ;

       If the -C switch is used, this default action will not be taken.
       This is used when the user wants to just parse the  grammar  and
       does  not care about generating a useful compiler. Specifically,
       -C is used together with switch -c.


-d
--define
       Write an extra output file containing macro definitions for  the
       token type names defined in the grammar.

       The file is named y.tab.h.

       This  output file is essential if you wish to put the definition
       of yylex in a separate source file, because yylex  needs  to  be
       able  to refer to token type codes and the variable yylaval.  In
```

such case y.tab.h should be included into the file containing yylex.

-D

Change the print option to debug file y.output. A user who checks the debug file should assume certain degree of knowledge to the LR(1) compiler theory and optimization algorithms.

If the -v options is used, a debug file y.output will be generated when hyacc parses the grammar file. Use of -D switch will automatically turn on the -v switch, and will allow to specify what kind of information to be included into y.output.

By default, use -v will output the information about the states, plus a short statistics summary of the number of the grammar's terminals, nonterminals, grammar rules and states. like the y.output file of yacc.

-D should be followed by a parameter from 0 ~ 14:

-D0
Include all the information available.

-D1
Include the grammar.

-D2
Include the parsing table.

-D3
Include the process of generating the parsing machine, basically, the number of states and the current state in each cycle.

-D4
This is useful only if at the time of compilation, in y.h USE_CONFIG_QUEUE_FOR_GET_CLOSURE is set to 0. This then will include the information of combining compatible configurations: the number of configurations before and after the combination. -D4 can be used together with -D3.

-D5
Include the information of the multi-rooted tree(s) built for the optimization of removing unit productions.

-D6
Include the information in the optimization of removing unit productions. Specifically, the new states created and the original states from which the new states are combined from.

-D7
Include the information of the step 4 in the optimization of removing unit productions. Specifically, this shows the states reachable from state 0.

-D8
Show the entire parsing table after removing unit productions,

including those states that will be removed.

-D9
Show a list of configurations and the theads of the strings after the scanning symbol.

-D10
Include information of the symbol hash table.

-D11
Include the information of shift/shift conflicts if any. This happens when the input grammar is not LR(1) or ambiguous, and the optimization of removing unit production is used. The occurrence of shift/shift conflicts means the optimization of removing unit productions (-O2 and -O3) cannot be applied to this grammar.

-D12
NOT to include the default information about states when the -v option is used. Use -D12 to show only the short statistics summary, and not the states list.

-D13
Include the statistics of configurations for each state, and also dump the state hash table.

-D14
Include the information of actual/pseudo states. An actual state number is the row number of that state in the parsing table. After the step of unit production removal, some states are removed but their rows still remain in the parsing table, thus the state's pseudo state number (counted by ignoring those removed states/rows) will be different.

-D15
Shows the originator and transitor list of each configuration, as well as the parent state list of each state. This is relevant when lane tracing is used.

-g
--graphviz
Generate a graphviz input file for the parsing machine.

-h
--help Print a usage summary of hyacc.

-l
--nolines
Don't put any #line preprocessor commands in the parser file. Ordinarily hyacc puts them in the parser file so that the C compiler and debuggers will associate errors with your source file,

the grammar file. This options causes them to associate errors with the parser file, treating it as an independent source file in its own right.

-m
--man-page
Show man page. Same as "man hyacc". This is used when the man page file exists in the same directory as the hyacc executable. So if installation moves this man page file to another location, you must use "man hyacc".

-o outfile
--output-file==outfile
Specify the name outfile for the parser file.

The other output files' names are constructed from outfile as described under the v and d switches.

-O
Specify the kind of optimization used to parse the yacc input file.

Basically, three optimizations are used: 1) Combine compatible states based on weak compatibility. 2) Remove unit productions. 3) Remove repeated states after optimization 2).

The -O switch should be followed by a parameter from 0 to 3:

-O0
No optimization is used.

-O1
Optimization 1) is used.

-O2
Optimizations 1) and 2) are used.

-O3
Optimizations 1), 2) and 3) are used.

By default, when -O switch is not specified, the optimization 1) of combining compatible states is used. So "hyacc file.y" is equivalent to "hyacc file.y -O1" or "hyacc -O1 file.y".

-P
--lane-tracing-pgm
Use LR(1) based on the lane-tracing algorithm. The lane-tracing algorithm starts from a LR(0) parsing machine, uses lane-tracing to obtain contexts to resolve shift/reduce and reduce/reduce conflicts. If some reduce/reduce conflicts are not resolved, then state-splitting based on the practical general method is used to resolve such conflicts.

-Q
--lane-tracing-ltt
Use LR(1) based on the lane-tracing algorithm. The lane-tracing algorithm starts from a LR(0) parsing machine, uses lane-tracing to obtain contexts to resolve shift/reduce and reduce/reduce conflicts. If some reduce/reduce conflicts are not resolved, then state-splitting based on a lane-tracing table is used to resolve such conflicts.

-R
--lalr1
Use LALR(1) based on lane-tracing.

-S
--lr0 Use LR(0).

-t
--debug
In the parser files, define the macro YYDEBUG to 1 if it is not already defined, so that the debugging facilities are compiled. When the generated compiler parses an input yacc file, the parse process will be recorded in an output file y.parse, which includes all the shift/reduce actions, associated state number and lookahead, as well as the content of state stack and symbol stack.

-v
--verbose
Write an extra output file containing verbose descriptions of the parser states and what is done for each type of lookahead token in that state.

This file also describes all the conflicts, both those resolved by operator precedence and the unresolved ones.

The file's name is y.output.

-V
--version
Print the version number of hyacc and exit.

EXAMPLES
Assume the input grammar file is arith.y.

The user wants y.tab.c only:
hyacc arith.y

The user wants y.tab.c and y.tab.h:

```
hyacc -d arith.y
```

The user wants the generated compiler create y.parse when parsing a program:

```
hyacc -dt arith.y
or
hyacc arith.y -d -t
```

The user wants y.ta.b, y.tab.h, and create a y.output file when parsing the grammar:

```
hyacc -dv arith.y
```

The user wants, y.tab.c, y.tab.h, y.output and wants to include no other information than the short statistics summary in y.output:

```
hyacc -dD12 arith.y
```

Here -D12 will suppress the states list.

The user wants y.tab.c and y.tab.h, use optimization 1) only, and wants a default y.output:

```
hyacc -d -O1 -v arith.y
or
hyacc -dO1v arith.y
```

The user wants to parse the grammar and check y.output for information, and doesn't need a compiler. While use all the optimizations, he wants to keep those unit productions with semantic actions:

```
hyacc -cCv arith.y
```

3.3 output files

y.tab.c and y.tab.h
The output compiler file is y.tab.c. If the –d switch is used, y.tab.h is created to be included by other source files, such as lex.yy.c created by Lex.

y.output
If the –v switch is used, y.output will be created, which contains various information about the LR(1) parser generator depending on the –Dn switch.

y.parse
If the –t switch is used, y.parse will be created when running the created compiler on a source file, explaining step by step the parsing process.

y.gviz
If the –g switch is used, y.gviz will be created, which can be used as the input file for graphviz to generate a graph for the parsing machine.

The user can change the output file names using the –o and –b switches.

4. Future perspective

The future will see Hyacc:

1) More complete in being compatible with the interface of Yacc/Bison.
2) More information in y.output file.
3) More optimizations to increase the performance.
4) More compression of the created parser tables.

5. References

[Aho] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. (1986)

[Knuth] Donald E. Knuth. On the translation of languages from left to right. Information & Control, 8(6):607–639. (1965)

[Nigel] Nigel P. Chapman. LR Parsing: Theory and Practice. (1987)

[Pager77] David Pager. A Practical General Method for Constructing LR(k) Parsers. Acta Informatica 7, 249 – 268 (1977)

[Pager77b] David Pager. Eliminating Unit Productions from LR Parsers. Acta Informatica 9, 31 – 59 (1977)

[Pager77c] David Pager. The Lane-Tracing Algorithm for Constructing LR(k) Parsers and Ways of Enhancing Its Efficiency. Information Sciences 12, 19 – 42 (1977)

[Pager73] David Pager. The lane tracing algorithm for constructing LR(k) parsers. Proceedings of the fifth annual ACM symposium on Theory of computing, 172 - 181 (1973)