AutoGen - The Automated Program Generator

For version 5.5, May 2003

Bruce Korb
bkorb@gnu.org

This is the second edition of the GNU AutoGen documentation,

Published by Bruce Korb, 910 Redwood Dr., Santa Cruz, CA 95060

1 Introduction

AutoGen is a tool designed for generating program files that contain repetitive text with varied substitutions. Its goal is to simplify the maintenance of programs that contain large amounts of repetitious text. This is especially valuable if there are several blocks of such text that must be kept synchronized in parallel tables.

One common example is the problem of maintaining the code required for processing program options. Processing options requires a minimum of four different constructs be kept in proper order in different places in your program. You need at least:

1. The flag character in the flag string,
2. code to process the flag when it is encountered,
3. a global state variable or two, and
4. a line in the usage text.

You will need more things besides this if you choose to implement long option names, rc/ini file processing, environment variables and so on. All of this can be done mechanically; with the proper templates and this program. In fact, it has already been done and AutoGen itself uses it See Chapter 7 [AutoOpts], page 66. For a simple example of Automated Option processing, See Section 7.2 [Quick Start], page 69. For a full list of the Automated Option features, See Section 7.1 [Features], page 66.

1.1 The Purpose of AutoGen

The idea of this program is to have a text file, a template if you will, that contains the general text of the desired output file. That file includes substitution expressions and sections of text that are replicated under the control of separate definition files.

AutoGen was designed with the following features:

1. The definitions are completely separate from the template. By completely isolating the definitions from the template it greatly increases the flexibility of the template implementation. A secondary goal is that a template user only needs to specify those data that are necessary to describe his application of a template.
2. Each datum in the definitions is named. Thus, the definitions can be rearranged, augmented and become obsolete without it being necessary to go back and clean up older definition files. Reduce incompatibilities!
3. Every definition name defines an array of values, even when there is only one entry. These arrays of values are used to control the replication of sections of the template.
4. There are named collections of definitions. They form a nested hierarchy. Associated values are collected and associated with a group name. These associated data are used collectively in sets of substitutions.
5. The template has special markers to indicate where substitutions are required, much like the `${VAR}` construct in a shell `here doc`. These markers are not fixed strings. They are specified at the start of each template. Template designers know best what fits into their syntax and can avoid marker conflicts.

   We did this because it is burdensome and difficult to avoid conflicts using either M4 tokenizaion or C preprocessor substitution rules. It also makes it easier to specify expressions that transform the value. Of course, our expressions are less cryptic than the shell methods.

6. These same markers are used, in conjunction with enclosed keywords, to indicate sections of text that are to be skipped and for sections of text that are to be repeated. This is a major improvement over using C preprocessing macros. With the C preprocessor, you have no way of selecting output text because it is an *un*varying, mechanical substitution process.
7. Finally, we supply methods for carefully controlling the output. Sometimes, it is just simply easier and clearer to compute some text or a value in one context when its application needs to be later. So, functions are available for saving text or values for later use.

1.2 A Simple Example

This is just one simple example that shows a few basic features. If you are interested, you also may run "make check" with the `VERBOSE` enviornment variable set and see a number of other examples in the '`agen5/test/testdir`' directory.

Assume you have an enumeration of names and you wish to associate some string with each name. Assume also, for the sake of this example, that it is either too complex or too large to maintain easily by hand. We will start by writing an abbreviated version of what the result is supposed to be. We will use that to construct our output templates.

In a header file, '`list.h`', you define the enumeration and the global array containing the associated strings:

```
typedef enum {
        IDX_ALPHA,
        IDX_BETA,
        IDX_OMEGA }  list_enum;

extern const char* az_name_list[ 3 ];
```

Then you also have '`list.c`' that defines the actual strings:

```
#include "list.h"
const char* az_name_list[] = {
        "some alpha stuff",
        "more beta stuff",
        "final omega stuff" };
```

First, we will define the information that is unique for each enumeration name/string pair. This would be placed in a file named, '`list.def`', for example.

```
autogen definitions list;
list = { list_element = alpha;
         list_info    = "some alpha stuff"; };
list = { list_info    = "more beta stuff";
         list_element = beta; };
list = { list_element = omega;
         list_info    = "final omega stuff"; };
```

The `autogen definitions list;` entry defines the file as an AutoGen definition file that uses a template named `list`. That is followed by three `list` entries that define the associations between the enumeration names and the strings. The order of the differently named elements inside of list is unimportant. They are reversed inside of the `beta` entry and the output is unaffected.

Now, to actually create the output, we need a template or two that can be expanded into the files you want. In this program, we use a single template that is capable of multiple output files. The definitions above refer to a '`list`' template, so it would normally be named, '`list.tpl`'.

It looks something like this. (For a full description, See Chapter 3 [Template File], page 17.)

```
[+ AutoGen5 template h c +]
[+ CASE (suffix) +][+
   ==  h  +]
typedef enum {[+
   FOR list "," +]
        IDX_[+ (string-upcase! (get "list_element")) +][+
   ENDFOR list +] }  list_enum;

extern const char* az_name_list[ [+ (count "list") +] ];
[+

   ==  c  +]
#include "list.h"
const char* az_name_list[] = {[+
  FOR list "," +]
        "[+list_info+]"[+
  ENDFOR list +] };[+

ESAC +]
```

The `[+ AutoGen5 template h c +]` text tells AutoGen that this is an AutoGen version 5 template file; that it is to be processed twice; that the start macro marker is `[+`; and the end marker is `+]`. The template will be processed first with a suffix value of `h` and then with `c`. Normally, the suffix values are appended to the '`base-name`' to create the output file name.

The `[+ == h +]` and `[+ == c +]` `CASE` selection clauses select different text for the two different passes. In this example, the output is nearly disjoint and could have been put in two separate templates. However, sometimes there are common sections and this is just an example.

The `[+FOR list "," +]` and `[+ ENDFOR list +]` clauses delimit a block of text that will be repeated for every definition of `list`. Inside of that block, the definition name-value pairs that are members of each `list` are available for substitutions.

The remainder of the macros are expressions. Some of these contain special expression functions that are dependent on AutoGen named values; others are simply Scheme expressions, the result of which will be inserted into the output text. Other expressions are names of AutoGen values. These values will be inserted into the output text. For example, `[+list_info+]` will result in the value associated with the name `list_info` being inserted between the double quotes and `(string-upcase! (get "list_element"))` will first "get" the value associated with the name `list_element`, then change the case of all the letters to upper case. The result will be inserted into the output document.

If you have compiled AutoGen, you can copy out the template and definitions as described above and run `autogen list.def`. This will produce exactly the hypothesized desired output.

One more point, too. Lets say you decided it was too much trouble to figure out how to use AutoGen, so you created this enumeration and string list with thousands of entries. Now, requirements have changed and it has become necessary to map a string containing the enumeration name into the enumeration number. With AutoGen, you just alter the template to emit the table of names. It will be guaranteed to be in the correct order, missing none of the entries. If you want to do that by hand, well, good luck.

1.3 csh/zsh caveat

AutoGen tries to use your normal shell so that you can supply shell code in a manner you are accustomed to using. If, however, you use csh or zsh, you cannot do this. Csh is sufficiently difficult to program that it is unsupported. Zsh, though largely programmable, also has some anomolies that make it incompatible with AutoGen usage. Therefore, when invoking AutoGen from these environments, you must be certain to set the SHELL environment variable to a Bourne-derived shell. e.g., sh, ksh or bash.

Any shell you choose for your own scripts need to follow these basic requirements:

1. It handles `trap $sig ":"` without output to standard out. This is done when the server shell is first started. If your shell does not handle this, then it may be able to by loading functions from its start up files.
2. At the beginning of each scriptlet, the command `\\cd $PWD` is inserted. This ensures that `cd` is not aliased to something peculiar and each scriptlet starts life in the execution directory.
3. At the end of each scriptlet, the command `echo mumble` is appended. The program you use as a shell must emit the single argument `mumble` on a line by itself.

1.4 A User's Perspective

Alexandre wrote:
>
> I'd appreciate opinions from others about advantages/disadvantages of
> each of these macro packages.

I am using AutoGen in my pet project, and find one of its best points to be that it separates the operational data from the implementation.

Indulge me for a few paragraphs, and all will be revealed: In the manual, Bruce cites the example of maintaining command line flags inside the source code; traditionally spreading usage information, flag names, letters and processing across several functions (if not files). Investing the time in writing a sort of boiler plate (a template in AutoGen terminology) pays by moving all of the option details (usage, flags names etc.) into a well structured table (a definition file if you will), so that adding a new command line option becomes a simple matter of adding a set of details to the table.

So far so good! Of course, now that there is a template, writing all of that tedious optargs processing and usage functions is no longer an issue. Creating a table of the options needed for the new project and running AutoGen generates all of the option processing code in C automatically from just the tabular data. AutoGen in fact already ships with such a template... AutoOpts.

One final consequence of the good separation in the design of AutoGen is that it is retargetable to a greater extent. The egcs/gcc/fixinc/inclhack.def can equally be used (with different templates) to create a shell script (inclhack.sh) or a c program (fixincl.c).

This is just the tip of the iceberg. AutoGen is far more powerful than these examples might indicate, and has many other varied uses. I am certain Bruce or I could supply you with many and varied examples, and I would heartily recommend that you try it for your project and see for yourself how it compares to m4.

As an aside, I would be interested to see whether someone might be persuaded to rationalise autoconf with AutoGen in place of m4... Ben, are you listening? autoconf-3.0! ‘kay? =)O|

Sincerely,
Gary V. Vaughan

2 AutoGen Definitions File

This chapter describes the syntax and semantics of the AutoGen definition file. In order to instantiate a template, you normally must provide a definitions file that identifies itself and contains some value definitions. Consequently, we keep it very simple. For "advanced" users, there are preprocessing directives, sparse arrays, named indexes and comments that may be used as well.

The definitions file is used to associate values with names. Every value is implicitly an array of values, even if there is only one value. Values may be either simple strings or compound collections of name-value pairs. An array may not contain both simple and compound members. Fundamentally, it is as simple as:

```
prog_name = "autogen";
flag = {
    name      = templ_dirs;
    value     = L;
    descrip   = "Template search directory list";
};
```

For purposes of commenting and controlling the processing of the definitions, C-style comments and most C preprocessing directives are honored. The major exception is that the `#if` directive is ignored, along with all following text through the matching `#endif` directive. The C preprocessor is not actually invoked, so C macro substitution is **not** performed.

2.1 The Identification Definition

The first definition in this file is used to identify it as a AutoGen file. It consists of the two keywords, '`autogen`' and '`definitions`' followed by the default template name and a terminating semi-colon (`;`). That is:

`AutoGen Definitions` *template-name*`;`

Note that, other than the name *template-name*, the words '`AutoGen`' and '`Definitions`' are searched for without case sensitivity. Most lookups in this program are case insensitive.

Also, if the input contains more identification definitions, they will be ignored. This is done so that you may include (see Section 2.5 [Directives], page 10) other definition files without an identification conflict.

AutoGen uses the name of the template to find the corresponding template file. It searches for the file in the following way, stopping when it finds the file:

1. It tries to open '`./`*template-name*'. If it fails,
2. it tries '`./`*template-name*`.tpl`'.
3. It searches for either of these files in the directories listed in the templ-dirs command line option.

If AutoGen fails to find the template file in one of these places, it prints an error message and exits.

2.2 Named Definitions

Any name may have multiple values associated with it in the definition file. If there is more than one instance, the **only** way to expand all of the copies of it is by using the FOR (see Section 3.6.13 [FOR], page 47) text function on it, as described in the next chapter.

There are two kinds of definitions, '`simple`' and '`compound`'. They are defined thus (see Section 2.9 [Full Syntax], page 14):

```
compound_name '=' '{' definition-list '}' ';'

simple_name '=' string ';'

no_text_name ';'
```

`No_text_name` is a simple definition with a shorthand empty string value. The string values for definitions may be specified in any of several formation rules.

2.2.1 Definition List

`definition-list` is a list of definitions that may or may not contain nested compound definitions. Any such definitions may **only** be expanded within a `FOR` block iterating over the containing compound definition. See Section 3.6.13 [FOR], page 47.

Here is, again, the example definitions from the previous chapter, with three additional name value pairs. Two with an empty value assigned (*first* and *last*), and a "global" *group_name*.

```
autogen definitions list;
group_name = example;
list = { list_element = alpha;  first;
         list_info    = "some alpha stuff"; };
list = { list_info    = "more beta stuff";
         list_element = beta; };
list = { list_element = omega;  last;
         list_info    = "final omega stuff"; };
```

2.2.2 Double Quote String

The string follows the C-style escaping (`\`, `\n`, `\f`, `\v`, etc.), plus octal character numbers specified as `\ooo`. The difference from "C" is that the string may span multiple lines. Like ANSI "C", a series of these strings, possibly intermixed with single quote strings, will be concatenated together.

2.2.3 Single Quote String

This is similar to the shell single-quote string. However, escapes `\` are honored before another escape, single quotes `'` and hash characters `#`. This latter is done specifically to disambiguate lines starting with a hash character inside of a quoted string. In other words,

```
fumble = '
#endif
';
```

could be misinterpreted by the definitions scanner, whereas this would not:

```
fumble = ’
\#endif
’;
```

As with the double quote string, a series of these, even intermixed with double quote strings, will be concatenated together.

2.2.4 Shell Output String

This is assembled according to the same rules as the double quote string, except that there is no concatenation of strings and the resulting string is written to a shell server process. The definition takes on the value of the output string.

NB The text is interpreted by a server shell. There may be left over state from previous `‘` processing and it may leave state for subsequent processing. However, a `cd` to the original directory is always issued before the new command is issued.

2.2.5 An Unquoted String

A simple string that does not contain white space *may* be left unquoted. The string must not contain any of the characters special to the definition text (i.e. `"`, `#`, `’`, `(`, `)`, `,`, `;`, `<`, `=`, `>`, `[`, `]`, `‘`, `{`, or `}`). This list is subject to change, but it will never contain underscore (`_`), period (`.`), slash (`/`), colon (`:`), hyphen (`-`) or backslash (`\\`). Basically, if the string looks like it is a normal DOS or UNIX file or variable name, and it is not one of two keywords (‘`autogen`’ or ‘`definitions`’) then it is OK to not quote it, otherwise you should.

2.2.6 Scheme Result String

A scheme result string must begin with an open parenthesis `(`. The scheme expression will be evaluated by Guile and the value will be the result. The AutoGen expression functions are **dis**abled at this stage, so do not use them.

2.2.7 A Here String

A ‘`here string`’ is formed in much the same way as a shell here doc. It is denoted with a doubled less than character and, optionally, a hyphen. This is followed by optional horizontal white space and an ending marker-identifier. This marker must follow the syntax rules for identifiers. Unlike the shell version, however, you must not quote this marker. The resulting string will start with the first character on the next line and continue up to but not including the newline that precedes the line that begins with the marker token. No backslash or any other kind of processing is done on this string. The characters are copied directly into the result string.

Here are two examples:

```
str1 = <<-  STR_END
        $quotes = " ’ ‘
        STR_END;

str2 = <<   STR_END
```

```
                $quotes = " ' `
                STR_END;
        STR_END;
```

The first string contains no new line characters. The first character is the dollar sign, the last the back quote.

The second string contains one new line character. The first character is the tab character preceeding the dollar sign. The last character is the semicolon after the `STR_END`. That `STR_END` does not end the string because it is not at the beginning of the line. In the preceeding case, the leading tab was stripped.

2.2.8 Concatenated Strings

If single or double quote characters are used, then you also have the option, a la ANSI-C syntax, of implicitly concatenating a series of them together, with intervening white space ignored.

NB You **cannot** use directives to alter the string content. That is,

```
str = "fumble"
#ifdef LATER
      "stumble"
#endif
      ;
```

will result in a syntax error. The preprocessing directives are not carried out by the C preprocessor. However,

```
str = '"fumble\n"
#ifdef LATER
"     stumble\n"
#endif
';
```

Will work. It will enclose the '`#ifdef LATER`' and '`#endif`' in the string. But it may also wreak havoc with the definition processing directives. The hash characters in the first column should be disambiguated with an escape \ or join them with previous lines: `"fumble\n#ifdef LATER....`

2.3 Assigning an Index to a Definition

In AutoGen, every name is implicitly an array of values. When assigning values, they are usually implicitly assiged to the next highest slot. They can also be specified explicitly:

```
mumble[9] = stumble;
mumble[0] = grumble;
```

If, subsequently, you assign a value to `mumble` without an index, its index will be `10`, not `1`. If indexes are specified, they must not cause conflicts.

`#define`-d names may also be used for index values. This is equivalent to the above:

```
#define FIRST 0
#define LAST  9
mumble[LAST]  = stumble;
```

```
mumble[FIRST] = grumble;
```

All values in a range do **not** have to be filled in. If you leave gaps, then you will have a sparse array. This is fine (see Section 3.6.13 [FOR], page 47). You have your choice of iterating over all the defined values, or iterating over a range of slots. This:

```
[+ FOR mumble +][+ ENDFOR +]
```

iterates over all and only the defined entries, whereas this:

```
[+ FOR mumble (for-by 1) +][+ ENDFOR +]
```

will iterate over all 10 "slots". Your template will likely have to contain something like this:

```
[+ IF (exist? (sprintf "mumble[%d]" (for-index))) +]
```

or else "mumble" will have to be a compound value that, say, always contains a "grumble" value:

```
[+ IF (exist? "grumble") +]
```

2.4 Dynamic Text

There are several methods for including dynamic content inside a definitions file. Three of them are mentioned above (Section 2.2.4 [shell-generated], page 8 and see Section 2.2.6 [scheme-generated], page 8) in the discussion of string formation rules. Another method uses the `#shell` processing directive. It will be discussed in the next section (see Section 2.5 [Directives], page 10). Guile/Scheme may also be used to yield to create definitions.

When the Scheme expression is preceeded by a backslash and single quote, then the expression is expected to be an alist of names and values that will be used to create AutoGen definitions.

This method can be be used as follows:

```
\'( (name  (value-expression))
    (name2 (another-expr))  )
```

This is entirely equivalent to:

```
name  = (value-expression);
name2 = (another-expr);
```

Under the covers, the expression gets handed off to a Guile function named `alist->autogen-def` in an expression that looks like this:

```
(alist->autogen-def
    ( (name (value-expression))  (name2 (another-expr)) ) )
```

2.5 Controlling What Gets Processed

Definition processing directives can **only** be processed if the '#' character is the first character on a line. Also, if you want a '#' as the first character of a line in one of your string assignments, you should either escape it by preceding it with a backslash '\', or by embedding it in the string as in `"\n#"`.

All of the normal C preprocessing directives are recognized, though several are ignored. There is also an additional `#shell` - `#endshell` pair. Another minor difference is that AutoGen directives must have the hash character (`#`) in column 1.

The final tweak is that `#!` is treated as a comment line. Using this feature, you can use: ‘`#! /usr/local/bin/autogen`’ as the first line of a definitons file, set the mode to executable and "run" the definitions file as if it were a direct invocation of AutoGen. This was done for its hack value.

The ignored directives are: ‘`#assert`’, ‘`#ident`’, ‘`#pragma`’, and ‘`#if`’. Note that when ignoring the `#if` directive, all intervening text through its matching `#endif` is also ignored, including the `#else` clause.

The AutoGen directives that affect the processing of definitions are:

`#define name [ <text> ]`

Will add the name to the define list as if it were a DEFINE program argument. Its value will be the first non-whitespace token following the name. Quotes are **not** processed.

After the definitions file has been processed, any remaining entries in the define list will be added to the environment.

`#elif`

This must follow an `#if` otherwise it will generate an error. It will be ignored.

`#else`

This must follow an `#if`, `#ifdef` or `#ifndef`. If it follows the `#if`, then it will be ignored. Otherwise, it will change the processing state to the reverse of what it was.

`#endif`

This must follow an `#if`, `#ifdef` or `#ifndef`. In all cases, this will resume normal processing of text.

`#endshell`

Ends the text processed by a command shell into autogen definitions.

`#error [ <descriptive text> ]`

This directive will cause AutoGen to stop processing and exit with a status of EXIT_FAILURE.

`#if [ <ignored conditional expression> ]`

`#if` expressions are not analyzed. **Everything** from here to the matching `#endif` is skipped.

`#ifdef name-to-test`

The definitions that follow, up to the matching `#endif` will be processed only if there is a corresponding `-Dname` command line option.

`#ifndef name-to-test`

The definitions that follow, up to the matching `#endif` will be processed only if there is **not** a corresponding `-Dname` command line option or there was a canceling `-Uname` option.

`#include unadorned-file-name`

This directive will insert definitions from another file into the current collection. If the file name is adorned with double quotes or angle brackets (as in a C program), then the include is ignored.

`#line`

Alters the current line number and/or file name. You may wish to use this directive if you extract definition source from other files. `getdefs` uses this mechanism so AutoGen will report the correct file and approximate line number of any errors found in extracted definitions.

`#option opt-name [ <text> ]`

This directive will pass the option name and associated text to the AutoOpts optionLoadLine routine (see [optionLoadLine], page 68). The option text may span multiple lines by continuing them with a backslash. The backslash/newline pair will be replaced with two space characters. This directive may be used to set a search path for locating template files For example, this:

```
#option templ-dirs $ENVVAR/dirname
```

will direct autogen to use the `ENVVAR` environment variable to find a directory named `dirname` that (may) contain templates. Since these directories are searched in most recently supplied first order, search directories supplied in this way will be searched before any supplied on the command line.

`#shell`

Invokes `$SHELL` or '`/bin/sh`' on a script that should generate AutoGen definitions. It does this using the same server process that handles the back-quoted ' text. **CAUTION** let not your `$SHELL` be `csh`.

`#undef name-to-undefine`

Will remove any entries from the define list that match the undef name pattern.

2.6 Pre-defined Names

When AutoGen starts, it tries to determine several names from the operating environment and put them into environment variables for use in both `#ifdef` tests in the definitions files and in shell scripts with environment variable tests. `__autogen__` is always defined. For other names, AutoGen will first try to use the POSIX version of the `sysinfo(2)` system call. Failing that, it will try for the POSIX `uname(2)` call. If neither is available, then only `"__autogen__"` will be inserted into the environment. In all cases, the associated names are converted to lower case, surrounded by doubled underscores and non-symbol characters are replaced with underscores.

With Solaris on a sparc platform, `sysinfo(2)` is available. The following strings are used:

- `SI_SYSNAME` (e.g., "__sunos__")
- `SI_HOSTNAME` (e.g., "__ellen__")
- `SI_ARCHITECTURE` (e.g., "__sparc__")
- `SI_HW_PROVIDER` (e.g., "__sun_microsystems__")
- `SI_PLATFORM` (e.g., "__sun_ultra_5_10__")
- `SI_MACHINE` (e.g., "__sun4u__")

For Linux and other operating systems that only support the `uname(2)` call, AutoGen will use these values:

- `sysname` (e.g., "__linux__")
- `machine` (e.g., "__i586__")
- `nodename` (e.g., "__bach__")

By testing these pre-defines in my definitions, you can select pieces of the definitions without resorting to writing shell scripts that parse the output of `uname(1)`. You can also segregate real C code from autogen definitions by testing for `"__autogen__"`.

```
#ifdef __bach__
  location = home;
#else
  location = work;
#endif
```

2.7 Commenting Your Definitions

The definitions file may contain C and C++ style comments.

```
/*
 *  This is a comment.  It continues for several lines and closes
 *  when the characters ’*’ and ’/’ appear together.
 */
// this comment is a single line comment
```

2.8 What it all looks like.

This is an extended example:

```
autogen definitions ‘template-name’;
/*
 *  This is a comment that describes what these
 *  definitions are all about.
 */
global = "value for a global text definition.";

/*
 *  Include a standard set of definitions
 */
#include standards.def

a_block = {
    a_field;
    a_subblock = {
        sub_name  = first;
        sub_field = "sub value.";
    };

#ifdef FEATURE
    a_subblock = {
        sub_name  = second;
    };
#endif

};
```

2.9 YACC Language Grammar

The preprocessing directives and comments are not part of the grammar. They are handled by the scanner/lexer. The following was extracted directly from the defParse.y source file:

```
definitions : identity def_list TK_END
                { $$ = (YYSTYPE)(rootDefCtx.pDefs = (tDefEntry*)$2); }
            | identity TK_END
                { $$ = makeEmptyDefs(); }
            ;

def_list    : definition           { $$ = $1; }
            | definition def_list  { $$ = addSibMacro( $1, $2 ); }
            | identity   def_list  { $$ = $2; }
            ;

identity    : TK_AUTOGEN TK_DEFINITIONS filename ';'
                { $$ = identify( $3 ); }
            ;

definition  : value_name ';'
                { $$ = makeMacro( $1, (YYSTYPE)"", VALTYP_TEXT ); }

            | value_name '=' text_list ';'
                { $$ = makeMacroList( $1, $3, VALTYP_TEXT ); }

            | value_name '=' block_list ';'
                { $$ = makeMacroList( $1, $3, VALTYP_BLOCK ); }
            ;

text_list   : anystring                { $$ = startList( $1 ); }
            | anystring ',' text_list  { $$ = appendList( $1, $3 ); }
            ;

block_list  : def_block                { $$ = startList( $1 ); }
            | def_block ',' block_list { $$ = appendList( $1, $3 ); }
            ;

def_block   : '{' def_list '}'         { $$ = $2; } ;

anystring   : filename      { $$ = $1; }
            | TK_NUMBER     { $$ = $1; } ;

filename    : TK_OTHER_NAME { $$ = $1; }
            | TK_STRING     { $$ = $1; }
```

```
                | TK_VAR_NAME   { $$ = $1; } ;

value_name      : TK_VAR_NAME
                    { $$ = findPlace( (YYSTYPE)$1, (YYSTYPE)NULL ); }

                | TK_VAR_NAME '[' TK_NUMBER ']'
                    { $$ = findPlace( (YYSTYPE)$1, (YYSTYPE)$3 ); }

                | TK_VAR_NAME '[' TK_VAR_NAME ']'
                    { $$ = findPlace( (YYSTYPE)$1, (YYSTYPE)$3 ); }
                ;
```

2.10 Alternate Definition Forms

There are several methods for supplying data values for templates.

'`no definitions`'

It is entirely possible to write a template that does not depend upon external definitions. Such a template would likely have an unvarying output, but be convenient nonetheless because of an external library of either AutoGen or Scheme functions, or both. This can be accommodated by providing the `--override-tpl` and `--no-definitions` options on the command line. See Chapter 5 [autogen Invocation], page 53.

'`CGI`' AutoGen behaves as a CGI server if the definitions input is from stdin and the environment variable `REQUEST_METHOD` is defined and set to either "GET" or "POST", See Section 6.2 [AutoGen CGI], page 63. Obviously, all the values are constrained to strings because there is no way to represent nested values.

'`XML`' AutoGen comes with a program named, `xml2ag`. Its output can either be redirected to a file for later use, or the program can be used as an AutoGen wrapper. See Section 8.6 [xml2ag Invocation], page 113.

The introductory template example (see Section 1.2 [Example Usage], page 2) can be rewritten in XML as follows:

```
<EXAMPLE  template="list.tpl">
<LIST list_element="alpha"
      list_info="some alpha stuff"/>
<LIST list_info="more beta stuff"
      list_element="beta"/>
<LIST list_element="omega"
      list_info="final omega stuff"/>
</EXAMPLE>
```

A more XML-normal form might look like this:

```
<EXAMPLE  template="list.tpl">
<LIST list_element="alpha">some alpha stuff</LIST>
<LIST list_element="beta" >more beta stuff</LIST>
<LIST list_element="omega">final omega stuff</LIST>
</EXAMPLE>
```

but you would have to change the template `list_info` references into `text` references.

‘`standard AutoGen definitions`’
Of course. :-)

3 AutoGen Template

The AutoGen template file defines the content of the output text. It is composed of two parts. The first part consists of a pseudo macro invocation and commentary. It is followed by the template proper.

This pseudo macro is special. It is used to identify the file as a AutoGen template file, fixing the starting and ending marks for the macro invocations in the rest of the file, specifying the list of suffixes to be generated by the template and, optionally, the shell to use for processing shell commands embedded in the template.

AutoGen-ing a file consists of copying text from the template to the output file until a start macro marker is found. The text from the start marker to the end marker constitutes the macro text. AutoGen macros may cause sections of the template to be skipped or processed several times. The process continues until the end of the template is reached. The process is repeated once for each suffix specified in the pseudo macro.

This chapter describes the format of the AutoGen template macros and the usage of the AutoGen native macros. Users may augment these by defining their own macros. See Section 3.6.4 [DEFINE], page 46.

3.1 Format of the Pseudo Macro

The pseudo macro is used to tell AutoGen how to process a template. It tells autogen:

1. The punctuation characters used to demarcate the start of a macro. It may be up to seven characters long and must be the first non-whitespace characters in the file.
2. That start marker must be immediately followed by the marker strings `"AutoGen5"` and then `"template"`, though capitalization is not important.

The next several components may be intermingled:

3. Zero, one or more suffix specifications tell AutoGen how many times to process the template file. No suffix specifications mean that it is to be processed once and that the generated text is to be written to stdout. The current suffix for each pass can be determined with the `(suffix)` scheme function (see Section 3.4.34 [SCM suffix], page 29).

   The suffix specification consists of a sequence of POSIX compliant file name characters and, optionally, an equal sign and a file name "printf"-style formatting string. Two string arguments are allowed for that string: the base name of the definition file and the current suffix (that being the text to the left of the equal sign). (Note: "POSIX compliant file name characters" consist of alphanumerics plus the period (`.`), hyphen (`-`) and underscore (`_`) characters.)
4. Comment lines: blank lines, lines starting with a hash mark [`#`]), and edit mode comments (text between pairs of `-*-` strings).
5. Scheme expressions may be inserted in order to make configuration changes before template processing begins. It is used, for example, to allow the template writer to specify the shell program that must be used to interpret the shell commands in the template. It can have no effect on any shell commands in the definitions file, as that file will have been processed by the time the pseudo macro is interpreted.

```
(setenv "SHELL" "/bin/sh")
```

This is extremely useful to ensure that the shell used is the one the template was written to use. By default, AutoGen determines the shell to use by user preferences. Sometimes, that can be the "csh", though.

The scheme expression can also be used to save a pre-existing output file for later text extraction (see Section 3.5.4 [SCM extract], page 31).

```
(shellf "mv -f %1$s.c %1$s.sav" (base-name))
```

6. Finally, the end-macro marker must be last. It must not begin with a POSIX file name character, and if it begins with an equal sign, then it must be separated from any suffix specification by white space.

It is generally a good idea to use some sort of opening bracket in the starting macro and closing bracket in the ending macro (e.g. `{`, `(`, `[`, or even `<` in the starting macro). It helps both visually and with editors capable of finding a balancing parenthesis. The closing marker may **not** begin with an open parenthesis, as that is used to enclose a scheme expression.

It is also helpful to avoid using the comment marker (`#`); the POSIXly acceptable file name characters period (`.`), hyphen (`-`) and underscore (`_`); and finally, it is advisable to avoid using any of the quote characters double, single or back-quote. But there is no special check for any of these advisories.

As an example, assume we want to use `[+` and `+]` as the start and end macro markers, and we wish to produce a ‘`.c`’ and a ‘`.h`’ file, then the first macro invocation will look something like this:

```
[+ AutoGen5 template -*- Mode: emacs-mode-of-choice -*-
h=chk-%s.h
c
# make sure we don’t use csh:
(setenv "SHELL" "/bin/sh")  +]
```

The template proper starts after the pseudo-macro. The starting character is either the first non-whitespace character or the first character after the newline that follows the end macro marker.

3.2 Naming a value

When an AutoGen value is specified in a template, it is specified by name. The name may be a simple name, or a compound name of several components. Since each named value in AutoGen is implicitly an array of one or more values, each component may have an index associated with it.

It looks like this:

```
comp-name-1 . comp-name-2 [ 2 ]
```

Note that if there are multiple components to a name, each component name is separated by a dot (`.`). Indexes follow a component name, enclosed in square brackets (`[` and `]`). The index may be either an integer or an integer-valued define name. The first component of the name is searched for in the current definition level. If not found, higher levels will be searched until either a value is found, or there are no more definition levels. Subsequent components of the name must be found within the context of the newly-current definition level. Also, if the named value is prefixed by a dot (`.`), then the value search is started in the current context only. No higher levels are searched.

If someone rewrites this, I'll incorporate it. :-)

3.3 Macro Expression Syntax

AutoGen has two types of expressions: full expressions and basic ones. A full AutoGen expression can appear by itself, or as the argument to certain AutoGen built-in macros: CASE, IF, ELIF, INCLUDE, INVOKE (explicit invocation, see Section 3.6.16 [INVOKE], page 49), and WHILE. If it appears by itself, the result is inserted into the output. If it is an argument to one of these macros, the macro code will act on it sensibly.

You are constrained to basic expressions only when passing arguments to user defined macros, See Section 3.6.4 [DEFINE], page 46.

The syntax of a full AutoGen expression is:

```
[[ <apply-code> ] <value-name> ] [ <basic-expr-1> [ <basic-expr-2> ]]
```

How the expression is evaluated depends upon the presence or absence of the apply code and value name. The "value name" is the name of an AutoGen defined value, or not. If it does not name such a value, the expression result is generally the empty string. All expressions must contain either a `value-name` or a `basic-expr`.

3.3.1 Apply Code

The "apply code" selected determines the method of evaluating the expression. There are five apply codes, including the non-use of an apply code.

‘`no apply code`’

This is the most common expression type. Expressions of this sort come in three flavors:

‘`<value-name>`’

The result is the value of `value-name`, if defined. Otherwise it is the empty string.

‘`<basic-expr>`’

The result of the basic expression is the result of the full expression, See Section 3.3.2 [basic expression], page 20.

‘`<value-name> <basic-expr>`’

If there is a defined value for `value-name`, then the `basic-expr` is evaluated. Otherwise, the result is the empty string.

‘`% <value-name> <basic-expr>`’

If `value-name` is defined, use `basic-expr` as a format string for sprintf. Then, if the `basic-expr` is either a back-quoted string or a parenthesized expression, then hand the result to the appropriate interpreter for further evaluation. Otherwise, for single and double quote strings, the result is the result of the sprintf operation. Naturally, if `value-name` is not defined, the result is the empty string.

For example, assume that `fumble` had the string value, `stumble`:

```
[+ % fumble ‘printf ’%%x\\n’ $%s‘ +]
```

This would cause the shell to evaluate `"printf ’%x\n’ $stumble"`. Assuming that the shell variable `stumble` had a numeric value, the expression result would be that number, in hex. Note the need for doubled percent characters and backslashes.

‘`? <value-name> <basic-expr-1> <basic-expr-2>`’

Two `basic-expr`-s are required. If the `value-name` is defined, then the first `basic-expr-1` is evaluated, otherwise `basic-expr-2` is.

‘`- <value-name> <basic-expr>`’

Evaluate `basic-expr` only if `value-name` is *not* defined.

‘`?% <value-name> <basic-expr-1> <basic-expr-2>`’

This combines the functions of ‘`?`’ and ‘`%`’. If `value-name` is defined, it behaves exactly like ‘`%`’, above, using `basic-expr-1`. If not defined, then `basic-expr-2` is evaluated.

For example, assume again that `fumble` had the string value, `stumble`:

```
[+ ?% fumble ‘cat $%s‘ ‘pwd‘ +]
```

This would cause the shell to evaluate `"cat $stumble"`. If `fumble` were not defined, then the result would be the name of our current directory.

3.3.2 Basic Expression

A basic expression can have one of the following forms:

‘`’STRING’`’

A single quoted string. Backslashes can be used to protect single quotes (`’`), hash characters (`#`), or backslashes (`\`) in the string. All other characters of STRING are output as-is when the single quoted string is evaluated. Backslashes are processed before the hash character for consistency with the definition syntax. It is needed there to avoid preprocessing conflicts.

‘`"STRING"`’

A double quoted string. This is a cooked text string as in C, except that they are not concatenated with adjacent strings. Evaluating `"STRING"` will output STRING with all backslash sequences interpreted.

‘`‘STRING‘`’

A back quoted string. When this expression is evaluated, STRING is first interpreted as a cooked string (as in ‘"STRING"’) and evaluated as a shell expression by the AutoGen server shell. This expression is replaced by the stdout output of the shell.

‘`(STRING)`’

A parenthesized expression. It will be passed to the Guile interpreter for evaluation and replaced by the resulting value.

Additionally, other than in the `%` and `?%` expressions, the Guile expressions may be introduced with the Guile comment character (`;`) and you may put a series of Guile expressions within a single macro. They will be implicitly evaluated as if they were arguments to the `(begin ...)` expression. The result will be the the result of the last Guile expression evaluated.

3.4 AutoGen Scheme Functions

AutoGen uses Guile to interpret Scheme expressions within AutoGen macros. All of the normal Guile functions are available, plus several extensions (see Section 3.5 [Common Functions], page 31) have been added to augment the repertoire of string manipulation functions and manage the state of AutoGen processing.

This section describes those functions that are specific to AutoGen. Please take note that these AutoGen specific functions are not loaded and thus not made available until after the command line options have been processed and the AutoGen definitions have been loaded. They may, of course, be used in Scheme functions that get defined at those times, but they cannot be invoked.

3.4.1 ‘`ag-function?`’ - test for function

Usage: (ag-function? ag-name)
return SCM_BOOL_T if a specified name is a user-defined AutoGen macro, otherwise return SCM_BOOL_F.

Arguments:
ag-name - name of AutoGen macro

3.4.2 ‘`base-name`’ - base output name

Usage: (base-name)
Returns a string containing the base name of the output file(s). Generally, this is also the base name of the definitions file.

This Scheme function takes no arguments.

3.4.3 ‘`count`’ - definition count

Usage: (count ag-name)
Count the number of entries for a definition. The input argument must be a string containing the name of the AutoGen values to be counted. If there is no value associated with the name, the result is an SCM immediate integer value of zero.

Arguments:
ag-name - name of AutoGen value

3.4.4 ‘`def-file`’ - definitions file name

Usage: (def-file)
Get the name of the definitions file. Returns the name of the source file containing the AutoGen definitions.

This Scheme function takes no arguments.

3.4.5 ‘`dne`’ - "Do Not Edit" warning

Usage: (dne prefix [first_prefix] [optpfx])
Generate a "DO NOT EDIT" or "EDIT WITH CARE" warning string. Which depends

on whether or not the `--writable` command line option was set. The first argument is a per-line string prefix. The optional second argument is a prefix for the first-line and, in read-only mode, activates the editor hints.

```
-*- buffer-read-only: t -*- vi: set ro:
```

The warning string also includes information about the template used to construct the file and the definitions used in its instantiation.

The optional third argument is used when the first argument is actually an invocation option and the prefix arguments get shifted. The first argument must be, specifically, `"-d"`. That is used to signify that the date stamp should not be inserted into the output.

Arguments:
prefix - string for starting each output line
first_prefix - Optional - for the first output line
optpfx - Optional - shifted prefix

3.4.6 ‘`error`’ - display message and exit

Usage: (error message)
The argument is a string that printed out as part of an error message. The message is formed from the formatting string:

```
DEFINITIONS ERROR in %s line %d for %s:  %s\n
```

The first three arguments to this format are provided by the routine and are: The name of the template file, the line within the template where the error was found, and the current output file name.

After displaying the message, the current output file is removed and autogen exits with the EXIT_FAILURE error code. IF, however, the argument begins with the number 0 (zero), or the string is the empty string, then processing continues with the next suffix.

Arguments:
message - message to display before exiting

3.4.7 ‘`exist?`’ - test for value name

Usage: (exist? ag-name)
return SCM_BOOL_T iff a specified name has an AutoGen value. The name may include indexes and/or member names. All but the last member name must be an aggregate definition. For example:

```
(exist? "foo[3].bar.baz")
```

will yield true if all of the following is true:
There is a member value of either group or string type named `baz` for some group value `bar` that is a member of the `foo` group with index `3`. There may be multiple entries of `bar` within `foo`, only one needs to contain a value for `baz`.

Arguments:
ag-name - name of AutoGen value

3.4.8 ‘`find-file`’ - locate a file in the search path

Usage: (find-file file-name [suffix])
AutoGen has a search path that it uses to locate template and definition files. This function will search the same list for ‘`file-name`’, both with and without the ‘`.suffix`’, if provided.

Arguments:
file-name - name of file with text
suffix - Optional - file suffix to try, too

3.4.9 ‘`first-for?`’ - detect first iteration

Usage: (first-for? [for_var])
Returns SCM_BOOL_T if the named FOR loop (or, if not named, the current innermost loop) is on the first pass through the data. Outside of any FOR loop, it returns SCM_UNDEFINED. See Section 3.6.13 [FOR], page 47.

Arguments:
for_var - Optional - which for loop

3.4.10 ‘`for-by`’ - set iteration step

Usage: (for-by by)
This function records the "step by" information for an AutoGen FOR function. Outside of the FOR macro itself, this function will emit an error. See Section 3.6.13 [FOR], page 47.

Arguments:
by - the iteration increment for the AutoGen FOR macro

3.4.11 ‘`for-from`’ - set initial index

Usage: (for-from from)
This function records the initial index information for an AutoGen FOR function. Outside of the FOR macro itself, this function will emit an error. See Section 3.6.13 [FOR], page 47.

Arguments:
from - the initial index for the AutoGen FOR macro

3.4.12 ‘`for-index`’ - get current loop index

Usage: (for-index [for_var])
Returns the current index for the named FOR loop. If not named, then the index for the innermost loop. Outside of any FOR loop, it returns SCM_UNDEFINED. See Section 3.6.13 [FOR], page 47.

Arguments:
for_var - Optional - which for loop

3.4.13 ‘`for-sep`’ - set loop separation string

Usage: (for-sep separator)
This function records the separation string that is to be inserted between each iteration of

an AutoGen FOR function. This is often nothing more than a comma. Outside of the FOR macro itself, this function will emit an error.

Arguments:
separator - the text to insert between the output of each FOR iteration

3.4.14 ‘`for-to`’ - set ending index

Usage: (for-to to)
This function records the terminating value information for an AutoGen FOR function. Outside of the FOR macro itself, this function will emit an error. See Section 3.6.13 [FOR], page 47.

Arguments:
to - the final index for the AutoGen FOR macro

3.4.15 ‘`get`’ - get named value

Usage: (get ag-name [alt-val])
Get the first string value associated with the name. It will either return the associated string value (if the name resolves), the alternate value (if one is provided), or else the empty string.

Arguments:
ag-name - name of AutoGen value
alt-val - Optional - value if not present

3.4.16 ‘`high-lim`’ - get highest value index

Usage: (high-lim ag-name)
Returns the highest index associated with an array of definitions. This is generally, but not necessarily, one less than the `count` value. (The indexes may be specified, rendering a non-zero based or sparse array of values.)

This is very useful for specifying the size of a zero-based array of values where not all values are present. For example:

```
tMyStruct myVals[ [+ (+ 1 (high-lim "my-val-list")) +] ];
```

Arguments:
ag-name - name of AutoGen value

3.4.17 ‘`last-for?`’ - detect last iteration

Usage: (last-for? [for_var])
Returns SCM_BOOL_T if the named FOR loop (or, if not named, the current innermost loop) is on the last pass through the data. Outside of any FOR loop, it returns SCM_UNDEFINED. See Section 3.6.13 [FOR], page 47.

Arguments:
for_var - Optional - which for loop

3.4.18 ‘`len`’ - get count of values

Usage: (len ag-name)
If the named object is a group definition, then "len" is the same as "count". Otherwise, if it is one or more text definitions, then it is the sum of their string lengths. If it is a single text definition, then it is equivalent to `(string-length (get "ag-name"))`.

Arguments:
ag-name - name of AutoGen value

3.4.19 ‘`low-lim`’ - get lowest value index

Usage: (low-lim ag-name)
Returns the lowest index associated with an array of definitions.

Arguments:
ag-name - name of AutoGen value

3.4.20 ‘`match-value?`’ - test for matching value

Usage: (match-value? op ag-name test-str)
This function answers the question, "Is there an AutoGen value named `ag-name` with a value that matches the pattern `test-str` using the match function `op`?" Return SCM_BOOL_T iff at least one occurrence of the specified name has such a value. The operator can be any function that takes two string arguments and yields a boolean. It is expected that you will use one of the string matching functions provided by AutoGen.
The value name must follow the same rules as the `ag-name` argument for `exist?` (see Section 3.4.7 [SCM exist?], page 23).

Arguments:
op - boolean result operator
ag-name - name of AutoGen value
test-str - string to test against

3.4.21 ‘`out-delete`’ - delete current output file

Usage: (out-delete)
Remove the current output file. Cease processing the template for the current suffix. It is an error if there are `push`-ed output files. Use the `(error "0")` scheme function instead. See Section 3.7 [output controls], page 50.

This Scheme function takes no arguments.

3.4.22 ‘`out-depth`’ - output file stack depth

Usage: (out-depth)
Returns the depth of the output file stack. See Section 3.7 [output controls], page 50.

This Scheme function takes no arguments.

3.4.23 ‘`out-move`’ - change name of output file

Usage: (out-move new-name)
Rename current output file. See Section 3.7 [output controls], page 50. Please note: changing the name will not save a temporary file from being deleted. It *may*, however, be used on the root output file.

Arguments:
new-name - new name for the current output file

3.4.24 ‘`out-name`’ - current output file name

Usage: (out-name)
Returns the name of the current output file. If the current file is a temporary, unnamed file, then it will scan up the chain until a real output file name is found. See Section 3.7 [output controls], page 50.

This Scheme function takes no arguments.

3.4.25 ‘`out-pop`’ - close current output file

Usage: (out-pop [disp])
If there has been a `push` on the output, then close that file and go back to the previously open file. It is an error if there has not been a `push`. See Section 3.7 [output controls], page 50.

If there is no argument, no further action is taken. Otherwise, the argument should be `#t` and the contents of the file are returned by the function.

Arguments:
disp - Optional - return contents of the file

3.4.26 ‘`out-push-add`’ - append output to file

Usage: (out-push-add file-name)
Identical to `push-new`, except the contents are **not** purged, but appended to. See Section 3.7 [output controls], page 50.

Arguments:
file-name - name of the file to append text to

3.4.27 ‘`out-push-new`’ - purge and create output file

Usage: (out-push-new [file-name])
Leave the current output file open, but purge and create a new file that will remain open until a `pop` `delete` or `switch` closes it. The file name is optional and, if omitted, the output will be sent to a temporary file that will be deleted when it is closed. See Section 3.7 [output controls], page 50.

Arguments:
file-name - Optional - name of the file to create

3.4.28 ‘`out-resume`’ - resume current output file

Usage: (out-resume suspName)
If there has been a suspended output, then make that output descriptor current again. That output must have been suspended with the same tag name given to this routine as its argument.

Arguments:
suspName - A name tag for reactivating

3.4.29 ‘`out-suspend`’ - suspend current output file

Usage: (out-suspend suspName)
If there has been a `push` on the output, then set aside the output descriptor for later reactiviation with `(out-resume "xxx")`. The tag name need not reflect the name of the output file. In fact, the output file may be an anonymous temporary file. You may also change the tag every time you suspend output to a file, because the tag names are forgotten as soon as the file has been "resumed".

Arguments:
suspName - A name tag for reactivating

3.4.30 ‘`out-switch`’ - close and create new output

Usage: (out-switch file-name)
Switch output files - close current file and make the current file pointer refer to the new file. This is equivalent to `out-pop` followed by `out-push-new`, except that you may not pop the base level output file, but you may `switch` it. See Section 3.7 [output controls], page 50.

Arguments:
file-name - name of the file to create

3.4.31 ‘`set-option`’ - Set a command line option

Usage: (set-option opt)
The text argument must be an option name followed by any needed option argument. Returns SCM_UNDEFINED.

Arguments:
opt - AutoGen option name + its argument

3.4.32 ‘`set-writable`’ - Make the output file be writable

Usage: (set-writable [set?])
This function will set the current output file to be writable (or not). This is only effective if neither the `--writable` nor `--not-writable` have been specified. This state is reset when the current suffix's output is complete.

Arguments:
set? - Optional - boolean arg, false to make output non-writable

3.4.33 ‘`stack`’ - make list of AutoGen values

Usage: (stack ag-name)
Create a scheme list of all the strings that are associated with a name. They must all be text values or we choke.

Arguments:
ag-name - AutoGen value name

3.4.34 ‘`suffix`’ - get the current suffix

Usage: (suffix)
Returns the current active suffix (see Section 3.1 [pseudo macro], page 17).

This Scheme function takes no arguments.

3.4.35 ‘`tpl-file`’ - get the template file name

Usage: (tpl-file)
Returns the name of the current template file.

This Scheme function takes no arguments.

3.4.36 ‘`tpl-file-line`’ - get the template file and line number

Usage: (tpl-file-line [msg-fmt])
Returns the file and line number of the current template macro using either the default format, "from %s line %d", or else the format you supply. For example, if you want to insert a "C" language file-line directive, you would supply the format "# %2$d %1$s".

Arguments:
msg-fmt - Optional - formatting for line message

3.4.37 ‘`make-header-guard`’ - make self-inclusion guard

Emit a `#ifndef`/`#define` sequence based upon the output file name and the provided prefix. It will also define a scheme variables named, `header-file` and `header-guard`. The `#define` name is composed as follows:

1. The first element is the string argument and a separating underscore.
2. That is followed by the name of the header file with illegal characters mapped to underscores.
3. The end of the name is always, `"_GUARD"`.
4. Finally, the entire string is mapped to upper case.

The final `#define` name is stored in an SCM symbol named `header-guard`. Consequently, the concluding `#endif` for the file should read something like:

```
#endif /* [+ (. header-guard) +] */
```

The name of the header file (the current output file) is also stored in an SCM symbol, `header-file`. Therefore, if you are also generating a C file that uses the previously generated header file, you can put this into that generated file:

```
#include "[+ (. header-file) +]"
```

Obviously, if you are going to produce more than one header file from a particular template, you will need to be careful how these SCM symbols get handled.

Arguments:
prefix - first segment of `#define` name

3.4.38 ‘`autogen-version`’ - autogen version number

This is a symbol defining the current AutoGen version number string. It was first defined in AutoGen-5.2.14. It is currently “5.5.4”.

3.5 Common Scheme Functions

This section describes a number of general purpose functions that make the kind of string processing that AutoGen does a little easier. Unlike the AutoGen specific functions (see Section 3.4 [AutoGen Functions], page 22), these functions are available for direct use during definition load time.

3.5.1 ‘`bsd`’ - BSD Public License

Usage: (bsd prog_name owner prefix)
Emit a string that contains the Free BSD Public License. It takes three arguments: `prefix` contains the string to start each output line. `owner` contains the copyright owner. `prog_name` contains the name of the program the copyright is about.

Arguments:
prog_name - name of the program under the BSD
owner - Grantor of the BSD License
prefix - String for starting each output line

3.5.2 ‘`c-string`’ - emit string for ANSI C

Usage: (c-string string)
Reform a string so that, when printed, the C compiler will be able to compile the data and construct a string that contains exactly what the current string contains. Many non-printing characters are replaced with escape sequences. Newlines are replaced with a backslash, an `n`, a closing quote, a newline, seven spaces and another re-opening quote. The compiler will implicitly concatenate them. The reader will see line breaks.

A K&R compiler will choke. Use `kr-string` for that compiler.

Arguments:
string - string to reformat

3.5.3 ‘`error-source-line`’ - display of file & line

Usage: (error-source-line)
This function is only invoked just before Guile displays an error message. It displays the file name and line number that triggered the evaluation error. You should not need to invoke this routine directly. Guile will do it automatically.

This Scheme function takes no arguments.

3.5.4 ‘`extract`’ - extract text from another file

Usage: (extract file-name marker-fmt [caveat] [default])
This function is used to help construct output files that may contain text that is carried from one version of the output to the next.

The first two arguments are required, the second are optional:

- The `file-name` argument is used to name the file that contains the demarcated text.

- The `marker-fmt` is a formatting string that is used to construct the starting and ending demarcation strings. The sprintf function is given the `marker-fmt` with two arguments. The first is either "START" or "END". The second is either "DO NOT CHANGE THIS COMMENT" or the optional `caveat` argument.
- `caveat` is presumed to be absent if it is the empty string (`""`). If absent, “DO NOT CHANGE THIS COMMENT” is used as the second string argument to the `marker-fmt`.
- When a `default` argument is supplied and no pre-existing text is found, then this text will be inserted between the START and END markers.

The resulting strings are presumed to be unique within the subject file. As a simplified example:

```
[+ (extract "fname" "// %s - SOMETHING - %s" ""
"example default") +]
```

will result in the following text being inserted into the output:

```
// START - SOMETHING - DO NOT CHANGE THIS COMMENT
example default
// END   - SOMETHING - DO NOT CHANGE THIS COMMENT
```

The “`example default`” string can then be carried forward to the next generation of the output, *provided* the output is not named `"fname"` *and* the old output is renamed to `"fname"` before AutoGen-eration begins.

NOTE: You can set aside previously generated source files inside the pseudo macro with a Guile/scheme function, extract the text you want to keep with this extract function. Just remember you should delete it at the end, too. Here is an example from my Finite State Machine generator:

```
[+ AutoGen5 Template  -*- Mode: text -*-
h=%s-fsm.h   c=%s-fsm.c
(shellf
"[ -f %1$s-fsm.h ] && mv -f %1$s-fsm.h .fsm.head
[ -f %1$s-fsm.c ] && mv -f %1$s-fsm.c .fsm.code" (base-name)) +]
```

This code will move the two previously produced output files to files named ".fsm.head" and ".fsm.code". At the end of the 'c' output processing, I delete them.

Arguments:
file-name - name of file with text
marker-fmt - format for marker text
caveat - Optional - warn about changing marker
default - Optional - default initial text

3.5.5 ‘`format-arg-count`’ - count the args to a format

Usage: (format-arg-count format)
Sometimes, it is useful to simply be able to figure out how many arguments are required by a format string. For example, if you are extracting a format string for the purpose of generating a macro to invoke a printf-like function, you can run the formatting string through this function to determine how many arguments to provide for in the macro. e.g. for this extraction text:

```
/*=fumble bumble
 * fmt: 'stumble %s: %d\n'
=*/
```

You may wish to generate a macro:

```
#define BUMBLE(a1,a2) printf_like(something,(a1),(a2))
```

You can do this by knowing that the format needs two arguments.

Arguments:
format - formatting string

3.5.6 ‘`fprintf`’ - format to a file

Usage: (fprintf port format [format-arg ...])
Format a string using arguments from the alist. Write to a specified port. The result will NOT appear in your output. Use this to print information messages to a template user.

Arguments:
port - Guile-scheme output port
format - formatting string
format-arg - Optional - list of arguments to formatting string

3.5.7 ‘`gperf`’ - perform a perfect hash function

Usage: (gperf name str)
Perform the perfect hash on the input string. This is only useful if you have previously created a gperf program with the `make-gperf` function See Section 3.5.15 [SCM make-gperf], page 35. The `name` you supply here must match the name used to create the program and the string to hash must be one of the strings supplied in the `make-gperf` string list. The result will be a perfect hash index.

See the documentation for `gperf(1GNU)` for more details.

Arguments:
name - name of hash list
str - string to hash

3.5.8 ‘`gpl`’ - GNU General Public License

Usage: (gpl prog-name prefix)
Emit a string that contains the GNU General Public License. It takes two arguments: `prefix` contains the string to start each output line, and `prog_name` contains the name of the program the copyright is about.

Arguments:
prog-name - name of the program under the GPL
prefix - String for starting each output line

3.5.9 ‘`hide-email`’ - convert eaddr to javascript

Usage: (hide-email display eaddr)
Hides an email address as a java scriptlett. The ’mailto:’ tag and the email address are coded bytes rather than plain text. They are also broken up.

Arguments:
display - display text
eaddr - email address

3.5.10 ‘`in?`’ - test for string in list

Usage: (in? test-string string-list ...)
Return SCM_BOOL_T if the first argument string is found in one of the entries in the second (list-of-strings) argument.

Arguments:
test-string - string to look for
string-list - list of strings to check

3.5.11 ‘`join`’ - join string list with separator

Usage: (join separator list ...)
With the first argument as the separator string, joins together an a-list of strings into one long string. The list may contain nested lists, partly because you cannot always control that.

Arguments:
separator - string to insert between entries
list - list of strings to join

3.5.12 ‘`kr-string`’ - emit string for K&R C

Usage: (kr-string string)
Reform a string so that, when printed, a K&R C compiler will be able to compile the data and construct a string that contains exactly what the current string contains. Many non-printing characters are replaced with escape sequences. New-lines are replaced with a backslash-n-backslash and newline sequence,

Arguments:
string - string to reformat

3.5.13 ‘`lgpl`’ - GNU Library General Public License

Usage: (lgpl prog_name owner prefix)
Emit a string that contains the GNU Library General Public License. It takes three arguments: `prefix` contains the string to start each output line. `owner` contains the copyright owner. `prog_name` contains the name of the program the copyright is about.

Arguments:
prog_name - name of the program under the LGPL
owner - Grantor of the LGPL
prefix - String for starting each output line

3.5.14 ‘`license`’ - an arbitrary license

Usage: (license lic_name prog_name owner prefix)
Emit a string that contains the named license. The license text is read from a file named, `lic_name`.lic, searching the standard directories. The file contents are used as a format argument to `printf`(3), with `prog_name` and `owner` as the two string formatting arguments. Each output line is automatically prefixed with the string `prefix`.

Arguments:
lic_name - file name of the license
prog_name - name of the licensed program or library
owner - Grantor of the License
prefix - String for starting each output line

3.5.15 ‘`make-gperf`’ - build a perfect hash function program

Usage: (make-gperf name strings ...)
Build a program to perform perfect hashes of a known list of input strings. This function produces no output, but prepares a program named, ‘`gperf_<name>`’ for use by the gperf function See Section 3.5.7 [SCM gperf], page 33.

This program will be obliterated within a few seconds after AutoGen exits.

Arguments:
name - name of hash list
strings - list of strings to hash

3.5.16 ‘`makefile-script`’ - create makefile script

Usage: (makefile-script text)
This function will take ordinary shell script text and reformat it so that it will work properly inside of a makefile shell script. Not every shell construct can be supported; the intent is to have most ordinary scripts work without much, if any, alteration.

The following transformations are performed on the source text:

1. Trailing whitespace on each line is stripped.
2. Except for the last line, the string, `" ; \\"` is appended to the end of every line that does not end with a backslash, semi-colon, conjunction operator or pipe. Note that this will mutilate multi-line quoted strings, but `make` renders it impossible to use multi-line constructs anyway.
3. If the line ends with a backslash, it is left alone.
4. If the line ends with one of the excepted operators, then a space and backslash is added.
5. The dollar sign character is doubled, unless it immediately precedes an opening parenthesis or the single character make macros ’*’, ’`<`’, ’`@`’, ’?’ or ’%’. Other single character make macros that do not have enclosing parentheses will fail. For shell usage of the `"$@"`, `"$?"` and `"$*"` macros, you must enclose them with curly braces, e.g., `"${?}"`. The ksh construct `$(<command>)` will not work. Though some `make`s accept `${var}` constructs, this function will assume it is for shell interpretation and double the dollar character. You must use `$(var)` for all `make` substitutions.

6. Double dollar signs are replaced by four before the next character is examined.
7. Every line is prefixed with a tab, unless the first line already starts with a tab.
8. The newline character on the last line, if present, is suppressed.
9. Blank lines are stripped.

This function is intended to be used approximately as follows:

```
$(TARGET) : $(DEPENDENCIES)
<+ (out-push-new) +>
....mostly arbitrary shell script text....
<+ (makefile-script (out-pop #t)) +>
```

Arguments:
text - the text of the script

3.5.17 ‘max’ - maximum value in list

Usage: (max list ...)
Return the maximum value in the list

Arguments:
list - list of values. Strings are converted to numbers

3.5.18 ‘min’ - minimum value in list

Usage: (min list ...)
Return the minimum value in the list

Arguments:
list - list of values. Strings are converted to numbers

3.5.19 ‘prefix’ - prefix lines with a string

Usage: (prefix prefix text)
Prefix every line in the second string with the first string.

For example, if the first string is "# " and the second contains:

```
two
lines
```

The result string will contain:

```
# two
# lines
```

Arguments:
prefix - string to insert at start of each line
text - multi-line block of text

3.5.20 ‘printf’ - format to stdout

Usage: (printf format [format-arg ...])
Format a string using arguments from the alist. Write to the standard out port. The result

will NOT appear in your output. Use this to print information messages to a template user. Use “(sprintf ...)” to add text to your document.

Arguments:
format - formatting string
format-arg - Optional - list of arguments to formatting string

3.5.21 ‘`raw-shell-str`’ - single quote shell string

Usage: (raw-shell-str string)
Convert the text of the string into a singly quoted string that a normal shell will process into the original string. (It will not do macro expansion later, either.) Contained single quotes become tripled, with the middle quote escaped with a backslash. Normal shells will reconstitute the original string.

NOTE: some shells will not correctly handle unusual non-printing characters. This routine works for most reasonably conventional ASCII strings.

Arguments:
string - string to transform

3.5.22 ‘`shell`’ - invoke a shell script

Usage: (shell command)
Generate a string by writing the value to a server shell and reading the output back in. The template programmer is responsible for ensuring that it completes within 10 seconds. If it does not, the server will be killed, the output tossed and a new server started.

Arguments:
command - shell command - the result value is stdout

3.5.23 ‘`shell-str`’ - double quote shell string

Usage: (shell-str string)
Convert the text of the string into a double quoted string that a normal shell will process into the original string, almost. It will add the escape character `\\` before two special characters to accomplish this: the backslash `\\` and double quote `"`.

NOTE: some shells will not correctly handle unusual non-printing characters. This routine works for most reasonably conventional ASCII strings.

WARNING:
This function omits the extra backslash in front of a backslash, however, if it is followed by either a backquote or a dollar sign. It must do this because otherwise it would be impossible to protect the dollar sign or backquote from shell evaluation. Consequently, it is not possible to render the strings `"\\$"` or ``"\\`"``. The lesser of two evils.

All others characters are copied directly into the output.

The `sub-shell-str` variation of this routine behaves identically, except that the extra backslash is omitted in front of `"` instead of `` ` ``. You have to think about it. I’m open to suggestions.

Meanwhile, the best way to document is with a detailed output example. If the backslashes make it through the text processing correctly, below you will see what happens with three example strings. The first example string contains a list of quoted `foo`s, the second is the same with a single backslash before the quote characters and the last is with two backslash escapes. Below each is the result of the `raw-shell-str`, `shell-str` and `sub-shell-str` functions.

```
foo[0]           ’foo’ "foo" ‘foo‘ $foo
raw-shell-str -> ’’\’’foo’\’’ "foo" ‘foo‘ $foo’
shell-str     -> "’foo’ \"foo\" ‘foo‘ $foo"
sub-shell-str -> ‘’foo’ "foo" \‘foo\‘ $foo‘

foo[1]           \’bar\’ \"bar\" \‘bar\‘ \$bar
raw-shell-str -> ’\’\’’bar\’\’’ \"bar\" \‘bar\‘ \$bar’
shell-str     -> "\\’bar\\’ \\\"bar\\\" \‘bar\‘ \$bar"
sub-shell-str -> ‘\\’bar\\’ \"bar\" \\\‘bar\\\‘ \$bar‘

foo[2]           \\’BAZ\\’ \\"BAZ\\" \\‘BAZ\\‘ \\$BAZ
raw-shell-str -> ’\\’\’’BAZ\\’\’’ \\"BAZ\\" \\‘BAZ\\‘ \\$BAZ’
shell-str     -> "\\\\’BAZ\\\\’ \\\\\"BAZ\\\\\" \\\‘BAZ\\\‘ \\\$BAZ"
sub-shell-str -> ‘\\\\’BAZ\\\\’ \\\"BAZ\\\" \\\\\‘BAZ\\\\\‘ \\\$BAZ‘
```

There should be four, three, five and three backslashes for the four examples on the last line, respectively. The next to last line should have four, five, three and three backslashes. If this was not accurately reproduced, take a look at the agen5/test/shell.test test. Notice the backslashes in front of the dollar signs. It goes from zero to one to three for the "cooked" string examples.

Arguments:
string - string to transform

3.5.24 ‘`shellf`’ - format a string, run shell

Usage: (shellf format [format-arg ...])
Format a string using arguments from the alist, then send the result to the shell for interpretation.

Arguments:
format - formatting string
format-arg - Optional - list of arguments to formatting string

3.5.25 ‘`sprintf`’ - format a string

Usage: (sprintf format [format-arg ...])
Format a string using arguments from the alist.

Arguments:
format - formatting string
format-arg - Optional - list of arguments to formatting string

3.5.26 ‘`string-capitalize`’ - capitalize a new string

Usage: (string-capitalize str)
Create a new SCM string containing the same text as the original, only all the first letter of each word is upper cased and all other letters are made lower case.

Arguments:
str - input string

3.5.27 ‘`string-capitalize!`’ - capitalize a string

Usage: (string-capitalize! str)
capitalize all the words in an SCM string.

Arguments:
str - input/output string

3.5.28 ‘`string-contains-eqv?`’ - caseless substring

Usage: (*=* text match)
string-contains-eqv?: Test to see if a string contains an equivalent string. ‘equivalent’ means the strings match, but without regard to character case and certain characters are considered ‘equivalent’. Viz., ’-’, ’_’ and ’^’ are equivalent.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.29 ‘`string-contains?`’ - substring match

Usage: (*==* text match)
string-contains?: Test to see if a string contains a substring. "strstr(3)" will find an address.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.30 ‘`string-downcase`’ - lower case a new string

Usage: (string-downcase str)
Create a new SCM string containing the same text as the original, only all the upper case letters are changed to lower case.

Arguments:
str - input string

3.5.31 ‘`string-downcase!`’ - make a string be lower case

Usage: (string-downcase! str)
Change to lower case all the characters in an SCM string.

Arguments:
str - input/output string

3.5.32 ‘`string-end-eqv-match?`’ - caseless regex ending

Usage: (*~ text match)
string-end-eqv-match?: Test to see if a string ends with a pattern. Case is not significant.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.33 ‘`string-end-match?`’ - regex match end

Usage: (*~~ text match)
string-end-match?: Test to see if a string ends with a pattern. Case is significant.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.34 ‘`string-ends-eqv?`’ - caseless string ending

Usage: (*= text match)
string-ends-eqv?: Test to see if a string ends with an equivalent string.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.35 ‘`string-ends-with?`’ - string ending

Usage: (*== text match)
string-ends-with?: Test to see if a string ends with a substring. strcmp(3) returns zero for comparing the string ends.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.36 ‘`string-equals?`’ - string matching

Usage: (== text match)
string-equals?: Test to see if two strings exactly match.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.37 ‘`string-eqv-match?`’ - caseless regex match

Usage: (~ text match)
string-eqv-match?: Test to see if a string fully matches a pattern. Case is not significant, but any character equivalences must be expressed in your regular expression.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.38 ‘`string-eqv?`’ - caseless string match

Usage: (= text match)
string-eqv?: Test to see if two strings are equivalent. ‘equivalent’ means the strings match, but without regard to character case and certain characters are considered ‘equivalent’. Viz., ’-’, ’_’ and ’^’ are equivalent. If the arguments are not strings, then the result of the numeric comparison is returned.

This is an overloaded operation. If the arguments are not both strings, then the query is passed through to `scm_num_eq_p()`.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.39 ‘`string-has-eqv-match?`’ - caseless regex contains

Usage: (*~* text match)
string-has-eqv-match?: Test to see if a string contains a pattern. Case is not significant.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.40 ‘`string-has-match?`’ - contained regex match

Usage: (*~~* text match)
string-has-match?: Test to see if a string contains a pattern. Case is significant.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.41 ‘`string-match?`’ - regex match

Usage: (~~ text match)
string-match?: Test to see if a string fully matches a pattern. Case is significant.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.42 ‘`string-start-eqv-match?`’ - caseless regex start

Usage: (~* text match)
string-start-eqv-match?: Test to see if a string starts with a pattern. Case is not significant.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.43 ‘`string-start-match?`’ - regex match start

Usage: (~~* text match)
string-start-match?: Test to see if a string starts with a pattern. Case is significant.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.44 ‘`string-starts-eqv?`’ - caseless string start

Usage: (=* text match)
string-starts-eqv?: Test to see if a string starts with an equivalent string.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.45 ‘`string-starts-with?`’ - string starting

Usage: (==* text match)
string-starts-with?: Test to see if a string starts with a substring.

Arguments:
text - text to test for pattern
match - pattern/substring to search for

3.5.46 ‘`string-substitute`’ - multiple global replacements

Usage: (string-substitute source match repl)
`match` and `repl` may be either a single string or a list of strings. Either way, they must have the same structure and number of elements. For example, to replace all less than and all greater than characters, do something like this:

```
(string-substitute source
("&"     "<"    ">")
("&amp;" "&lt;" "&gt;"))
```

Arguments:
source - string to transform
match - substring or substring list to be replaced
repl - replacement strings or substrings

3.5.47 ‘`string->c-name!`’ - map non-name chars to underscore

Usage: (string->c-name! str)
Change all the graphic characters that are invalid in a C name token into underscores. Whitespace characters are ignored. Any other character type (i.e. non-graphic and non-white) will cause a failure.

Arguments:
str - input/output string

3.5.48 ‘`string-tr`’ - convert characters with new result

Usage: (string-tr source match translation)
This is identical to `string-tr!`, except that it does not over-write the previous value.

Arguments:
source - string to transform
match - characters to be converted
translation - conversion list

3.5.49 ‘`string-tr!`’ - convert characters

Usage: (string-tr! source match translation)
This is the same as the `tr(1)` program, except the string to transform is the first argument. The second and third arguments are used to construct mapping arrays for the transformation of the first argument.

It is too bad this little program has so many different and incompatible implementations!

Arguments:
source - string to transform
match - characters to be converted
translation - conversion list

3.5.50 ‘`string-upcase`’ - upper case a new string

Usage: (string-upcase str)
Create a new SCM string containing the same text as the original, only all the lower case letters are changed to upper case.

Arguments:
str - input string

3.5.51 ‘`string-upcase!`’ - make a string be upper case

Usage: (string-upcase! str)
Change to upper case all the characters in an SCM string.

Arguments:
str - input/output string

3.5.52 ‘`sub-shell-str`’ - back quoted (sub-)shell string

Usage: (sub-shell-str string)
This function is substantially identical to `shell-str`, except that the quoting character is ‘ and the "leave the escape alone" character is ".

Arguments:
string - string to transform

3.5.53 ‘`sum`’ - sum of values in list

Usage: (sum list ...)
Compute the sum of the list of expressions.

Arguments:
list - list of values. Strings are converted to numbers

3.5.54 ‘`html-escape-encode`’ - escape special chars

Usage: (html-escape-encode str)
Substitute escape sequences for characters that are special to HTML/XML. It will replace `"&"`, `"<"` and `">"` with the strings, `"&amp;"`, `"&lt;"`, and `"&gt;"`, respectively.

Arguments:
str - string to transform

3.6 AutoGen Native Macros

This section describes the various AutoGen natively defined macros. Unlike the Scheme functions, some of these macros are "block macros" with a scope that extends through a terminating macro. Block macros must not overlap. That is to say, a block macro started within the scope of an encompassing block macro must have its matching end macro appear before the encompassing block macro is either ended or subdivided.

The block macros are these:

`CASE`	This macro has scope through the `ESAC` macro. The scope is subdivided by `SELECT` macros. You must have at least one `SELECT` macro.
`DEFINE`	This macro has scope through the `ENDDEF` macro. The defined user macro can never be a block macro.
`FOR`	This macro has scope through the `ENDFOR` macro.
`IF`	This macro has scope through the `ENDIF` macro. The scope may be subdivided by `ELIF` and `ELSE` macros. Obviously, there may be only one `ELSE` macro and it must be the last of these subdivisions.
`INCLUDE`	This macro has the scope of the included file. It is a block macro in the sense that the included file must not contain any incomplete block macros.
`WHILE`	This macro has scope through the `ENDWHILE` macro.

3.6.1 AutoGen Macro Syntax

The general syntax is:

```
[ { <native-macro-name> | <user-defined-name> } ] [ <arg> ... ]
```

The syntax for `<arg>` depends on the particular macro, but is generally a full expression (see Section 3.3 [expression syntax], page 19). Here are the exceptions to that general rule:

1. `INVOKE` macros, implicit or explicit, must be followed by a list of name/string value pairs. The string values are *simple expressions*, as described above.

   That is, the `INVOKE` syntax is either:

```
<user-macro-name> [ <name> [ = <expression> ] ... ]
```

or

```
INVOKE <name-expression> [ <name> [ = <expression> ] ... ]
```

2. AutoGen FOR macros must be in one of two forms:

   ```
   FOR <name> [ <separator-string> ]
   ```

   or

   ```
   FOR <name> (...Scheme expression list)
   ```

   where `<name>` must be a simple name and the Scheme expression list is expected to contain one or more of the `for-from`, `for-to`, `for-by`, and `for-sep` functions. (See Section 3.6.13 [FOR], page 47, and Section 3.4 [AutoGen Functions], page 22)
3. AutoGen `DEFINE` macros must be followed by a simple name. Anything after that is ignored. See Section 3.6.4 [DEFINE], page 46.
4. The AutoGen `COMMENT`, `ELSE`, `ESAC` and the `END*` macros take no arguments and ignore everything after the macro name (e.g. see Section 3.6.3 [COMMENT], page 46)

3.6.2 CASE - Select one of several template blocks

The arguments are evaluated and converted to a string, if necessary. (see Section 3.6.12 [EXPR], page 47) The scope of the macro is up to the matching ESAC function. Within the scope of a CASE, this string is matched against case selection macros. There are sixteen match macros that are derived from four different ways the test may be performed, plus an "always true" match. The code for each selection expression is formed as follows:

1. Must the match start matching from the beginning of the string? If not, then the match macro code starts with an asterisk (`*`).
2. Must the match finish matching at the end of the string? If not, then the match macro code ends with an asterisk (`*`).
3. Is the match a pattern match or a string comparison? If a comparison, use an equal sign (`=`). If a pattern match, use a tilde (`~`).
4. Is the match case sensitive? If alphabetic case is important, double the tilde or equal sign.
5. Do you need a default match when none of the others match? Use a single asterisk (`*`).

For example:

```
[+ CASE <full-expression> +]
[+ ~~*  "[Tt]est" +]reg exp must match at start, not at end
[+ ==   "TeSt"    +]a full-string, case sensitive compare
[+ =    "TEST"    +]a full-string, case insensitive compare
[+ *              +]always match - no testing
[+ ESAC +]
```

`<full-expression>` (see Section 3.3 [expression syntax], page 19) may be any expression, including the use of apply-codes and value-names. If the expression yields a number, it is converted to a decimal string.

These case selection codes have also been implemented as Scheme expression functions using the same codes (see Section 3.5 [Common Functions], page 31).

3.6.3 COMMENT - A block of comment to be ignored

This function can be specified by the user, but there will never be a situation where it will be invoked at emit time. The macro is actually removed from the internal representation.

If the native macro name code is `#`, then the entire macro function is treated as a comment and ignored.

3.6.4 DEFINE - Define a user AutoGen macro

This function will define a new macro. You must provide a name for the macro. You do not specify any arguments, though the invocation may specify a set of name/value pairs that are to be active during the processing of the macro.

```
[+ define foo +]
... macro body with macro functions ...
[+ enddef +]
... [+ foo bar='raw text' baz=<<text expression>> +]
```

Once the macro has been defined, this new macro can be invoked by specifying the macro name as the first token after the start macro marker. Alternatively, you may make the invocation explicitly invoke a defined macro by specifying `INVOKE` in the macro invocation. If you do that, the macro name can be computed with an expression that gets evaluated every time the INVOKE macro is encountered. See Section 3.6.16 [INVOKE], page 49.

Any remaining text in the macro invocation will be used to create new name/value pairs that only persist for the duration of the processing of the macro. The expressions are evaluated the same way basic expressions are evaluated. See Section 3.3 [expression syntax], page 19.

The resulting definitions are handled much like regular definitions, except:

1. The values may not be compound. That is, they may not contain nested name/value pairs.
2. The bindings go away when the macro is complete.
3. The name/value pairs are separated by whitespace instead of semi-colons.
4. Sequences of strings are not concatenated.

3.6.5 ELIF - Alternate Conditional Template Block

This macro must only appear after an `IF` function, and before any associated `ELSE` or `ENDIF` functions. It denotes the start of an alternate template block for the `IF` function. Its expression argument is evaluated as are the arguments to `IF`. For a complete description See Section 3.6.14 [IF], page 48.

3.6.6 ELSE - Alternate Template Block

This macro must only appear after an `IF` function, and before the associated `ENDIF` function. It denotes the start of an alternate template block for the `IF` function. For a complete description See Section 3.6.14 [IF], page 48.

3.6.7 ENDDEF - Ends a macro definition.

This macro ends the `DEFINE` function template block. For a complete description See Section 3.6.4 [DEFINE], page 46.

3.6.8 ENDFOR - Terminates the `FOR` function template block

This macro ends the `FOR` function template block. For a complete description See Section 3.6.13 [FOR], page 47.

3.6.9 ENDIF - Terminate the `IF` Template Block

This macro ends the `IF` function template block. For a complete description See Section 3.6.14 [IF], page 48.

3.6.10 ENDWHILE - Terminate the `WHILE` Template Block

This macro ends the `WHILE` function template block. For a complete description See Section 3.6.19 [WHILE], page 49.

3.6.11 ESAC - Terminate the `CASE` Template Block

This macro ends the `CASE` function template block. For a complete description, See Section 3.6.2 [CASE], page 45.

3.6.12 EXPR - Evaluate and emit an Expression

This macro does not have a name to cause it to be invoked explicitly, though if a macro starts with one of the apply codes or one of the simple expression markers, then an expression macro is inferred. The result of the expression evaluation (see Section 3.3 [expression syntax], page 19) is written to the current output.

3.6.13 FOR - Emit a template block multiple times

This macro has a slight variation on the standard syntax:

```
FOR <value-name> [ <separator-string> ]
```

or

```
FOR <value-name> (...Scheme expression list
```

or

```
FOR <value-name> IN "quoted string" unquoted-string ...
```

Other than for the last form, the first macro argument must be the name of an AutoGen value. If there is no value associated with the name, the `FOR` template block is skipped entirely. The scope of the `FOR` macro extends to the corresponding `ENDFOR` macro. The last form will create an array of string values named `<value-name>` that only exists within the context of this `FOR` loop. With this form, in order to use a `separator-string`, you must code it into the end of the template block using the `(last-for?)` predicate function (see Section 3.4.17 [SCM last-for?], page 25).

If there are any arguments after the `value-name`, the initial characters are used to determine the form. If the first character is either a semi-colon (`;`) or an opening parenthesis (`(`), then it is presumed to be a Scheme expression containing the FOR macro specific functions `for-from`, `for-by`, `for-to`, and/or `for-sep`. See Section 3.4 [AutoGen Functions], page 22. If it consists of an '`i`' an '`n`' and separated by white space from more text, then the `FOR x IN` form is processed. Otherwise, the remaining text is presumed to be a string for inserting between each iteration of the loop. This string will be emitted one time less than the number of iterations of the loop. That is, it is emitted after each loop, excepting for the last iteration.

If the from/by/to functions are invoked, they will specify which copies of the named value are to be processed. If there is no copy of the named value associated with a particular index, the `FOR` template block will be instantiated anyway. The template must use methods for detecting missing definitions and emitting default text. In this fashion, you can insert entries from a sparse or non-zero based array into a dense, zero based array.

NB: the `for-from`, `for-to`, `for-by` and `for-sep` functions are disabled outside of the context of the `FOR` macro. Likewise, the `first-for`, `last-for` and `for-index` functions are disabled outside of the range of a `FOR` block.

```
[+FOR var (for-from 0) (for-to <number>) (for-sep ",") +]
... text with various substitutions ...[+
ENDFOR var+]
```

this will repeat the `... text with various substitutions ...` <number>+1 times. Each repetition, except for the last, will have a comma `,` after it.

```
[+FOR var ",\n" +]
... text with various substitutions ...[+
ENDFOR var +]
```

This will do the same thing, but only for the index values of `var` that have actually been defined.

3.6.14 IF - Conditionally Emit a Template Block

Conditional block. Its arguments are evaluated (see Section 3.6.12 [EXPR], page 47) and if the result is non-zero or a string with one or more bytes, then the condition is true and the text from that point until a matched `ELIF`, `ELSE` or `ENDIF` is emitted. `ELIF` introduces a conditional alternative if the `IF` clause evaluated FALSE and `ELSE` introduces an unconditional alternative.

```
[+IF <full-expression> +]
emit things that are for the true condition[+

ELIF <full-expression-2> +]
emit things that are true maybe[+

ELSE "This may be a comment" +]
emit this if all but else fails[+

ENDIF "This may *also* be a comment" +]
```

`<full-expression>` may be any expression described in the `EXPR` expression function, including the use of apply-codes and value-names. If the expression yields an empty string, it is interpreted as *false*.

3.6.15 INCLUDE - Read in and emit a template block

The entire contents of the named file is inserted at this point. The contents of the file are processed for macro expansion. The arguments are eval-ed, so you may compute the name of the file to be included. The included file must not contain any incomplete function blocks. Function blocks are template text beginning with any of the macro functions '`CASE`', '`DEFINE`', '`FOR`', '`IF`' and '`WHILE`'; extending through their respective terminating macro functions.

3.6.16 INVOKE - Invoke a User Defined Macro

User defined macros may be invoked explicitly or implicitly. If you invoke one implicitly, the macro must begin with the name of the defined macro. Consequently, this may **not** be a computed value. If you explicitly invoke a user defined macro, the macro begins with the macro name `INVOKE` followed by a *basic expression* that must yield a known user defined macro. A macro name _must_ be found, or AutoGen will issue a diagnostic and exit.

Arguments are passed to the invoked macro by name. The text following the macro name must consist of a series of names each of which is followed by an equal sign (`=`) and a *basic expression* that yields a string.

The string values may contain template macros that are parsed the first time the macro is processed and evaluated again every time the macro is evaluated.

3.6.17 SELECT - Selection block for CASE function

This macro selects a block of text by matching an expression against the sample text expression evaluated in the `CASE` macro. See Section 3.6.2 [CASE], page 45.

You do not specify a `SELECT` macro with the word "select". Instead, you must use one of the 17 match operators described in the `CASE` macro description.

3.6.18 UNKNOWN - Either a user macro or a value name.

The macro text has started with a name not known to AutoGen. If, at run time, it turns out to be the name of a defined macro, then that macro is invoked. If it is not, then it is a conditional expression that is evaluated only if the name is defined at the time the macro is invoked.

You may not specify `UNKNOWN` explicitly.

3.6.19 WHILE - Conditionally loop over a Template Block

Conditionally repeated block. Its arguments are evaluated (see Section 3.6.12 [EXPR], page 47) and as long as the result is non-zero or a string with one or more bytes, then the condition is true and the text from that point until a matched `ENDWHILE` is emitted.

```
[+WHILE <full-expression> +]
emit things that are for the true condition[+

ENDWHILE +]
```

`<full-expression>` may be any expression described in the `EXPR` expression function, including the use of apply-codes and value-names. If the expression yields an empty string, it is interpreted as *false*.

3.7 Redirecting Output

AutoGen provides a means for redirecting the template output to different files. It is accomplished by providing a set of Scheme functions named `out-*` (see Section 3.4 [AutoGen Functions], page 22).

‘`out-push (see Section 3.4.27 [SCM out-push-new], page 27)`’
This allows you to logically "push" output files onto a stack.

‘`out-pop (see Section 3.4.25 [SCM out-pop], page 27)`’
This function closes the current output file and resumes output to the next one in the stack.

‘`out-suspend (see Section 3.4.29 [SCM out-suspend], page 28)`’
This function does not close the current output, but instead sets it aside for resumption with

‘`out-resume (see Section 3.4.28 [SCM out-resume], page 28)`’
This will put a named file descriptor back onto the top of stack so that it becomes the current output again.

‘`out-switch (see Section 3.4.30 [SCM out-switch], page 28)`’
This closes the current output and creates a new file, purging any preexisting one. This is a shortcut for "pop" followed by "push", but can also be done at the base level.

‘`out-move (see Section 3.4.23 [SCM out-move], page 27)`’
Renames the current output file without closing it.

There are also several functions for determining the output status. See Section 3.4 [AutoGen Functions], page 22.

4 Augmenting AutoGen Features

AutoGen was designed to be simple to enhance. You can do it by providing shell commands, Guile/Scheme macros or callout functions that can be invoked as a Guile macro. Here is how you do these.

4.1 Shell Output Commands

Shell commands are run inside of a server process. This means that, unlike '`make`', context is kept from one command to the next. Consequently, you can define a shell function in one place inside of your template and invoke it in another. You may also store values in shell variables for later reference. If you load functions from a file containing shell functions, they will remain until AutoGen exits.

If your shell script should determine that AutoGen should stop processing, the recommended method for stopping AutoGen is:

```
echo "some error text" >&2
kill -2 ${AG_pid}
```

4.2 Guile Macros

Guile also maintains context from one command to the next. This means you may define functions and variables in one place and reference them elsewhere. You also may load Guile macro definitions from a Scheme file by using the `--load-scheme` command line option (see Section 5.7 [autogen load-scheme], page 56). Beware, however, that the AutoGen specific scheme functions have not been loaded at this time, so though you may define functions that reference them, do not invoke the AutoGen functions at this time.

If your Scheme script should determine that AutoGen should stop processing, the recommended method for stopping AutoGen is:

```
(error "some error text")
```

4.3 Guile Callout Functions

Callout functions must be registered with Guile to work. This can be accomplished either by putting your routines into a shared library that contains a `void scm_init( void )` routine that registers these routines, or by building them into AutoGen.

To build them into AutoGen, you must place your routines in the source directory and name the files '`exp*.c`'. You also must have a stylized comment that '`getdefs`' can find that conforms to the following:

```
/*=gfunc <function-name>
 *
 * what:    <short one-liner>
 * general_use:
 * string:  <invocation-name-string>
 * exparg:  <name>, <description> [, ['optional'] [, 'list']]
 * doc:     A long description telling people how to use
 *          this function.
```

```
=*/
SCM
ag_scm_<function-name>( SCM arg_name[, ...] )
{ <code> }
```

‘`gfunc`’ You must have this exactly thus.

‘`<function-name>`’
This must follow C syntax for variable names

‘`<short one-liner>`’
This should be about a half a line long. It is used as a subsection title in this document.

‘`general_use:`’
You must supply this unless you are an AutoGen maintainer and are writing a function that queries or modifies the state of AutoGen.

‘`<invocation-name-string>`’
Normally, the `function-name` string will be transformed into a reasonable invocation name. However, that is not always true. If the result does not suit your needs, then supply an alternate string.

‘`exparg:`’ You must supply one for each argument to your function. All optional arguments must be last. The last of the optional arguments may be a list, if you choose.

‘`doc:`’ Please say something meaningful.

‘`[, ...]`’ Do not actually specify an ANSI ellipsis here. You must provide for all the arguments you specified with `exparg`.

See the Guile documentation for more details. More information is also available in a large comment at the beginning of the ‘`agen5/snarf.tpl`’ template file.

4.4 AutoGen Macros

There are two kinds those you define yourself and AutoGen native. The user-defined macros may be defined in your templates or loaded with the `--lib-template` option (See Section 3.6.4 [DEFINE], page 46 and Section 5.4 [autogen lib-template], page 55).

As for AutoGen native macros, do not add any. It is easy to do, but I won’t like it. The basic functions needed to accomplish looping over and selecting blocks of text have proven to be sufficient over a period of several years. New text transformations can be easily added via any of the AutoGen extension methods, as discussed above.

5 Invoking autogen

AutoGen creates text files from templates using external definitions. The definitions file ('<def-file>') can be specified with the 'definitions' option or as the command argument, but not both. Omitting it or specifying '-' will result in reading definitions from standard input.

The output file names are based on the template, but generally use the base name of the definition file. If standard in is read for the definitions, then 'stdin' will be used for that base name. The suffixes to the base name are gotten from the template. However, the template file may specify the entire output file name. The generated files are always created in the current directory. If you need to place output in an alternate directory, 'cd' to that directory and use the '–templ_dirs' option to search the original directory.

'loop-limit' is used in debugging to stop runaway expansions.

This chapter was generated by **AutoGen**, the aginfo template and the option descriptions for the **autogen** program. It documents the autogen usage text and option meanings.

This software is released under the GNU General Public License.

5.1 autogen usage help (-?)

This is the automatically generated usage text for autogen:

```
autogen - The Automated Program Generator - Ver. 5.5.4
USAGE:  autogen [ -<flag> [<val>] | --<name>[{=| }<val>] ]... [ <def-file> ]
  Flg Arg Option-Name    Description
   -L Str templ-dirs     Template search directory list
                                - may appear multiple times
   -T Str override-tpl   Override template file
                                - may not be preset
   -l Str lib-template   Library template file
                                - may appear multiple times
   -b Str base-name      Base name for output file(s)
                                - may not be preset
      Str definitions    Definitions input file
                                - disabled as --no-definitions
                                - enabled by default
                                - may not be preset
   -S Str load-scheme    Scheme code file to load
   -F Str load-functions Load scheme callout library
   -s Str skip-suffix    Omit the file with this suffix
                                - may not be preset
                                - may appear multiple times
   -o opt select-suffix  specify this output suffix
                                - may not be preset
                                - may appear multiple times
      no  source-time    set mod times to latest source
                                - disabled as --no-source-time
      Str equate         characters considered equivalent
      no  writable       Allow output files to be writable
```

```
                                - disabled as --not-writable
                                - may not be preset

The following options are often useful while debugging new templates:

  Flg Arg Option-Name    Description
      Num loop-limit     Limit on increment loops
                                  it must lie in one of the ranges:
                                  -1 exactly, or
                                  1 to 16777216
   -t Num timeout        Time limit for servers
                                  it must lie in the range: 0 to 3600
      KWd trace          tracing level of detail
      Str trace-out      tracing output file or filter

These options can be used to control what gets processed
in the definitions files and template files.

  Flg Arg Option-Name    Description
   -D Str define         name to add to definition list
                                - may appear multiple times
   -U Str undefine       definition list removal pattern
                                - an alternate for define

Auto-supported Options:

  Flg Arg Option-Name    Description
   -v opt version        Output version information and exit
   -? no  help           Display usage information and exit
   -! no  more-help      Extended usage information passed thru pager
   -> opt save-opts      Save the option state to an rc file
   -< Str load-opts      Load options from an rc file
                                - disabled as --no-load-opts
                                - may appear multiple times

Options are specified by doubled hyphens and their name
or by a single hyphen and the flag character.

AutoGen creates text files from templates using external definitions.

The following option preset mechanisms are supported:
 - reading file /dev/null
 - reading file ./.autogenrc
 - examining environment variables named AUTOGEN_*

The valid trace option keywords are:
        nothing
        templates
        block-macros
        expressions
        everything

The definitions file (‘<def-file>’) can be specified with the
‘definitions’ option or as the command argument, but not both.
Omitting it or specifying ‘-’ will result in reading definitions from
standard input.

The output file names are based on the template, but generally use the
base name of the definition file.  If standard in is read for the
definitions, then ‘stdin’ will be used for that base name.  The
suffixes to the base name are gotten from the template.  However, the
template file may specify the entire output file name.  The generated
files are always created in the current directory.  If you need to
place output in an alternate directory, ‘cd’ to that directory and use
the ‘--templ_dirs’ option to search the original directory.

‘loop-limit’ is used in debugging to stop runaway expansions.

please send bug reports to:  autogen-bugs@lists.sf.net
```

5.2 templ-dirs option (-L)

This is the “template search directory list” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

Add a directory to the list of directories to search when opening a template, either as the primary template or an included one. The last entry has the highest priority in the search list. That is to say, they are searched in reverse order.

5.3 override-tpl option (-T)

This is the “override template file” option.

This option has some usage constraints. It:

- may not be preset with environment variables or in initialization (rc) files.

Definition files specify the standard template that is to be expanded. This option will override that name and expand a different template.

5.4 lib-template option (-l)

This is the “library template file” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

DEFINE macros are saved from this template file for use in processing the main macro file. Template text aside from the DEFINE macros is is ignored.

5.5 base-name option (-b)

This is the “base name for output file(s)” option.

This option has some usage constraints. It:

- may not be preset with environment variables or in initialization (rc) files.

A template may specify the exact name of the output file. Normally, it does not. Instead, the name is composed of the base name of the definitions file with suffixes appended. This option will override the base name derived from the definitions file name. This is required if there is no definitions file and advisable if definitions are being read from stdin. If the definitions are being read from standard in, the base name defaults to ‘`stdin`’.

5.6 definitions option

This is the “definitions input file” option.

This option has some usage constraints. It:

- is enabled by default.
- may not be preset with environment variables or in initialization (rc) files.

Use this argument to specify the input definitions file with a command line option. If you do not specify this option, then there must be a command line argument that specifies the file, even if only to specify stdin with a hyphen (`-`). Specify, `--no-definitions` when you wish to process a template without any active AutoGen definitions.

5.7 load-scheme option (-S)

This is the “scheme code file to load” option. Use this option to pre-load Scheme scripts into the Guile interpreter before template processing begins. Please note that the AutoGen specific functions are not loaded until after argument processing. So, though they may be specified in lambda functions you define, they may not be invoked until after option processing is complete.

5.8 load-functions option (-F)

This is the “load scheme callout library” option. This option is used to load Guile-scheme callout functions. The automatically called initialization routine `scm_init` must be used to register these routines or data. This routine can be generated by using the following command and the ‘snarf.tpl’ template. Read the introductory comment in ‘snarf.tpl’ to see what the ‘getdefs(1AG)’ comment must contain.

First, create a config file for `getdefs`, and then invoke `getdefs` loading that file:

```
cat > getdefs.cfg <<EOF
subblock    exparg=arg_name,arg_desc,arg_optional,arg_list
defs-to-get gfunc
template    snarf
srcfile
linenum
assign      group = name_of_some_group
assign      init  = _init
EOF

getdefs load=getdefs.cfg <<source-file-list>>
```

Note, however, that your functions must be named:

```
name_of_some_group_scm_<<function_name>>(...)
```

so you may wish to use a shorter group name.

5.9 skip-suffix option (-s)

This is the “omit the file with this suffix” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.
- may not be preset with environment variables or in initialization (rc) files.

Occasionally, it may not be desirable to produce all of the output files specified in the template. (For example, only the ‘`.h`’ header file, but not the ‘`.c`’ program text.) To do this specify `--skip-suffix=c` on the command line.

5.10 select-suffix option (-o)

This is the “specify this output suffix” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.
- may not be preset with environment variables or in initialization (rc) files.

If you wish to override the suffix specifications in the template, you can use one or more copies of this option. See the suffix specification in the Section 3.1 [pseudo macro], page 17 section of the info doc.

5.11 source-time option

This is the “set mod times to latest source” option. If you stamp your output files with the ‘DNE’ macro output, then your output files will always be different, even if the content has not really changed. If you use this option, then the modification time of the output files will change only if the input files change. This will help reduce unneeded builds.

5.12 equate option

This is the “characters considered equivalent” option. This option will alter the list of characters considered equivalent. The default are the three characters, `"_-^"`. (The latter is conventional on a Tandem, and I do a lot of work on the Tandem.)

5.13 writable option

This is the “allow output files to be writable” option.

This option has some usage constraints. It:

- may not be preset with environment variables or in initialization (rc) files.

This option will leave output files writable.Normally, output files are read-only.

5.14 loop-limit option

This is the “limit on increment loops” option. This option prevents runaway loops. For example, if you accidentally specify, `"FOR x (for-from 1) (for-to -1) (for-by 1)"`, it will take a long time to finish. If you do have more than 256 entries in tables, you will need to specify a new limit with this option.

5.15 timeout option (-t)

This is the “time limit for servers” option.

This option has some usage constraints. It:

- must be compiled in by defining `SHELL_ENABLED` during the compilation.

AutoGen works with a shell server process. Most normal commands will complete in less than 10 seconds. If, however, your commands need more time than this, use this option.

The valid range is 0 to 3600 seconds (1 hour). Zero will disable the server time limit.

5.16 trace option

This is the “tracing level of detail” option. This option will cause AutoGen to display a trace of its template processing. There are five levels:

‘`nothing`’	Does no tracing at all (default)
‘`templates`’	Traces the invocation of `DEFINE`d macros and `INCLUDE`s
‘`block-macros`’	Traces all block macros. The above, plus `IF`, `FOR`, `CASE` and `WHILE`.
‘`expressions`’	Displays the results of expression evaluations
‘`everything`’	Displays the invocation of every AutoGen macro, even `TEXT` macros.

5.17 trace-out option

This is the “tracing output file or filter” option. The output specified may be either a file name, or, if the option argument begins with the `pipe` operator (`|`), a command that will receive the tracing output as standard in. For example, `--traceout='| less'` will run the trace output through the `less` program.

5.18 show-defs option

This is the "show the definition tree" option.

This option has some usage constraints. It:

- must be compiled in by defining `DEBUG` during the compilation.
- may not be preset with environment variables or in initialization (rc) files.

This will print out the complete definition tree before processing the template.

5.19 show-shell option

This is the "show shell commands" option.

This option has some usage constraints. It:

- must be compiled in by defining `DEBUG` during the compilation.
- may not be preset with environment variables or in initialization (rc) files.

This will cause `set -x` to be executed in the shell, with the resultant output printed to /dev/tty. This option will have no effect if '`--disable-shell`' was specified at configure time.

5.20 define option (-D)

This is the "name to add to definition list" option.

This option has some usage constraints. It:

- may appear an unlimited number of times.
- is a member of the define class of options.

The AutoGen define names are used for the following purposes:

1. Sections of the AutoGen definitions may be enabled or disabled by using C-style #ifdef and #ifndef directives.
2. When defining a value for a name, you may specify the index for a particular value. That index may be a literal value, a define option or a value #define-d in the definitions themselves.
3. The name of a file may be prefixed with `$NAME/`. The `$NAME` part of the name string will be replaced with the define-d value for `NAME`.
4. When AutoGen is finished loading the definitions, the defined values are exported to the environment with, `putenv(3)`. These values can then be used in shell scripts with `${NAME}` references and in templates with `(getenv "NAME")`.
5. While processing a template, you may specify an index to retrieve a specific value. That index may also be a define-d value.

5.21 undefine option (-U)

This is the “definition list removal pattern” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.
- may not be preset with environment variables or in initialization (rc) files.
- is a member of the define class of options.

Just like ’C’, AutoGen uses `#ifdef/#ifndef` preprocessing directives. This option will cause the matching names to be removed from the list of defined values.

6 Configuring and Installing

6.1 Configuring AutoGen

AutoGen is configured and built using Libtool, Automake and Autoconf. Consequently, you can install it whereever you wish using the various ‘`--prefix`’ options. To the various configuration options supplied by these tools, AutoGen adds a few of its own:

‘`--disable-shell`’

AutoGen is now capable of acting as a CGI forms server, See Section 6.2 [AutoGen CGI], page 63. As such, it will gather its definitions using either ‘`GET`’ or ‘`POST`’ methods. All you need to do is have a template named ‘`cgi.tpl`’ handy or specify a different one with a command line option.

However, doing this without disabling the server shell brings considerable risk. If you were to pass user input to a script that contained, say, the classic `"‘‘rm -rf /‘’"`, you might have a problem. This configuration option will cause shell template commands to simply return the command string as the result. No mistakes. Much safer. Strongly recommended. The default is to have server shell scripting enabled.

Disabling the shell will have some build side effects, too.

- Many of the make check tests will fail, since they assume a working server shell.
- The getdefs and columns programs are not built. The options are distributed as definition files and they cannot be expanded with a shell-disabled AutoGen.
- Similarly, the documentation cannot be regenerated because the documentation templates depend on subshell functionality.

‘`--enable-debug`’

Turning on AutoGen debugging enables very detailed inspection of the input definitions and monitoring shell script processing. These options are not particularly useful to anyone not directly involved in maintaining AutoGen. If you do choose to enable AutoGen debugging, be aware that the usage page was generated without these options, so when the build process reaches the documentation rebuild, there will be a failure. ‘`cd`’ into the ‘`agen5`’ build directory, ‘`make`’ the ‘`autogen.texi`’ file and all will be well thereafter.

‘`--with-regex-header`’
‘`--with-header-path`’
‘`--with-regex-lib`’

These three work together to specify how to compile with and link to a particular POSIX regular expression library. The value for ‘`--with-regex-header=value`’ must be the name of the relevant header file. The AutoGen sources will attempt to include that source with a `#include <value>` C preprocessing statement. The `path` from the ‘`--with-header-path=path`’ will be added to `CPPFLAGS` as ‘`-Ipath`’. The `lib-specs` from '`--with-regex-lib=lib-specs`' will be added to `LDFLAGS` without any adornment.

6.2 AutoGen as a CGI server

AutoGen is now capable of acting as a CGI forms server. It behaves as a CGI server if the definitions input is from stdin and the environment variable `REQUEST_METHOD` is defined and set to either "GET" or "POST". If set to anything else, AutoGen will exit with a failure message. When set to one of those values, the CGI data will be converted to AutoGen definitions (see Chapter 2 [Definitions File], page 6) and the template named `"cgi.tpl"` will be processed.

This works by including the name of the real template to process in the form data and having the `"cgi.tpl"` template include that template for processing. I do this for processing the form `http://autogen.sourceforge.net/conftest.html`. The `"cgi.tpl"` looks approximately like this:

```
<? AutoGen5 Template ?>
<?
IF (not (exist? "template"))                       ?><?
  form-error                                       ?><?

ELIF (=* (get "template") "/")                     ?><?
  form-error                                       ?><?

ELIF (define tpl-file (string-append "cgi-tpl/"
                      (get "template")))
     (access? tpl-file R_OK)                       ?><?
  INCLUDE (. tpl-file)                             ?><?

ELIF (set! tpl-file (string-append tpl-file ".tpl"))
     (access? tpl-file R_OK)                       ?><?
  INCLUDE (. tpl-file)                             ?><?

ELSE                                               ?><?
  form-error                                       ?><?
ENDIF                                              ?>
```

This forces the template to be found in the `"cgi-tpl/"` directory. Note also that there is no suffix specified in the pseudo macro (see Section 3.1 [pseudo macro], page 17). That tells AutoGen to emit the output to stdout.

The output is actually spooled until it is complete so that, in the case of an error, the output can be discarded and a proper error message can be written in its stead.

Please also note that it is advisable, *especially* for network accessible machines, to configure AutoGen (see Section 6.1 [configuring], page 61) with shell processing disabled (`--disable-shell`). That will make it impossible for any referenced template to hand data to a subshell for interpretation.

6.3 Signal Names

When AutoGen is first built, it tries to use `psignal(3)`, `sys_siglist`, `strsigno(3)` and `strsignal(3)` from the host operating system. If your system does not supply these, the AutoGen distribution will. However, it will use the distributed mapping and this mapping is unlikely to match what your system uses. This can be fixed. Once you have installed autogen, the mapping can be rebuilt on the host operating system. To do so, you must perform the following steps:

1. Build and install AutoGen in a place where it will be found in your search path.
2. `cd ${top_srcdir}/compat`
3. `autogen strsignal.def`
4. Verify the results by examining the '`strsignal.h`' file produced.
5. Re-build and re-install AutoGen.

If you have any problems or peculiarities that cause this process to fail on your platform, please send me copies of the header files containing the signal names and numbers, along with the full path names of these files. I will endeavor to fix it. There is a shell script inside of '`strsignal.def`' that tries to hunt down the information.

6.4 Installing AutoGen

There are several files that get installed. The number depend whether or not both shared and archive libraries are to be installed. The following assumes that everything is installed relative to `$prefix`. You can, of course, use `configure` to place these files where you wish.

NB AutoGen does not contain any compiled-in path names. All support directories are located via option processing, the environment variable `HOME` or finding the directory where the executable came from.

The installed files are:

1. The executables in '`bin`' (autogen, getdefs and columns).
2. The AutoOpts link libraries as '`lib/libopts.*`'.
3. An include file in '`include/options.h`', needed for Automated Option Processing (see next chapter).
4. Several template files and a scheme script in '`share/autogen`', needed for Automated Option Processing (see Chapter 7 [AutoOpts], page 66), parsing definitions written with scheme syntax (see Section 2.4 [Dynamic Text], page 10), the templates for producing documentation for your program (see Section 7.3.5 [documentation attributes], page 81), autoconf test macros, and AutoFSM.
5. Info-style help files as '`info/autogen.info*`'. These files document AutoGen, the option processing library AutoOpts, and several add-on components.
6. The three man pages for the three executables are installed in man/man1.

This program, library and supporting files can be installed with three commands:

- <src-dir>/configure [<configure-options>]
- make
- make install

However, you may wish to insert `make check` before the `make install` command.

If you do perform a `make check` and there are any failures, you will find the results in `<module>/test/FAILURES`. Needless to say, I would be interested in seeing the contents of those files and any associated messages. If you choose to go on and analyze one of these failures, you will need to invoke the test scripts individually. You may do so by specifying the test (or list of test) in the TESTS make variable, thus:

```
gmake TESTS=test-name.test check
```

I specify `gmake` because most makes will not let you override internal definitions with command line arguments. `gmake` does.

All of the AutoGen tests are written to honor the contents of the `VERBOSE` environment variable. Normally, any commentary generated during a test run is discarded unless the `VERBOSE` environment variable is set. So, to see what is happening during the test, you might invoke the following with *bash* or *ksh*:

```
VERBOSE=1 gmake TESTS="for.test forcomma.test" check
```

Or equivalently with *csh*:

```
env VERBOSE=1 gmake TESTS="for.test forcomma.test" check
```

7 Automated Option Processing

AutoOpts 19.0 is bundled with AutoGen. It is a tool that virtually eliminates the hassle of processing options and keeping man pages, info docs and usage text up to date. This package allows you to specify several program attributes, up to a hundred option types and many option attributes. From this, it then produces all the code necessary to parse and handle the command line and initialization file options, and the documentation that should go with your program as well.

AutoOpts is distributed under The GNU Lesser General Public License. "Lesser" meaning you have greater license with it and may link it into commercial programs.

7.1 AutoOpts Features

AutoOpts supports option processing; option state saving; and program documentation with innumerable features. Here, we list a few obvious ones and some important ones, but the full list is really defined by all the attributes defined in the Section 7.3 [Option Definitions], page 70 section.

1. POSIX-compliant short (flag) option processing.
2. GNU-style long options processing. Long options are recognized without case sensitivity, and they may be abbreviated.
3. Environment variable initializations.
4. Initialization from RC or INI files, and saving the option state back into one.
5. Options may be marked as *dis*`-abled` with a disablement prefix. Such options may default to either an enabled or a disabled state. You may also provide an enablement prifix, too, e.g., `--allow-mumble` and `--prevent-mumble`.
6. Verify that required options are present between the minimum and maximum number of times on the command line.
7. Verify that conflicting options do not appear together, and that options that require the presence of other options are, in fact, used in the presence of other options.
8. Provides a callable routine to parse a text string as if it were from one of the RC/INI files.
9. `--help` and `--version` are automatically supported. `--more-help` will page the generated help.
10. By adding a ‘`doc`’ and ‘`arg-name`’ attributes to each option, AutoGen will also be able to produce a man page and the ‘`invoking`’ section of a texinfo document.
11. Insert the option processing state into Scheme-defined variables. Thus, Guile based applications that are linked with private `main()` routines can take advantage of all of AutoOpts’ functionality.
12. If `test-main` is defined, the output ‘`.c`’ file will contain a main routine that will be compiled when `TEST_<prog-name>_OPTS` is defined. See Section 7.7 [shell options], page 95. If you choose to compile this program, it is currently capable of producing one of three results:
    a. A program that processes the arguments and writes to standard out portable shell commands containing the digested options.

   b. A program that will generate portable shell commands to parse the defined options. The expectation is that this result will be copied into a shell script and used there.
   c. `test-main` may specify a routine that will be called with a pointer to the option descriptions as the single argument. You must supply this routine and, obviously, you can cause it to do whatever you wish it to do.
13. Library suppliers can specify command line options that their client programs will accept. They specify option definitions that get `#include`-d into the client option definitions and they specify an `"anchor"` option that has a callback and must be invoked. That will give the library access to the option state for their options.
14. The generated usage text can be emitted in either AutoOpts standard format (maximizing the information about each option), or GNU-ish normal form. The default form is selected by either specifying or not specifying the `gnu-usage` attribute (see Section 7.3.3 [information attributes], page 73). This can be overridden by the user himself with the `AUTOOPTS_USAGE` environment variable. If it exists and is set to the string `gnu`, it will force GNU-ish style format; if it is set to the string `autoopts`, it will force AutoOpts standard format; otherwise, it will have no effect.

Explanatory details:

loading rc files

Initialization files are enabled by specifying the program attribute `homerc` (see Section 7.3.1 [program attributes], page 71). The initialization values are identified by the long option name followed by white space and any associated value. The value, if any, will continue through the end of the last line not continued with a backslash. Leading and trailing white space is stripped.

Initialization files are selected both by the `homerc` entries and, optionally, via an automatically supplied command line option. The first component of the `homerc` entry may be an environment variable such as `$HOME`, or it may also be `$$` (**two** dollar sign characters) to specify the directory of the executable. For example:

```
homerc = "$$/../share/autogen";
```

will cause the AutoOpts library to look in the normal autogen datadir for an initialization file.

The initialization files are processed in the order they are specified by the `homerc` attribute, so that each new file will normally override the settings of the previous files. A few options may be marked for `immediate action` (see Section 7.3.4.4 [Immediate Action], page 77). Any such options are acted upon in **reverse** order. The disabled `load-opts` (`--no-load-opts`) option, for example, is an immediate action option. Its presence in the last `homerc` file will prevent the processing of any prior `homerc` files.

Further initialization file processing can be **suppressed** by specifying `--no-load-opts` on the command line, or `PROGRAM_LOAD_OPTS=no` in the environment, or `no-load-opts` in any of the specified `homerc` files.

saving rc files

When initialization files are enabled for an application, the user is also provided with an automatically supplied `--save-opts` option. All of the known option state will be written to either the specified output file or, if it is not specified, then to the last specified `homerc` file.

process a text string for options

The `optionLoadLine` takes two arguments:

1. The pointer to the option descriptor.
2. A pointer to a NUL-terminated string that contains a long option name followed, optionally, by a string value.

Leading and trailing white space is trimmed from the value, but otherwise new lines are not discarded. The caller is expected to have NUL terminated the string at the correct point.

7.2 Quick Start

Since it is generally easier to start with a simple example than it is to look at the options that AutoGen uses itself, here is a very simple AutoOpts example. You can copy this example out of the Info file and into a source file to try it. You can then embellish it into what you really need. For more extensive examples, you can also examine the help output and option definitions for the commands `columns`, `getdefs` and `autogen` itself.

For our simple example, assume you have a program named `check` that takes two options:

1. A list of directories to check over for whatever it is `check` does. You want this option available as a POSIX-style flag option and a GNU long option. You want to allow as many of these as the user wishes.
2. An option to show or not show the definition tree being used. Only one occurrence is to be allowed, specifying one or the other.

First, specify your program attributes and its options to AutoOpts, as with the following example.

```
AutoGen Definitions options;
prog-name     = check;
prog-title    = "Checkout Automated Options";
long-opts;
test_main;

flag = {
    name      = check_dirs;
    value     = L;        /* flag style option character */
    arg_type  = string;   /* option argument indication  */
    max       = NOLIMIT;  /* occurrence limit (none)     */
    stack_arg;            /* save opt args in a stack    */
    descrip   = "Checkout directory list";
};

flag = {
    name      = show_defs;
    descrip   = "Show the definition tree";
    disable   = dont;     /* mark as enable/disable type */
                          /* option.  Disable as ‘dont-’ */
};
```

Then perform the following steps:

1. `autogen checkopt.def`
2. `cc -o check -DTEST_CHECK_OPTS -g checkopt.c -L $prefix/lib -lopts`

And now, `./check --help` yields:

```
check - Checkout Automated Options
USAGE:  check [-<flag> [<val>]]... [--<name>[{=| }<val>]]...
  Flg Arg Option-Name    Description
   -L YES check-dirs     Checkout directory list
                                - may appear without limit
      no  show-defs      Show the definition tree
```

```
                                   - disabled as --dont-show-defs
   -? no  help           Display usage information and exit
   -! no  more-help      Extended usage information passed thru pager

Options may be specified by doubled markers and their name
or by a single marker and the flag character/option value.
```

Normally, however, you would compile '`checkopt.c`' as in:

```
cc -o checkopt.o -I$prefix/include -c checkopt.c
```

and link '`checkopt.o`' with the rest of your program. The main program causes the options to be processed by calling `optionProcess` (see Section 7.4.28.3 [libopts-optionProcess], page 91):

```
main( int argc, char** argv )
{
  {
    int optct = optionProcess( &checkOptions, argc, argv );
    argc -= optct;
    argv += optct;
  }
```

The options are tested and used as in the following fragment:

```
if (HAVE_OPT( SHOW_DEFS ) && HAVE_OPT( CHECK_DIRS )) {
  int    dirct = STACKCT_OPT( CHECK_DIRS );
  char** dirs  = STACKLST_OPT( CHECK_DIRS );
  while (dirct-- > 0) {
    char* dir = *dirs++;
    ...
```

A lot of magic happens to make this happen. The rest of this chapter will describe the myriad of option attributes supported by AutoOpts. However, keep in mind that, in general, you won't need much more than what was described in this "quick start" section.

7.3 Option Definitions

AutoOpts uses an AutoGen definitions file for the definitions of the program options and overall configuration attributes. The complete list of program and option attributes is quite extensive, so if you are reading to understand how to use AutoOpts, I recommend reading the "Quick Start" section (see Section 7.2 [Quick Start], page 69) and paying attention to the following:

1. `prog-name`, `prog-title`, and `argument`, program attributes, See Section 7.3.1 [program attributes], page 71.
2. `name` and `descrip` option attributes, See Section 7.3.4.1 [Required Attributes], page 75.
3. `value` (flag character) and `min` (occurrence counts) option attributes, See Section 7.3.4.2 [Common Attributes], page 75.
4. `arg-type` from the option argument specification section, See Section 7.3.4.6 [Option Arguments], page 78.
5. Read the overall how to, See Section 7.6 [Using AutoOpts], page 94.

6. Highly recommended, but not required, are the several `"man"` and `"info"` documentation attributes, See Section 7.3.5 [documentation attributes], page 81.

Keep in mind that the majority are rarely used and can be safely ignored. However, when you have special option processing requirements, the flexibility is there.

7.3.1 Program Description Attributes

The following global definitions are used to define attributes of the entire program. These generally alter the configuration or global behavior of the AutoOpts option parser. The first two are required of every program. The rest have been alphabetized. Except as noted, there may be only one copy of each of these definitions:

‘`prog-name`’

This attribute is required. Variable names derived from this name are derived using `string->c_name!` (see Section 3.5.47 [SCM string->c-name!], page 42).

‘`prog-title`’

This attribute is required and may be any descriptive text.

‘`allow-errors`’

The presence of this attribute indicates ignoring any command line option errors. This may also be turned on and off by invoking the macros `ERRSKIP_OPTERR` and `ERRSTOP_OPTERR` from the generated interface file.

‘`argument`’

Specifies the syntax of the arguments that **follow** the options. It may not be empty, but if it is not supplied, then option processing must consume all the arguments. If it is supplied and starts with an open bracket (`[`), then there is no requirement on the presence or absence of command line argument following the options. Lastly, if it is supplied and does not start with an open bracket, then option processing must **not** consume all of the command line arguments.

‘`environrc`’

Indicates looking in the environment for values of variables named, `PROGRAM_OPTNAME`, where `PROGRAM` is the upper cased `C-name` of the program and `OPTNAME` is the upper cased `C-name` of the option. The `C-names` are the regular names with all special characters converted to underscores (`_`).

If a particular option may be disabled, then its disabled state is indicated by setting the value to the disablement prefix. So, for example, if the disablement prefix were `dont`, then you can disable the `optname` option by setting the `PROGRAM_OPTNAME`’ environment variable to ‘*dont*’. See Section 7.3.4.2 [Common Attributes], page 75.

‘`export`’

This string is inserted into the .h interface file. Generally used for global variables or `#include` directives required by `flag_code` text and shared with other program text. Do not specify your configuration header (‘`config.h`’) in this attribute or the `include` attribute, however. Instead, use `config-header`, below.

‘`config-header`’
The contents of this attribute should be just the name of the configuration file. A "#include" naming this file will be inserted at the top of the generated header.

‘`homerc`’ Specifies either a directory or a file using a specific path (like `.` or ‘`/usr/local/share/progname`’) or an environment variable (like ‘`$HOME/rc/`’ or ‘`$PREFIX/share/progname`’) or the directory where the executable was found (‘`$$[/...]`’) to use to try to find the rcfile. Use as many as you like. The presence of this attribute activates the `--save-opts` and `--load-opts` options. See [loading rc files], page 67.

‘`include`’ This string is inserted into the .c file. Generally used for global variables required only by `flag_code` program text.

‘`long-opts`’
Presence indicates GNU-standard long option processing. If any options do not have an option value (flag character) specified, and least one does specify such a value, then you must specify `long-opts`. If none of your options specify an option value (flag character) and you do not specify `long-opts`, then command line arguments are processed in "named option mode". This means that:

- Every command line argument must be a long option.
- The flag markers `-` and `--` are completely optional.
- The `argument` program attribute is disallowed.
- One of the options may be specified as the default (as long as it has a required option argument).

‘`prefix`’ This value is inserted into **all** global names. This will disambiguate them if more than one set of options are to be compiled into a single program.

‘`rcfile`’ Specifies the initialization file name. This is only useful if you have provided at least one `homerc` attribute. default: .<prog-name>rc

‘`version`’ Specifies the program version and activates the VERSION option, See Section 7.3.6 [automatic options], page 82.

7.3.2 Generated main Procedures

AutoOpts can generate the main procedure in certain circumstances. It will do this to help with integrating with the guile library environment, and for creating a program to convert command line options into environment variables for use in processing shell script options and for testing the command line interface.

‘`before-guile-boot`’
If `guile-main` has been specified and if this is specified as well, then this code will be inserted into the actual `main()` procedure before `gh_enter()` is called.

‘`guile-main`’
Creates a guile-style main and inner-main procedures. The inner main procedure will call `optionProcess()` and will invoke any code specified by this attribute. If this attribute does not specify any code, then calls to the AutoOpts

library procedure `export_options_to_guile()` and then `scm_shell()` will be inserted into `inner_main()`.

‘`main-text`’

If you need to specify the content of the main procedure generated for the "option testing" program, you can do that with this attribute. The result will be a procedure that looks like this:

```
int main( int argc, char** argv ) {
[+  main-text  +]
    return EXIT_SUCCESS;
}
```

‘`test-main`’

Creates a test main procedure for testing option processing. The resulting program may also be used for several purposes.

1. If the text of `test-main` is short (3 or fewer characters), the generated main() will call putBourneShell. That routine will emit Bourne shell commands that can be eval-ed by a Bourne-derived shell to incorporate the digested options into the shell's environment, See Section 7.7 [shell options], page 95. You would use it thus:

   ```
   eval "‘./programopts $@‘"
   test -z "${OPTION_CT}" ] && exit 1
   test ${OPTION_CT} -gt 0 && shift ${OPTION_CT}
   ```

2. If the text of `test-main` contains `putShellParse`, the program will generate portable Bourne shell commands that will parse the command line options. The expectation is that this result will be copied into a shell script and used there, See Section 7.7 [shell options], page 95.
3. Any other text must be the name of a routine that you will write yourself. That routine will be called after processing the command line options and it will be passed the option processing descriptor pointer as its sole argument.

7.3.3 Program Information Attributes

These attributes are used to define how and what information is displayed to the user of the program.

‘`copyright`’

The `copyright` is a structured value containing three to five values. If `copyright` is used, then the first three are required.

1. ‘`date`’ - the list of applicable dates for the copyright.
2. ‘`owner`’ - the name of the copyright holder.
3. ‘`type`’ - specifies the type of distribution license. AutoOpts/AutoGen will automatically support the text of the GNU Public License (‘`GPL`’), the GNU General Public License with Library extensions (‘`LGPL`’), the Free BSD license (‘`BSD`’), and a write-it-yourself copyright notice (‘`NOTE`’). Only these values are recognized.

4. ‘`text`’ - the text of the copyright notice. It is only needed if ‘`type`’ is set to ‘`NOTE`’.
5. ‘`author`’ - in case the author name is to appear in the documentation and is different from the copyright owner.
6. ‘`eaddr`’ - email address of the author or copyright holder.

An example of this might be:

```
copyright = {
    date  = "1992-2003";
    owner = "Bruce Korb";
    eaddr = 'bkorb@gnu.org';
    type  = GPL;
};
```

‘`detail`’ This string is added to the usage output when the HELP option is selected.

‘`explain`’ Gives additional information whenever the usage routine is invoked..

‘`preserve-case`’

This attribute will not change anything except appearance. Normally, the option names are all documented in lower case. However, if you specify this attribute, then they will display in the case used in their specification. Command line options will still be matched without case sensitivity.

‘`prog-desc` **and**’
‘`opts-ptr`’

These define global pointer variables that point to the program descriptor and the first option descriptor for a library option. This is intended for use by certain libraries that need command line and/or initialization file option processing. These definitions have no effect on the option template output, but are used for creating a library interface file. Normally, the first "option" for a library will be a documentation option that cannot be specified on the command line, but is marked as `settable`. The library client program will invoke the `SET_OPTION` macro which will invoke a handler function that will finally set these global variables.

‘`rcsection`’

If you have a collection of option descriptions that are intended to use the same RC/ini files, then you will likely want to partition that file. That will be possible by specifying this attribute with AutoOpts version 9.2 and later.

Every RC file will be considered partitioned by lines that commence with the square open bracket (`[`). All text before such a line is always processed. Once such a line is found, the **upper-cased** c-variable-syntax program name will be compared against the text following that bracket. If there is a match and the next character after that is a square close bracket (`]`), then the section is processed and the file closed. Otherwise, the section is ignored and a matching section is searched for.

For exampe, if the `fumble-stumble` options had a sectioned RC file, then a line containing `[FUMBLE_STUMBLE]` would be searched for.

'`usage`' Optionally names the usage procedure, if the library routine `optionUsage()` does not work for you. If you specify `gnu_usage` as the value of this attribute, for example, you will use a procedure by that name for displaying usage. Of course, you will need to provide that procedure.

'`gnu-usage`'
Normally, the default format produced by the `optionUsage` procedure is *AutoOpts Standard*. By specifying this attribute, the default format will be *GNU-ish style*. Either default may be overridden by the user with the `AUTOOPTS_USAGE` environment variable. If it is set to `gnu` or `autoopts`, it will alter the style appropriately. This attribute will conflict with the `usage` attribute.

7.3.4 Option Attributes

For each option you wish to specify, you must have a block macro named `flag` defined. There are two required attributes: `name` and `descrip`. If any options do not have a `value` (traditional flag character) attribute, then the `long-opts` global text macro must also be defined. As a special exception, if no options have a `value` **and** `long-opts` is not defined **and** `argument` is not defined, then all arguments to the program are named. In this case, the `-` and `--` command line option markers are optional.

7.3.4.1 Required Attributes

Every option must have exactly one copy of both of these attributes.

'`name`' Long name for the option. Even if you are not accepting long options and are only accepting flags, it must be provided. AutoOpts generates private, named storage that requires this name.

'`descrip`' Except for documentation options, a **very** brief description of the option. About 40 characters on one line, maximum. It appears on the `usage()` output next to the option name. If, however, the option is a documentation option, it will appear on one or more lines by itself. It is thus used to visually separate and comment upon groups of options in the usage text.

7.3.4.2 Common Option Attributes

These option attributes are optional. Any that do appear in the definition of a flag, may appear only once.

'`value`' The flag character to specify for traditional option flags. e.g. `-L`.

'`max`' Maximum occurrence count (invalid if *disable* present).

'`min`' Minimum occurrence count. If present, then the option **must** appear on the command line. Preset values do not count towards the minimum occurrence count.

'`must-set`'
If an option must be specified, but it need not be specified on the command line, then specify this attribute for the option.

‘`enable`’ Long-name prefix for enabling the option (invalid if *disable* **not** present). Only useful if long option names are being processed.

‘`disable`’ Prefix for disabling (inverting sense of) the option. Only useful if long option names are being processed.

‘`enabled`’ If default is for option being enabled. (Otherwise, the OPTST_DISABLED bit is set at compile time.) Only useful if the option can be disabled.

‘`ifdef`’

‘`ifndef`’ If an option is relevant on certain platforms or when certain features are enabled or disabled, you can specify the compile time flag used to indicate when the option should be compiled in or out. For example, if you have a configurable feature, `mumble` that is indicated with the compile time define, `WITH_MUMBLING`, then add:

```
ifdef = WITH_MUMBLING;
```

Note that case and spelling must match whatever you are using. Do not confuse these attributes with the AutoGen directives of the same names, See Section 2.5 [Directives], page 10. These cause C pre-processing directives to be inserted into the generated C text.

7.3.4.3 Special Option Handling

These option attributes do not fit well with other categories.

‘`no-preset`’

If presetting this option is not allowed. (Thus, environment variables and values set in rc/ini files will be ignored.)

‘`settable`’

If the option can be set outside of option processing. If this attribute is defined, special macros for setting this particular option will be inserted into the interface file. For example, `TEMPL_DIRS` is a settable option for AutoGen, so a macro named `SET_OPT_TEMPL_DIRS(a)` appears in the interface file. This attribute interacts with the *documentation* attribute.

‘`equivalence`’

Generally, when several options are mutually exclusive and basically serve the purpose of selecting one of several processing modes, these options can be considered an equivalence class. Sometimes, it is just easier to deal with them as such.

For an option equivalence class, there is a single occurrence counter for all the members of the class. All members of the equivalence class must contain the same equivalenced-to option, including the equivalenced-to option itself. Thus, it must be a member.

As an example, `cpio(1)` has three options `-o`, `-i`, and `-p` that define the operational mode of the program (`create`, `extract` and `pass-through`, respectively). They form an equivalence class from which one and only one member must appear on the command line. If `cpio` were an AutoOpt-ed program, then each of these option definitions would contain:

```
equivalence = create;
```

and the program would be able to determine the operating mode with code that worked something like this:

```
switch (WHICH_IDX_CREATE) {
case INDEX_OPT_CREATE:       ...
case INDEX_OPT_EXTRACT:      ...
case INDEX_OPT_PASS_THROUGH: ...
default:    /* cannot happen */
}
```

‘`documentation`’

This attribute means the option exists for the purpose of separating option description text in the usage output. Libraries may choose to make it settable so that the library can determine which command line option is the first one that pertains to the library.

If present, this option disables all other attributes except `settable`, `call_proc` and `flag_code`. `settable` must be and is only specified if `call_proc` or `flag_code` has been specified. When present, the `descrip` attribute will be displayed only when the `--help` option has been specified. It will be displayed flush to the left hand margin and may consist of one or more lines of text. The name of the option will not be printed.

Documentation options are for clarifying the usage text and will not appear in generated man pages or in the generated invoking texinfo doc.

7.3.4.4 Immediate Action Attributes

Certain options may need to be processed early. For example, in order to suppress the processing of RC files, it is necessary to process the command line option `--no-load-opts` **before** the RC files are processed. To accommodate this, certain options may have their enabled or disabled forms marked for immediate processing. The consequence of this is that they are processed ahead of all other options in the reverse of normal order.

Normally, the first options processed are the options specified in the first `homerc` file, followed by then next `homerc` file through to the end of RC file processing. Next, environment variables are processed and finally, the command line options. The later options override settings processed earlier. That actually gives them higher priority. Command line immediate action options actually have the lowest priority of all. They would be used only if they are to have an effect on the processing of subsequent options.

‘`immediate`’

Use this option attribute to specify that the enabled form of the option is to be processed immediately. The `help` and `more-help` options are so specified. They will also call `exit()` upon completion, so they **do** have an effect on the processing of the remaining options :-).

‘`immed-disable`’

Use this option attribute to specify that the disabled form of the option is to be processed immediately. The `load-opts` option is so specified. The `--no-load-opts` command line option will suppress the processing of RC files and environment variables. Contrariwise, the `--load-opts` command line option is processed normally. That means that the options specified in that file will be processed after all the `homerc` files and, in fact, after options that precede it on the command line.

7.3.4.5 Option Conflict Attributes

These attributes may be used as many times as you need. They are used at the end of the option processing to verify that the context within which each option is found does not conflict with the presence or absence of other options.

This is not a complete cover of all possible conflicts and requirements, but it simple to implement and covers the more common situations.

‘`flags-must`’
one entry for every option that **must** be present when this option is present

‘`flags-cant`’
one entry for every option that **cannot** be present when this option is present

7.3.4.6 Option Argument Specification

Several attributes relate to the handling of arguments to options. Each may appear only once, except for the `arg-range` attribute. It may appear as often as may be needed.

‘`arg-type`’
This specifies the type of argument the option will take. If not present, the option cannot take an argument. If present, it must be one of the following four. The bracketed part of the name is optional.

‘`str[`*ing*`]`’ The argument may be any arbitrary string.

‘`num[`*ber*`]`’ The argument must be a correctly formed integer, without any trailing U’s or L’s. If you specify your own callback validation routine, read the `arg-range` section below for some considerations. AutoOpts contains a library procedure to convert the string to a number. If you specify range checking with `arg-range`, then AutoOpts produces a special purpose procedure for this option.

‘`bool[`*ean*`]`’
The argument will be interpreted and always yield either AG_TRUE or AG_FALSE. False values are the empty string, the number zero, or a string that starts with `f`, `F`, `n` or `N` (representing False or No). Anything else will be interpreted as True.

‘`key[`*word*`]`’
The argument must match a specified list of strings. Assuming you have named the option, `optn-name`, the strings will be converted into an enumeration of type `te_Optn_Name` with the values `OPTN_NAME_KEYWORD`. If you have **not** specified a default value, the value `OPTN_NAME_UNDEFINED` will be inserted with the value zero. The

option will be initialized to that value. You may now use this in your code as follows:

```
te_Optn_Name opt = OPT_VALUE_OPTN_NAME;
switch (opt) {
case OPTN_NAME_UNDEFINED:  /* undefined things */ break;
case OPTN_NAME_KEYWORD:    /* ‘keyword’ things */ break;
default: /* utterly impossible */ ;
}
```

AutoOpts produces a special purpose procedure for this option.

‘`keyword`’ If the `arg-type` is `keyword`, then you must specify the list of keywords by a series of `keyword` entries. The interface file will contain an enumeration of `<`*OPTN_NAME*`>_<`*KEYWORD*`>` for each keyword entry.

‘`arg-optional`’

This attribute indicates that the user does not have to supply an argument for the option. This is only valid if the *arg-type* is `string` or `keyword`. If it is `keyword`, then this attribute may also specify the default keyword to assume when the argument is not supplied. If left empty, *arg-default* or the zero-valued keyword will be used.

‘`arg-default`’

This specifies the default value to be used when the option is not specified or preset.

‘`default`’ If your program processes its arguments in named option mode (See “long-opts” in Section 7.3.1 [program attributes], page 71), then you may select **one** of your options to be the default option. Do so with this attribute. The option so specified must have an `arg-type` specified, but not the `arg-optional` attribute. That is to say, the option argument must be required.

If you have done this, then any arguments that do not match an option name and do not contain an equal sign (`=`) will be interpreted as an option argument to the default option.

‘`arg-range`’

If the `arg-type` is `number`, then `arg-range`s may be specified, too. If you specify one or more of these option attributes, then AutoOpts will create a callback procedure for handling it. The argument value supplied for the option must match one of the range entries. Each arg-range should consist of either an integer by itself or an integer range. The integer range is specified by one or two integers separated by the two character sequence, `->`.

The generated procedure imposes some constraints:

- A number by itself will match that one value.
- The high end of the range may not be `INT_MIN`, both for obvious reasons and because that value is used to indicate a single-valued match.
- An omitted lower value implies a lower bound of INT_MIN.
- An omitted upper value implies a upper bound of INT_MAX.
- The argument value is required. It may not be optional.

The usage procedure displays these ranges by calling the callback with the `pOptDesc` pointer set to `NULL`. Therefore, all callback procedures designed to handle options with numeric arguments **must** be prepared to handle a call with that pointer set `NULL`.

7.3.4.7 Option Argument Handling

AutoOpts will automatically generate a callback procedure for options with enumerated keyword arguments and numeric arguments with range checking. If you have specified such an option, you may not specify any of the attributes listed here.

Otherwise, you may pick zero or one of the following attributes. The first two attributes interact with the `documentation` and `settable` attributes, See Section 7.3.4.3 [Special Option Handling], page 76.

‘`flag-code`’

statements to execute when the option is encountered. The generated procedure will look like this:

```
static void
doOpt<name>( tOptions* pOptions, tOptDesc* pOptDesc )
{
<flag_code>
}
```

Only certain fields within the `tOptions` and `tOptDesc` structures may be accessed. See Section 7.4.1 [Option Processing Data], page 84.

‘`extract-code`’

This is effectively identical to `flag_code`, except that the source is kept in the output file instead of the definitions file. A long comment is used to demarcate the code. You must not modify that marker. *Before* regenerating the option code file, the old file is renamed from MUMBLE.c to MUMBLE.c.save. The template will be looking there for the text to copy into the new output file.

‘`call-proc`’

external procedure to call when option is encountered. The calling sequence must conform to the sequence defined above for the generated procedure, `doOpt<name>`. It has the same restrictions regarding the fields within the structures passed in as arguments. See Section 7.4.1 [Option Processing Data], page 84.

‘`flag-proc`’

Name of another option whose `flag_code` can be executed when this option is encountered.

‘`stack-arg`’

Call a special library routine to stack the option’s arguments. Special macros in the interface file are provided for determining how many of the options were found (`STACKCT_OPT(NAME)`) and to obtain a pointer to a list of pointers to the argument values (`STACKLST_OPT(NAME)`). Obviously, for a stackable argument, the `max` attribute needs to be set higher than `1`.

If two options are equivalenced (see Section 7.3.4.3 [Special Option Handling], page 76) and specify this attribute, then the "equivalenced-to" option will add entries to the stack, and the "equivalencing" option, if specified, will *remove* entries that match the regular expression argument. The pattern, `".*"` will efficiently remove all the entries in the stack. It would not be useful to have more than two options in this equivalence class.

If the stacked option has a disablement prefix, then the entire stack of arguments will be cleared even more efficiently than the `".*"` regular expression. A stacked, equivalencing option with a disablement prefix will cause undefined results.

If all of this is confusing, then don't mess with equivalenced stacked option arguments. If you really want to know, the AutoGen `--define` option (see Section 5.20 [autogen define], page 59) has a stacked argument, and `--undefine` (see Section 5.21 [autogen undefine], page 60) is equivalenced to it.

7.3.5 Man and Info doc Attributes

AutoOpts includes AutoGen templates for producing abbreviated man pages and for producing the invoking section of an info document. To take advantage of these templates, you must add several attributes to your option definitions.

'`doc`' First, every `flag` definition *other than* "documentation" definitions, must have a `doc` attribute defined. If the option takes an argument, then it will need an `arg-name` attribute as well. The `doc` text should be in plain sentences with minimal formatting. The Texinfo commands `@code`, and `@var` will have its enclosed text made into **\fB** entries in the man page, and the `@file` text will be made into **\fI** entries. The `arg-name` attribute is used to display the option's argument in the man page.

Options marked with the "documentation" attribute are for documenting the usage text. All other options should have the "doc" attribute in order to document the usage of the option in the generated man pages.

'`arg-name`'

If an option has an argument, the argument should have a name for documentation purposes. It will default to `arg-type`, but it will likely be clearer with something else like, `file-name` instead of `string` (the type).

'`prog-man-descrip`'
'`prog-info-descrip`'

Then, you need to supply a brief description of what your program does. If you already have a `detail` definition, this may be sufficient. If not, or if you need special formatting for one of the manual formats, then you will need either a definition for `prog-man-descrip` or `prog-info-descrip` or both. These will be inserted verbatim in the man page document and the info document, respectively.

'`man-doc`' Finally, if you need to add man page sections like `SEE ALSO` or `USAGE` or other, put that text in a `man-doc` definition. This text will be inserted verbatim in the man page after the `OPTIONS` section and before the `AUTHOR` section.

7.3.6 Automatically Supported Options

AutoOpts provides automated support for five options. `help` and `more-help` are always provided. `version` is provided if `version` is defined in the option definitions See Section 7.3.1 [program attributes], page 71. `save-opts` and `load-opts` are provided if at least one `homerc` is defined See Section 7.3.1 [program attributes], page 71.

Below are the option names and flag values. The flags are activated if and only if at least one user-defined option also uses a flag value. These flags may be deleted or changed to characters of your choosing by specifying `xxx-value = "y";`, where `xxx` is one of the five names below and `y` is either empty or the character of your choice. For example, to change the help flag from `?` to `h`, specify `help-value = "h";`; and to require that `save-opts` be specified only with its long option name, specify `save-opts-value = "";`.

'`help -?`' This option will immediately invoke the `USAGE()` procedure and display the usage line, a description of each option with its description and option usage information. This is followed by the contents of the definition of the `detail` text macro.

'`more-help -!`'

This option is identical to the `help` option, except that it also includes the contents of the `detail-file` file (if provided and found), plus the output is passed through a pager program. (`more` by default, or the program identified by the `PAGER` environment variable.)

'`version -v`'

This will print the program name, title and version. If it is followed by the letter `c` and a value for `copyright` and `owner` have been provided, then the copyright will be printed, too. If it is followed by the letter `n`, then the full copyright notice (if available) will be printed.

'`save-opts ->`'

This option will cause the option state to be printed in RC/INI file format when option processing is done but not yet verified for consistency. The program will terminate successfully without running when this has completed. Note that for most shells you will have to quote or escape the flag character to restrict special meanings to the shell.

The output file will be the RC/INI file name (default or provided by `rcfile`) in the last directory named in a `homerc` definition.

This option may be set from within your program by invoking the `"SET_OPT_SAVE_OPTS(`*filename*`)"` macro (see Section 7.4.16 [SET_OPT_name], page 87). Invoking this macro will set the file name for saving the option processing state, but the state will **not** actually be saved. You must call `optionSaveFile` to do that (see Section 7.4.28.5 [libopts-optionSaveFile], page 92). **CAVEAT:** if, after invoking this macro, you call `optionProcess`, the option processing state will be saved to this file and `optionProcess` will not return. You may wish to invoke `CLEAR_OPT( SAVE_OPTS )` (see Section 7.4.2 [CLEAR_OPT], page 85) beforehand.

'`load-opts -<`'

This option will load options from the named file. They will be treated exactly as if they were loaded from the normally found RC/INI files, but will not be loaded until the option is actually processed. This can also be used within an RC/INI file causing them to nest.

It is ultimately intended that specifying the option, `no-load-opts` will suppress the processing of rc/ini files and environment variables. To do this, AutoOpts must first implement pre-scanning of the options, environment and rc files.

7.3.7 Library of Standard Options

AutoOpts has developed a set of standardized options. You may incorporate these options in your program simply by *first* adding a `#define` for the options you want, and then the line,

```
#include stdoptions.def
```

in your option definitions. The supported options are specified thus:

```
#define DEBUG
#define DIRECTORY
#define DRY_RUN
#define INPUT
#define INTERACTIVE
#define OUTPUT
#define WARN

#define SILENT
#define QUIET
#define BRIEF
#define VERBOSE
```

By default, only the long form of the option will be available. To specify the short (flag) form, suffix these names with `_FLAG`. e.g.,

```
#define DEBUG_FLAG
```

`--silent`, `--quiet`, `--brief` and `--verbose` are related in that they all indicate some level of diagnostic output. These options are all designed to conflict with each other. Instead of four different options, however, several levels can be incorporated by `#define`-ing `VERBOSE_ENUM`. In conjunction with `VERBOSE`, it incorporates the notion of *5* levels in an enumeration: `silent`, `quiet`, `brief`, `informative` and `verbose`; with the default being `brief`.

7.4 Programmatic Interface

The user interface for access to the argument information is completely defined in the generated header file and in the portions of the distributed file "options.h" that are marked "public".

In the following macros, text marked `<NAME>` or `name` is the name of the option **in upper case** and **segmented with underscores `_`**. The macros and enumerations defined in the options header (interface) file are used as follows:

To see how these `#define` macros are used in a program, the reader is referred to the several '`opts.h`' files included with the AutoGen sources.

7.4.1 Data for Option Processing

This section describes the data that may be accessed from within the option processing callback routines. The following fields may be used in the following ways and may be used for read only. The first set is addressed from the `tOptDesc*` pointer:

'`optIndex`'
'`optValue`'

These may be used by option procedures to determine which option they are working on (in case they handle several options).

'`optActualIndex`'
'`optActualValue`'

These may be used by option procedures to determine which option was used to set the current option. This may be different from the above if the options are members of an equivalence class.

'`optOccCt`'

If AutoOpts is processing command line arguments, then this value will contain the current occurrence count. During the option preset phase (reading rc/ini files and examining environment variables), the value is zero.

'`fOptState`'

The field may be tested for the following bit values (prefix each name with `OPTST_`, e.g. `OPTST_INIT`):

'`INIT`'	Initial compiled value. As a bit test, it will always yield FALSE.
'`SET`'	The option was set via the `SET_OPT()` macro.
'`PRESET`'	The option was set via an RC/INI file, or a call to the `optionLoadLine()` routine.
'`DEFINED`'	The option was set via a command line option.
'`SET_MASK`'	This is a mask of flags that show the set state, one of the above four values.
'`EQUIVALENCE`'	This bit is set when the option was selected by an equivalenced option.
'`DISABLED`'	This bit is set if the option is to be disabled. (Meaning it was a long option prefixed by the disablement prefix, or the option has not been specified yet and initializes as `disabled`.)

As an example of how this might be used, in AutoGen I want to allow template writers to specify that the template output can be left in a writable or read-only state. To support this, there is a Guile function named `set-writable` (see Section 3.4.32 [SCM set-writable], page 28). Also, I provide for command options `--writable` and `--not-writable`. I give precedence to command line and RC file options, thus:

```
switch (STATE_OPT( WRITABLE )) {
case OPTST_DEFINED:
case OPTST_PRESET:
    fprintf( stderr, zOverrideWarn, pCurTemplate->pzFileName,
             pCurMacro->lineNo );
    break;

default:
    if (gh_boolean_p( set ) && (set == SCM_BOOL_F))
        CLEAR_OPT( WRITABLE );
    else
        SET_OPT_WRITABLE;
}
```

'`pzLastArg`'
Pointer to the latest argument string. BEWARE If the argument type is numeric or an enumeration, then this will be the argument **value** and not a pointer to a string.

The following two fields are addressed from the `tOptions*` pointer:

'`pzProgName`'
Points to a NUL-terminated string containing the current program name, as retrieved from the argument vector.

'`pzProgPath`'
Points to a NUL-terminated string containing the full path of the current program, as retrieved from the argument vector. (If available on your system.)

Note these fields get filled in during the first call to `optionProcess()`. All other fields are private, for the exclusive use of AutoOpts code and is subject to change.

7.4.2 CLEAR_OPT(<NAME>) - Clear Option Markings

Make as if the option had never been specified. `HAVE_OPT(<NAME>)` will yield `FALSE` after invoking this macro.

7.4.3 COUNT_OPT(<NAME>) - Definition Count

This macro will tell you how many times the option was specified on the command line. It does not include counts of preset options.

```
if (COUNT_OPT( NAME ) != desired-count) {
    make-an-undesirable-message.
}
```

7.4.4 DESC(<NAME>) - Option Descriptor

This macro is used internally by other AutoOpt macros. It is not for general use. It is used to obtain the option description corresponding to its **UPPER CASED** option name argument. This is primarily used in other macro definitions.

7.4.5 DISABLE_OPT_name - Disable an option

This macro is emitted if it is both settable and it can be disabled. If it cannot be disabled, it may always be CLEAR-ed (see above).

The form of the macro will actually depend on whether the option is equivalenced to another, and/or has an assigned handler procedure. Unlike the `SET_OPT` macro, this macro does not allow an option argument.

```
DISABLE_OPT_NAME;
```

7.4.6 ENABLED_OPT(<NAME>) - Is Option Enabled?

Yields true if the option defaults to disabled and `ISUNUSED_OPT()` would yield true. It also yields true if the option has been specified with a disablement prefix, disablement value or the `DISABLE_OPT_NAME` macro was invoked.

7.4.7 ERRSKIP_OPTERR - Ignore Option Errors

When it is necessary to continue (return to caller) on option errors, invoke this option. It is reversible. See Section 7.4.8 [ERRSTOP_OPTERR], page 86.

7.4.8 ERRSTOP_OPTERR - Stop on Errors

After invoking this macro, if `optionProcess()` encounters an error, it will call `exit(1)` rather than return. This is the default processing mode. It can be overridden by specifying `allow-errors` in the definitions file, or invoking the macro See Section 7.4.7 [ERRSKIP_OPTERR], page 86.

7.4.9 HAVE_OPT(<NAME>) - Have this option?

This macro yields true if the option has been specified in any fashion at all. It is used thus:

```
if (HAVE_OPT( NAME )) {
    <do-things-associated-with-opt-name>;
}
```

7.4.10 ISSEL_OPT(<NAME>) - Is Option Selected?

This macro yields true if the option has been specified either on the command line or via a SET/DISABLE macro.

7.4.11 ISUNUSED_OPT(<NAME>) - Never Specified?

This macro yields true if the option has never been specified, or has been cleared via the `CLEAR_OPT()` macro.

7.4.12 OPTION_CT - Full Count of Options

The full count of all options, both those defined and those generated automatically by AutoOpts. This is primarily used to initialize the program option descriptor structure.

7.4.13 OPT_ARG(<NAME>) - Option Argument String

The option argument value as a pointer to string. Note that argument values that have been specified as numbers are stored as numbers or keywords. For such options, use instead the `OPT_VALUE_name` define. It is used thus:

```
if (HAVE_OPT( NAME )) {
    char* p = OPT_ARG( NAME );
    <do-things-with-opt-name-argument-string>;
}
```

7.4.14 OPT_VALUE_name - Option Argument Value

This macro gets emitted only for options that take numeric or keyword arguments. The macro yields a word-sized integer containing the enumeration or numeric value of the option argument.

```
int opt_val = OPT_VALUE_NAME;
```

7.4.15 RESTART_OPT(n) - Resume Option Processing

If option processing has stopped (either because of an error or something was encountered that looked like a program argument), it can be resumed by providing this macro with the index `n` of the next option to process and calling `optionProcess()` again.

7.4.16 SET_OPT_name - Force an option to be set

This macro gets emitted only when the given option has the `settable` attribute specified.

The form of the macro will actually depend on whether the option is equivalenced to another, has an option argument and/or has an assigned handler procedure. If the option has an argument, then this macro will too. Beware that the argument is not reallocated, so the value must not be on the stack or deallocated in any other way for as long as the value might get referenced.

If you have supplied at least one '`homerc`' file (see Section 7.3.1 [program attributes], page 71), this macro will be emitted for the `--save-opts` option.

```
SET_OPT_SAVE_OPTS( "filename" );
```

See Section 7.3.6 [automatic options], page 82, for a discussion of the implications of using this particular example.

7.4.17 STACKCT_OPT(<NAME>) - Stacked Arg Count

When the option handling attribute is specified as `stack_arg`, this macro may be used to determine how many of them actually got stacked.

Do not use this on options that have not been stacked or has not been specified (the `stack_arg` attribute must have been specified, and `HAVE_OPT(<NAME>)` must yield TRUE). Otherwise, you will likely seg fault.

```
if (HAVE_OPT( NAME )) {
    int     ct = STACKCT_OPT(  NAME );
    char**  pp = STACKLST_OPT( NAME );

    do  {
        char* p = *pp++;
        do-things-with-p;
    } while (--ct > 0);
}
```

7.4.18 STACKLST_OPT(<NAME>) - Argument Stack

The address of the list of pointers to the option arguments. The pointers are ordered by the order in which they were encountered in the option presets and command line processing.

Do not use this on options that have not been stacked or has not been specified (the `stack_arg` attribute must have been specified, and `HAVE_OPT(<OPTION>)` must yield TRUE). Otherwise, you will likely seg fault.

```
if (HAVE_OPT( NAME )) {
    int     ct = STACKCT_OPT(  NAME );
    char**  pp = STACKLST_OPT( NAME );

    do  {
        char* p = *pp++;
        do-things-with-p;
    } while (--ct > 0);
}
```

7.4.19 START_OPT - Restart Option Processing

This is just a shortcut for RESTART_OPT(1) (See Section 7.4.15 [RESTART_OPT], page 87.)

7.4.20 STATE_OPT(<NAME>) - Option State

If you need to know if an option was set because of presetting actions (RC/INI processing or environment variables), versus a command line entry versus one of the SET/DISABLE macros, then use this macro. It will yield one of four values: `OPTST_INIT`, `OPTST_SET`, `OPTST_PRESET` or `OPTST_DEFINED`. It is used thus:

```
switch (STATE_OPT( NAME )) {
    case OPTST_INIT:
        not-preset, set or on the command line.  (unless CLEAR-ed)

    case OPTST_SET:
        option set via the SET_OPT_NAME() macro.
    case OPTST_PRESET:
        option set via an RC/INI file or environment variable

    case OPTST_DEFINED:
        option set via a command line option.

    default:
        cannot happen :)
}
```

7.4.21 USAGE(exit-code) - Usage invocation macro

This macro invokes the procedure registered to display the usage text. Normally, this will be `optionUsage` from the AutoOpts library, but you may select another procedure by specifying `usage = "proc_name"` program attribute. This procedure must take two arguments first, a pointer to the option descriptor, and second the exit code. The macro supplies the option descriptor automatically. This routine is expected to call `exit(3)` with the provided exit code.

The `optionUsage` routine also behaves differently depending on the exit code. If the exit code is zero, it is assumed that assistance has been requested. Consequently, a little more information is provided than when displaying usage and exiting with a non-zero exit code.

7.4.22 VALUE_OPT_name - Option Flag Value

This is a #define for the flag character used to specify an option on the command line. If `value` was not specified for the option, then it is a unique number associated with the option. `option value` refers to this value, `option argument` refers to the (optional) argument to the option.

```
switch (WHICH_OPT_OTHER_OPT) {
case VALUE_OPT_NAME:
    this-option-was-really-opt-name;
case VALUE_OPT_OTHER_OPT:
    this-option-was-really-other-opt;
}
```

7.4.23 VERSION - Version and Full Version

If the `version` attribute is defined for the program, then a stringified version will be #defined as PROGRAM_VERSION and PROGRAM_FULL_VERSION. PROGRAM_FULL_VERSION is used for printing the program version in response to the version option. The version option is automatically supplied in response to this attribute, too.

You may access PROGRAM_VERSION via `programOptions.pzFullVersion`.

7.4.24 WHICH_IDX_name - Which Equivalenced Index

This macro gets emitted only for equivalenced-to options. It is used to obtain the index for the one of the several equivalence class members set the equivalenced-to option.

```
switch (WHICH_IDX_OTHER_OPT) {
case INDEX_OPT_NAME:
    this-option-was-really-opt-name;
case INDEX_OPT_OTHER_OPT:
    this-option-was-really-other-opt;
}
```

7.4.25 WHICH_OPT_name - Which Equivalenced Option

This macro gets emitted only for equivalenced-to options. It is used to obtain the value code for the one of the several equivalence class members set the equivalenced-to option.

```
switch (WHICH_OPT_OTHER_OPT) {
case VALUE_OPT_NAME:
    this-option-was-really-opt-name;
case VALUE_OPT_OTHER_OPT:
    this-option-was-really-other-opt;
}
```

7.4.26 teOptIndex - Option Index and Enumeration

This enum defines the complete set of options, both user specified and automatically provided. This can be used, for example, to distinguish which of the equivalenced options was actually used.

```
switch (pOptDesc->optActualIndex) {
case INDEX_OPT_FIRST:
    stuff;
case INDEX_OPT_DIFFERENT:
    different-stuff;
default:
    unknown-things;
}
```

7.4.27 OPTIONS_STRUCT_VERSION - active version

You will not actually need to reference this value, but you need to be aware that it is there. It is the first value in the option descriptor that you pass to `optionProcess`. It contains a magic number and version information. Normally, you should be able to work with a more recent option library than the one you compiled with. However, if the library is changed incompatibly, then the library will detect the out of date magic marker, explain the difficulty and exit. You will then need to rebuild and recompile your option definitions. This has rarely been necessary.

7.4.28 libopts External Procedures

These are the routines that libopts users may call directly from their code. There are several other routines that can be called by code generated by the libopts option templates, but they are not to be called from any other user code. The ‘`options.h`’ is fairly clear about this, too.

This subsection was automatically generated by AutoGen using extracted information and the aginfo3.tpl template.

7.4.28.1 optionFree

free allocated option processing memory

Usage:

```
optionFree( pOpts );
```

Where the arguments are:

Name	Type	Description
——	——	————-
pOpts	`tOptions*`	program options descriptor

AutoOpts sometimes allocates memory and puts pointers to it in the option state structures. This routine deallocates all such memory.

As long as memory has not been corrupted, this routine is always successful.

7.4.28.2 optionLoadLine

process a string for an option name and value

Usage:

```
optionLoadLine( pOpts, pzLine );
```

Where the arguments are:

Name	Type	Description
——	——	————-
pOpts	`tOptions*`	program options descriptor
pzLine	`const char*`	NUL-terminated text

This is a user callable routine for setting options from, for example, the contents of a file that they read in. Only one option may appear in the text. It will be treated as a normal (non-preset) option.

When passed a pointer to the option struct and a string, it will find the option named by the first token on the string and set the option argument to the remainder of the string. The caller must NUL terminate the string. Any embedded new lines will be included in the option argument.

Invalid options are silently ignored. Invalid option arguments will cause a warning to print, but the function should return.

7.4.28.3 optionProcess

this is the main option processing routine

Usage:

```
int res = optionProcess( pOpts, argc, argv );
```

Where the arguments are:

Name	Type	Description

—–	—–	———–
pOpts	`tOptions*`	program options descriptor
argc	`int`	program arg count
argv	`char**`	program arg vector
returns	int	the count of the arguments processed

This is the main entry point for processing options. It is intended that this procedure be called once at the beginning of the execution of a program. Depending on options selected earlier, it is sometimes necessary to stop and restart option processing, or to select completely different sets of options. This can be done easily, but you generally do not want to do this.

The number of arguments processed always includes the program name. If one of the arguments is "–", then it is counted and the processing stops. If an error was encountered and errors are to be tolerated, then the returned value is the index of the argument causing the error.

Errors will cause diagnostics to be printed. `exit(3)` may or may not be called. It depends upon whether or not the options were generated with the "allow-errors" attribute, or if the ERRSKIP_OPTERR or ERRSTOP_OPTERR macros were invoked.

7.4.28.4 optionRestore

restore option state from memory copy

Usage:

```
optionRestore( pOpts );
```

Where the arguments are:

Name	Type	Description
—–	—–	———–
pOpts	`tOptions*`	program options descriptor

Copy back the option state from saved memory. The allocated memory is left intact, so this routine can be called repeatedly without having to call optionSaveState again.

If you have not called `optionSaveState` before, a diagnostic is printed to `stderr` and exit is called.

7.4.28.5 optionSaveFile

saves the option state to a file

Usage:

```
optionSaveFile( pOpts );
```

Where the arguments are:

Name	Type	Description
—–	—–	———–
pOpts	`tOptions*`	program options descriptor

This routine will save the state of option processing to a file. The name of that file can be specified with the argument to the `--save-opts` option, or by appending the `rcfile` attribute to the last `homerc` attribute. If no `rcfile` attribute was specified, it will default to `.`*programname*`rc`. If you wish to specify another file, you should invoke the `SET_OPT_SAVE_OPTS(` *filename* `)` macro.

If no `homerc` file was specified, this routine will silently return and do nothing. If the output file cannot be created or updated, a message will be printed to `stderr` and the routine will return.

7.4.28.6 optionSaveState

saves the option state to memory

Usage:

```
optionSaveState( pOpts );
```

Where the arguments are:

Name	Type	Description
pOpts	`tOptions*`	program options descriptor

This routine will allocate enough memory to save the current option processing state. If this routine has been called before, that memory will be reused. You may only save one copy of the option state. This routine may be called before optionProcess(3).

If it fails to allocate the memory, it will print a message to stderr and exit. Otherwise, it will always succeed.

7.4.28.7 optionVersion

return the compiled AutoOpts version number

Usage:

```
const char* res = optionVersion();
```

Where the arguments are:

Name	Type	Description
returns	const char*	the version string in constant memory

Returns the full version string compiled into the library. The returned string cannot be modified.

7.5 Option Descriptor File

This is the module that is to be compiled and linked with your program. It contains internal data and procedures subject to change. Basically, it contains a single global data structure containing all the information provided in the option definitions, plus a number of static strings and any callout procedures that are specified or required. You should never have need for looking at this, except, perhaps, to examine the code generated for implementing the `flag_code` construct.

7.6 Using AutoOpts

There are actually several levels of “using” autoopts. Which you choose depends upon how you plan to distribute (or not) your application.

7.6.1 local-only use

To use AutoOpts in your application where you do not have to worry about distribution issues, your issues are simple and few.

- Create a file ‘`myopts.def`’, according to the documentation above. It is probably easiest to start with the example in Section 7.2 [Quick Start], page 69 and edit it into the form you need.
- Run AutoGen to create the option interface file (`myopts.h`) and the option descriptor code (`myopts.c`):

```
autogen myopts.def
```

- In all your source files where you need to refer to option state, `#include "myopts.h"`.
- In your main routine, code something along the lines of:

```
#define ARGC_MIN some-lower-limit
#define ARGC_MAX some-upper-limit
main( int argc, char** argv )
{
    {
        int arg_ct = optionProcess( &myprogOptions, argc, argv );
        argc -= arg_ct;
        if ((argc < ARGC_MIN) || (argc > ARGC_MAX)) {
            fprintf( stderr, "%s ERROR:  remaining args (%d) "
                     "out of range\n", myprogOptions.pzProgName,
                     argc );

            USAGE( EXIT_FAILURE );
        }
        argv += arg_ct;
    }
    if (HAVE_OPT(OPTN_NAME))
        respond_to_optn_name();
    ...
}
```

- Compile ‘`myopts.c`’ and link your program with the following additional arguments:

```
myopts.c -I$prefix/include -L $prefix/lib -lopts
```

These values can be derived from the “autoopts-config” script:

```
myopts.c ‘autoopts-config cflags‘ ‘autoopts-config ldflags‘
```

7.6.2 binary distro, AutoOpts not installed

If you will be distributing (or copying) your project to a system that does not have AutoOpts installed, you will need to statically link the AutoOpts library, “libopts” into your program. Add the output from the following to your link command:

```
autoopts-config static-libs
```

7.6.3 binary distro, AutoOpts pre-installed

If you will be distributing (or copying) your project to a system that does have AutoOpts installed, you will still need to ensure that the library is findable at program load time, or you will still have to statically link. The former can be accomplished by linking your project with `--rpath` or by setting the `LD_LIBRARY_PATH` appropriately. Otherwise, See Section 7.6.2 [binary not installed], page 94.

7.6.4 source distro, AutoOpts pre-installed

If you will be distributing your project to a system that will build your product but it may not be pre-installed with AutoOpts, you will need to do some configuration checking before you start the build. Assuming you are willing to fail the build if AutoOpts has not been installed, you will still need to do a little work.

AutoOpts is distributed with a configuration check M4 script, ‘`autoopts.m4`’. It will add an `autoconf` macro named, `AG_PATH_AUTOOPTS`. Add this to your ‘`configure.ac`’ script and use the following substitution values:

`AUTOGEN`	the name of the autogen executable
`AUTOGEN_TPLIB`	the directory where AutoGen template library is stored
`AUTOOPTS_CFLAGS`	the compile time options needed to find the AutoOpts headers
`AUTOOPTS_LIBS`	the link options required to access the `libopts` library

7.6.5 source distro, AutoOpts not installed

If you will be distributing your project to a system that will build your product but it may not be pre-installed with AutoOpts, you may wish to incorporate the sources for `libopts` in your project. To do this, I recommend reading the tear-off libopts library ‘`README`’ that you can find in the ‘`pkg/libopts`’ directory. You can also examine an example package (blocksort) that incorporates this tear off library in the autogen distribution directory. There is also a web page that describes what you need to do:

```
http://autogen.sourceforge.net/blocksort.html
```

7.7 AutoOpts for Shell Scripts

AutoOpts may be used with shell scripts by automatically creating a complete program that will process command line options and pass back the results to the invoking shell by issuing shell variable assignment commands. It may also be used to generate portable shell code that can be inserted into your script.

The functionality of these features, of course, is somewhat constrained compared with the normal program facilities. Specifically, you cannot invoke callout procedures with either of these methods. Additionally, if you generate a shell script:

1. You cannot obtain options from RC/INI files.
2. You cannot obtain options from environment variables.
3. You cannot save the option state to an option file.
4. Option conflict/requirement verification is disabled.

Both of these methods are enabled by running AutoGen on the definitions file with the additional global attribute:

```
test-main [ = proc-to-call ] ;
```

If you do not supply a `proc-to-call`, it will default to `putBourneShell`. That will produce a program that will process the options and generate shell text for the invoking shell to interpret. If you supply the name, `putShellParse`, then you will have a program that will generate a shell script that can parse the options. If you supply a different procedure name, you will have to provide that routine and it may do whatever you like.

In summary, you will need to issue approximately the following two commands to have a working program:

```
autogen -L <opt-template-dir> program.def
cc -o progopts -L <opt-lib-dir> -I <opt-include-dir> \
        -DTEST_program_OPTS program.c -lopts
```

The resulting program can be used within your shell script as follows:

```
eval ‘./progopts $@‘
if [ -z "${OPTION_CT}" ] ; then exit 1 ; fi
shift ${OPTION_CT}
```

If you had used `test-main = putShellParse` instead, then you can, at this point, merely run the program and it will write the parsing script to standard out. You may also provide this program with command line options to specify the shell script file to create or edit, and you may specify the shell program to use on the first shell script line. That program's usage text would look something like this:

```
genshellopt - Generate Shell Option Processing Script - Ver. 1
USAGE:  genshellopt [ -<flag> [<val>] | --<name>[{=| }<val>] ]...
  Flg Arg Option-Name    Description
   -o Str script         Output Script File
   -s Str shell          Shell name (follows "#!" magic)
                                - disabled as --no-shell
                                - enabled by default
   -v opt version        Output version information and exit
   -? no  help           Display usage information and exit
   -! no  more-help      Extended usage information passed thru pager

Options are specified by doubled hyphens and their name
or by a single hyphen and the flag character.

Note that ‘shell’ is only useful if the output file does not already
exist.  If it does, then the shell name and optional first argument
will be extracted from the script file.

If the script file already exists and contains Automated Option Processing
```

```
text, the second line of the file through the ending tag will be replaced
by the newly generated text.  The first `#!' line will be regenerated.

please send bug reports to:  autogen-bugs@lists.sf.net

= = = = = = = =

This incarnation of genshell will produce
a shell script to parse the options for getdefs:

getdefs - AutoGen Definition Extraction Tool - Ver. 1.4
USAGE:  getdefs [ <option-name>[{=| }<val>] ]...
   Arg Option-Name    Description
   Str defs-to-get    Regexp to look for after the "/*="
   opt ordering       Alphabetize or use named file
   Num first-index    The first index to apply to groups
   Str input          Input file to search for defs
   Str subblock       subblock definition names
   Str listattr       attribute with list of values
   opt filelist       Insert source file names into defs
   Str assign         Global assignments
   Str common-assign  Assignments common to all blocks
   Str copy           File(s) to copy into definitions
   opt srcfile        Insert source file name into each def
   opt linenum        Insert source line number into each def
   Str output         Output file to open
   opt autogen        Invoke AutoGen with defs
   Str template       Template Name
   Str agarg          AutoGen Argument
   Str base-name      Base name for output file(s)
   opt version        Output version information and exit
   no  help           Display usage information and exit
   no  more-help      Extended usage information passed thru pager
   opt save-opts      Save the option state to an rc file
   Str load-opts      Load options from an rc file

All arguments are named options.

If no ``input'' argument is provided or is set to simply "-", and if
``stdin'' is not a ``tty'', then the list of input files will be
read from ``stdin''.

please send bug reports to:  autogen-bugs@lists.sf.net
```

7.8 Automated Info Docs

AutoOpts provides two templates for producing '`.texi`' documentation. '`aginfo.tpl`' for the invoking section, and '`aginfo3.tpl`' for describing exported library functions and macros.

For both types of documents, the documentation level is selected by passing a '`-DLEVEL=<level-name>`' argument to AutoGen when you build the document. (See the example invocation below.)

Two files will be produced, a '`.texi`' file and a '`.menu`' file. You should include the '`.menu`' file in your document where you wish to reference the '`invoking`' chapter, section or subsection.

The '`.texi`' file will contain an introductory paragraph, a menu and a subordinate section for the invocation usage and for each documented option. The introductory paragraph is normally the boiler plate text, along the lines of:

```
This chapter documents the @file{AutoOpts} generated usage text
and option meanings for the @file{your-program} program.
```

or:

```
These are the publicly exported procedures from the libname library.
Any other functions mentioned in the header file are for the private use
of the library.
```

7.8.1 "invoking" info docs

Using the option definitions for an AutoOpt client program, the '`aginfo.tpl`' template will produce texinfo text that documents the invocation of your program. The text emitted is designed to be included in the full texinfo document for your product. It is not a standalone document. The usage text for the Section 5.1 [autogen usage], page 53, Section 8.5.1 [getdefs usage], page 107 and Section 8.4.1 [columns usage], page 103 programs, are included in this document and are all generated using this template.

If your program's option definitions include a '`prog-info-descrip`' section, then that text will replace the boilerplate introductory paragraph.

These files are produced by invoking the following command:

```
autogen -L ${prefix}/share/autogen -T aginfo.tpl \
        -DLEVEL=section your-opts.def
```

Where '`${prefix}`' is the AutoGen installation prefix and '`your-opts.def`' is the name of your product's option definition file.

7.8.2 library info docs

The '`texinfo`' doc for libraries is derived from mostly the same information as is used for producing man pages See Section 7.9.2 [man3], page 99. The main difference is that there is only one output file and the individual functions are referenced from a `.texi` menu. There is also a small difference in the global attributes used:

lib_description	A description of the library. This text appears before the menu. If not provided, the standard boilerplate version will be inserted.
see_also	The `SEE ALSO` functionality is not supported for the '`texinfo`' documentation, so any `see_also` attribute will be ignored.

These files are produced by invoking the following commands:

```
getdefs linenum srcfile template=aginfo3.tpl output=libexport.def \
       <source-file-list>

autogen -L ${prefix}/share/autogen -DLEVEL=section libexport.def
```

Where '`${prefix}`' is the AutoGen installation prefix and '`libexport.def`' is some name that suits you.

An example of this can be seen in this document, See Section 7.4.28 [libopts procedures], page 90.

7.9 Automated Man Pages

AutoOpts provides two templates for producing man pages. The command ('`man1`') pages are derived from the options definition file, and the library ('`man3`') pages are derived from stylized comments (see Section 8.5 [getdefs Invocation], page 106).

7.9.1 command line man pages

Using the option definitions for an AutoOpt client program, the '`agman1.tpl`' template will produce an nroff document suitable for use as a '`man(1)`' page document for a command line command. The description section of the document is either the '`prog-man-descrip`' text, if present, or the '`detail`' text.

Each option in the option definitions file is fully documented in its usage. This includes all the information documented above for each option (see Section 7.3.4 [option attributes], page 75), plus the '`doc`' attribute is appended. Since the '`doc`' text is presumed to be designed for `texinfo` documentation, `sed` is used to convert some constructs from `texi` to `nroff`-for-`man`-pages. Specifically,

```
convert @code, @var and @samp into \fB...\fP phrases
convert @file into \fI...\fP phrases
Remove the '@' prefix from curly braces
Indent example regions
Delete the example commands
Replace `end example' command with ".br"
Replace the `@*' command with ".br"
```

This document is produced by invoking the following command:

```
AutoGen -L ${prefix}/share/autogen -T agman1.tpl options.def
```

Where '`${prefix}`' is the AutoGen installation prefix and '`options.def`' is the name of your product's option definition file. I do not use this very much, so any feedback or improvements would be greatly appreciated.

7.9.2 library man pages

Two global definitions are required, and then one library man page is produced for each `export_func` definition that is found. It is generally convenient to place these definitions as '`getdefs`' comments (see Section 8.5 [getdefs Invocation], page 106) near the procedure definition, but they may also be a separate AutoGen definitions file (see Chapter 2 [Definitions File], page 6). Each function will be cross referenced with their sister functions in a

'`SEE ALSO`' section. A global `see_also` definition will be appended to this cross referencing text.

The two global definitions required are:

library	This is the name of your library, without the '`lib`' prefix. The AutoOpts library is named '`libopts.so...`', so the `library` attribute would have the value `opts`.
header	Generally, using a library with a compiled program entails `#include`-ing a header file. Name that header with this attribute. In the case of AutoOpts, it is generated and will vary based on the name of the option definition file. Consequently, '`your-opts.h`' is specified.

The `export_func` definition should contain the following attributes:

name	The name of the procedure the library user may call.
what	A brief sentence describing what the procedure does.
doc	A detailed description of what the procedure does. It may ramble on for as long as necessary to properly describe it.
err	A short description of how errors are handled.
ret_type	The data type returned by the procedure. Omit this for `void` procedures.
ret_desc	Describe what the returned value is, if needed.
private	If specified, the function will **not** be documented. This is used, for example, to produce external declarations for functions that are not available for public use, but are used in the generated text.
arg	This is a compound attribute that contains:

arg_type	The data type of the argument.
arg_name	A short name for it.
arg_desc	A brief description.

As a '`getdefs`' comment, this would appear something like this:

```
/*=--subblock=arg=arg_type,arg_name,arg_desc =*/
/*=*
 * library: opts
 * header:  your-opts.h
=*/
/*=export_func optionProcess
 *
 * what: this is the main option processing routine
 * arg:  + tOptions* + pOpts + program options descriptor +
 * arg:  + int       + argc  + program arg count  +
 * arg:  + char**    + argv  + program arg vector +
 * ret_type:  int
 * ret_desc:  the count of the arguments processed
 *
 * doc:  This is what it does.
 * err:  When it can't, it does this.
=*/
```

Note the `subblock` and `library` comments. `subblock` is an embedded '`getdefs`' option (see Section 8.5.6 [getdefs subblock], page 109) that tells it how to parse the `arg` attribute.

The `library` and `header` entries are global definitions that apply to all the documented functions.

8 Add-on packages for AutoGen

This chapter includes several programs that either work closely with AutoGen (extracting definitions or providing special formatting functions), or leverage off of AutoGen technology. There is also a formatting library that helps make AutoGen possible.

AutoOpts ought to appear in this list as well, but since it is the primary reason why many people would even look into AutoGen at all, I decided to leave it in the list of chapters.

8.1 Automated Finite State Machine

The templates to generate a finite state machine in C or C++ is included with AutoGen. The documentation is not. The documentation is in HTML format for viewing, or you can download FSM.

8.2 Combined RPC Marshalling

The templates and NFSv4 definitions are not included with AutoGen in any way. The folks that designed NFSv4 noticed that much time and bandwidth was wasted sending queries and responses when many of them could be bundled. The protocol bundles the data, but there is no support for it in rpcgen. That means you have to write your own code to do that. Until now. Download this and you will have a large, complex example of how to use `AutoXDR` for generating the marshalling and unmarshalling of combined RPC calls. There is a brief example on the web, but you should download AutoXDR.

8.3 Automated Event Management

Large software development projects invariably have a need to manage the distribution and display of state information and state changes. In other words, they need to manage their software events. Generally, each such project invents its own way of accomplishing this and then struggles to get all of its components to play the same way. It is a difficult process and not always completely successful. This project helps with that.

AutoEvents completely separates the tasks of supplying the data needed for a particular event from the methods used to manage the distribution and display of that event. Consequently, the programmer writing the code no longer has to worry about that part of the problem. Likewise the persons responsible for designing the event management and distribution no longer have to worry about getting programmers to write conforming code.

This is a work in progress. See my web page on the subject, if you are interested. I have some useful things put together, but it is not ready to call a product.

8.4 Invoking columns

This program was designed for the purpose of generating compact, columnized tables. It will read a list of text items from standard in or a specified input file and produce a columnized listing of all the non-blank lines. Leading white space on each line is preserved, but trailing white space is stripped. Methods of applying per-entry and per-line embellishments are provided. See the formatting and separation arguments below.

This program is used by AutoGen to help clean up and organize its output.

See ‘`autogen/agen5/fsm.tpl`’ and the generated output ‘`pseudo-fsm.h`’.

This function was not implemented as an expression function because either it would have to be many expression functions, or a provision would have to be added to provide options to expression functions. Maybe not a bad idea, but it is not being implemented at the moment.

A side benefit is that you can use it outside of AutoGen to columnize input, a la the `ls` command.

This section was generated by **AutoGen**, the aginfo template and the option descriptions for the **columns** program. It documents the columns usage text and option meanings.

This software is released under the GNU General Public License.

8.4.1 columns usage help (-?)

This is the automatically generated usage text for columns:

```
columns - Columnize Input Text - Ver. 1.1
USAGE:  columns [ -<flag> [<val>] | --<name>[{=| }<val>] ]...
  Flg Arg Option-Name    Description
   -W Num width          Maximum Line Width
   -c Num columns        Desired number of columns
   -w Num col-width      Set width of each column
      Num spread         maximum spread added to column width
   -I Str indent         Line prefix or indentation
      Str first-indent   First line prefix
                                - requires these options:
                                indent
      Num tab-width      tab width
   -s opt sort           Sort input text
   -f Str format         Formatting string for each input
   -S Str separation     Separation string - follows all but last
      Str line-separation string at end of all lines but last
      no  by-columns     Print entries in column order
   -i Str input          Input file (if not stdin)
   -v opt version        Output version information and exit
   -? no  help           Display usage information and exit
   -! no  more-help      Extended usage information passed thru pager

Options are specified by doubled hyphens and their name
or by a single hyphen and the flag character.
```

```
This program was designed for the purpose of generating compact,
columnized tables.  It will read a list of text items from standard
in or a specified input file and produce a columnized listing of
all the non-blank lines.  Leading white space on each line is
preserved, but trailing white space is stripped.  Methods of
applying per-entry and per-line embellishments are provided.
See the formatting and separation arguments below.

This program is used by AutoGen to help clean up and organize
its output.

please send bug reports to:  autogen-bugs@lists.sf.net
```

8.4.2 width option (-W)

This is the "maximum line width" option. This option specifies the full width of the output line, including any start-of-line indentation. The output will fill each line as completely as possible, unless the column width has been explicitly specified. If the maximum width is less than the length of the widest input, you will get a single column of output.

8.4.3 columns option (-c)

This is the "desired number of columns" option. Use this option to specify exactly how many columns to produce. If that many columns will not fit within *line_width*, then the count will be reduced to the number that fit.

8.4.4 col-width option (-w)

This is the "set width of each column" option. Use this option to specify exactly how many characters are to be allocated for each column. If it is narrower than the widest entry, it will be over-ridden with the required width.

8.4.5 spread option

This is the "maximum spread added to column width" option. Use this option to specify exactly how many characters may be added to each column. It allows you to prevent columns from becoming too far apart.

8.4.6 indent option (-I)

This is the "line prefix or indentation" option. If a number, then this many spaces will be inserted at the start of every line. Otherwise, it is a line prefix that will be inserted at the start of every line.

8.4.7 first-indent option

This is the "first line prefix" option.

This option has some usage constraints. It:

- must appear in combination with the following options: indent.

If a number, then this many spaces will be inserted at the start of the first line. Otherwise, it is a line prefix that will be inserted at the start of that line.

8.4.8 tab-width option

This is the “tab width” option. If an indentation string contains tabs, then this value is used to compute the ending column of the prefix string.

8.4.9 sort option (-s)

This is the “sort input text” option. Causes the input text to be sorted. If an argument is supplied, it is presumed to be a pattern and the sort is based upon the matched text. If the pattern starts with or consists of an asterisk (`*`), then the sort is case insensitive.

8.4.10 format option (-f)

This is the “formatting string for each input” option. If you need to reformat each input text, the argument to this option is interpreted as an `sprintf(3)` format that is used to produce each output entry.

8.4.11 separation option (-S)

This is the “separation string - follows all but last” option. Use this option if, for example, you wish a comma to appear after each entry except the last.

8.4.12 line-separation option

This is the “string at end of all lines but last” option. Use this option if, for example, you wish a backslash to appear at the end of every line, except the last.

8.4.13 by-columns option

This is the “print entries in column order” option. Normally, the entries are printed out in order by rows and then columns. This option will cause the entries to be ordered within columns. The final column, instead of the final row, may be shorter than the others.

8.4.14 input option (-i)

This is the “input file (if not stdin)” option. This program normally runs as a `filter`, reading from standard input, columnizing and writing to standard out. This option redirects input to a file.

8.5 Invoking getdefs

If no `input` argument is provided or is set to simply `"-"`, and if `stdin` is not a `tty`, then the list of input files will be read from `stdin`. This program extracts AutoGen definitions from a list of source files. Definitions are delimited by '/*=<entry-type> <entry-name>\n' and '=*/\n'. From that, this program creates a definition of the following form:

```
#line nnn "source-file-name"
entry_type = {
    name = entry_name;
    ...
};
```

The ellipsis "..." is filled in by text found between the two delimiters, using the following rules:

1. Each entry is located by the pattern "\n[^*\n]*\\*[\t]*([a-z][a-z0-9_]*):". Fundamentally, it finds a line that, after the first asterisk on the line, contains whitespace then a name and is immediately followed by a colon. The name becomes the name of the attribute and what follows, up to the next attribute, is its value.
2. If the first character of the value is either a single or double quote, then you are responsible for quoting the text as it gets inserted into the output definitions.
3. All the leading text on a line is stripped from the value. The leading text is everything before the first asterisk, the asterisk and all the whitespace characters that immediately follow it. If you want whitespace at the beginnings of the lines of text, you must do something like this:

   ```
    * mumble:
    * "  this is some\n"
    * "  indented text."
   ```

4. If the '<entry-name>' is followed by a comma, the word 'ifdef' (or 'ifndef') and a name 'if_name', then the above entry will appear as:

   ```
   #ifdef if_name
   #line nnn "source-file-name"
   entry_type = {
       name = entry_name;
       ...
   };
   #endif
   ```

5. If you use of the `subblock` option, you can specify a nested value, See Section 8.5.6 [getdefs subblock], page 109. That is, this text:

   ```
    * arg:  int, this, what-it-is
   ```

   with the '–subblock=arg=type,name,doc' option would yield:

   ```
   arg = { type = int; name = this; doc = what-it-is; };
   ```

This section was generated by **AutoGen**, the aginfo template and the option descriptions for the **getdefs** program. It documents the getdefs usage text and option meanings.

This software is released under the GNU General Public License.

8.5.1 getdefs usage help

This is the automatically generated usage text for getdefs:

```
getdefs - AutoGen Definition Extraction Tool - Ver. 1.4
USAGE:  getdefs [ <option-name>[{=| }<val>] ]...
   Arg Option-Name    Description
   Str defs-to-get    Regexp to look for after the "/*="
   opt ordering       Alphabetize or use named file
                                - disabled as --no-ordering
                                - enabled by default
   Num first-index    The first index to apply to groups
   Str input          Input file to search for defs
                                - may appear multiple times
                                - default option for unnamed options
   Str subblock       subblock definition names
                                - may appear multiple times
   Str listattr       attribute with list of values
                                - may appear multiple times
   opt filelist       Insert source file names into defs

Definition insertion options

   Arg Option-Name    Description
   Str assign         Global assignments
                                - may appear multiple times
   Str common-assign  Assignments common to all blocks
                                - may appear multiple times
   Str copy           File(s) to copy into definitions
                                - may appear multiple times
   opt srcfile        Insert source file name into each def
   opt linenum        Insert source line number into each def

Definition output disposition options:

   Arg Option-Name    Description
   Str output         Output file to open
                                - an alternate for autogen
   opt autogen        Invoke AutoGen with defs
                                - disabled as --no-autogen
                                - enabled by default
   Str template       Template Name
   Str agarg          AutoGen Argument
                                - prohibits these options:
                                output
                                - may appear multiple times
   Str base-name      Base name for output file(s)
                                - prohibits these options:
                                output
```

```
Auto-supported Options:

   Arg Option-Name    Description
   opt version        Output version information and exit
   no  help           Display usage information and exit
   no  more-help      Extended usage information passed thru pager
   opt save-opts      Save the option state to an rc file
   Str load-opts      Load options from an rc file
                                - disabled as --no-load-opts
                                - may appear multiple times

All arguments are named options.

If no ‘‘input’’ argument is provided or is set to simply "-", and if
‘‘stdin’’ is not a ‘‘tty’’, then the list of input files will be
read from ‘‘stdin’’.

The following option preset mechanisms are supported:
 - reading file /dev/null

This program extracts AutoGen definitions from a list of source files.
Definitions are delimited by ‘/*=<entry-type> <entry-name>\n’ and
‘=*/\n’.  From that, this program creates a definition of the
following form:

    #line nnn "source-file-name"
    entry_type = {
        name = entry_name;
        ...
    };

The ellipsis ’...’ is filled in by text found between the two
delimiters, with everything up through the first sequence of
asterisks deleted on every line.

There are two special ‘‘entry types’’:

*  The entry_type enclosure and the name entry will be omitted
   and the ellipsis will become top-level definitions.

-- The contents of the comment must be a single getdefs option.
   The option name must follow the double hyphen and its argument
   will be everything following the name.  This is intended for use
   with the ‘‘subblock’’ and ‘‘listattr’’ options.

please send bug reports to:  autogen-bugs@lists.sf.net
```

8.5.2 defs-to-get option

This is the “regexp to look for after the "/*="” option. If you want definitions only from a particular category, or even with names matching particular patterns, then specify this regular expression for the text that must follow the `/*=`.

8.5.3 ordering option

This is the “alphabetize or use named file” option.

This option has some usage constraints. It:

- is enabled by default.

By default, ordering is alphabetical by the entry name. Use, `no-ordering` if order is unimportant. Use `ordering` with no argument to order without case sensitivity. Use `ordering=<file-name>` if chronological order is important. getdefs will maintain the text content of `file-name`. `file-name` need not exist.

8.5.4 first-index option

This is the “the first index to apply to groups” option. By default, the first occurrance of a named definition will have an index of zero. Sometimes, that needs to be a reserved value. Provide this option to specify a different starting point.

8.5.5 input option

This is the “input file to search for defs” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

All files that are to be searched for definitions must be named on the command line or read from `stdin`. If there is only one `input` option and it is the string, `"-"`, then the input file list is read from `stdin`. If a command line argument is not an option name and does not contain an assignment operator (`=`), then it defaults to being an input file name. At least one input file must be specified.

8.5.6 subblock option

This is the “subblock definition names” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

This option is used to create shorthand entries for nested definitions. For example, with:

using subblock thus

```
--subblock=arg=argname,type,null
```

and defining an `arg` thus

```
arg: this, char *
```

will then expand to:

```
arg = { argname = this; type = "char *"; };
```

The "this, char *" string is separated at the commas, with the white space removed. You may use characters other than commas by starting the value string with a punctuation character other than a single or double quote character. You may also omit intermediate values by placing the commas next to each other with no intervening white space. For example, "+mumble++yes+" will expand to:
`arg = { argname = mumble; null = "yes"; };`.

8.5.7 listattr option

This is the "attribute with list of values" option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

This option is used to create shorthand entries for definitions that generally appear several times. That is, they tend to be a list of values. For example, with:
`listattr=foo` defined, the text:
`foo: this, is, a, multi-list` will then expand to:
`foo = 'this', 'is', 'a', 'multi-list';`
The texts are separated by the commas, with the white space removed. You may use characters other than commas by starting the value string with a punctuation character other than a single or double quote character.

8.5.8 filelist option

This is the "insert source file names into defs" option. Inserts the name of each input file into the output definitions. If no argument is supplied, the format will be:

```
infile = '%s';
```

If an argument is supplied, that string will be used for the entry name instead of *infile*.

8.5.9 assign option

This is the "global assignments" option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

The argument to each copy of this option will be inserted into the output definitions, with only a semicolon attached.

8.5.10 common-assign option

This is the "assignments common to all blocks" option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

The argument to each copy of this option will be inserted into each output definition, with only a semicolon attached.

8.5.11 copy option

This is the “file(s) to copy into definitions” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

The content of each file named by these options will be inserted into the output definitions.

8.5.12 srcfile option

This is the “insert source file name into each def” option. Inserts the name of the input file where a definition was found into the output definition. If no argument is supplied, the format will be:

```
srcfile = ’%s’;
```

If an argument is supplied, that string will be used for the entry name instead of *srcfile*.

8.5.13 linenum option

This is the “insert source line number into each def” option. Inserts the line number in the input file where a definition was found into the output definition. If no argument is supplied, the format will be:

```
linenum = ’%s’;
```

If an argument is supplied, that string will be used for the entry name instead of *linenum*.

8.5.14 output option

This is the “output file to open” option.

This option has some usage constraints. It:

- is a member of the autogen class of options.

If you are not sending the output to an AutoGen process, you may name an output file instead.

8.5.15 autogen option

This is the “invoke autogen with defs” option.

This option has some usage constraints. It:

- is enabled by default.
- is a member of the autogen class of options.

This is the default output mode. Specifying `no-autogen` is equivalent to `output=-`. If you supply an argument to this option, that program will be started as if it were AutoGen and its standard in will be set to the output definitions of this program.

8.5.16 template option

This is the “template name” option. Specifies the template name to be used for generating the final output.

8.5.17 agarg option

This is the “autogen argument” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.
- must not appear in combination with any of the following options: output.

This is a pass-through argument. It allows you to specify any arbitrary argument to be passed to AutoGen.

8.5.18 base-name option

This is the “base name for output file(s)” option.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: output.

When output is going to AutoGen, a base name must either be supplied or derived. If this option is not supplied, then it is taken from the `template` option. If that is not provided either, then it is set to the base name of the current directory.

8.6 Invoking xml2ag

This program will convert any arbitrary XML file into equivalent AutoGen definitions, and invoke AutoGen. The template used will be derived from either:

- The –**override-tpl** command line option
- A top level XML attribute named, `"template"`

The *base-name* for the output will similarly be either:

- The –**base-name** command line option.
- The base name of the '`.xml`' file.

The definitions derived from XML generally have an extra layer of definition. Specifically, this XML input:

```
<mumble attr="foo">
  mumble-1
  <grumble>
  grumble, grumble, grumble.
</grumble>mumble, mumble
</mumble>
```

Will get converted into this:

```
mumble = {
  grumble = {
    text = 'grumble, grumble, grumble';
  };
  text = 'mumble-1';
  text = 'mumble, mumble';
};
```

Please notice that some information is lost. AutoGen cannot tell that "grumble" used to lie between the mumble texts. Also please note that you cannot assign:

```
grumble = 'grumble, grumble, grumble.';
```

because if another "grumble" has an attribute or multiple texts, it becomes impossible to have the definitions be the same type (compound or text values).

This section was generated by **AutoGen**, the aginfo template and the option descriptions for the **xml2ag** program. It documents the xml2ag usage text and option meanings.

This software is released under the GNU General Public License.

8.6.1 xml2ag usage help (-?)

This is the automatically generated usage text for xml2ag:

```
xml2ag - XML to AutoGen Definiton Converter - Ver. 5.5.4
USAGE:  xml2ag [ -<flag> [<val>] | --<name>[{=| }<val>] ]... [ <def-file> ]
  Flg Arg Option-Name    Description
   -O Str output         Output file in lieu of AutoGen processing
   -L Str templ-dirs     Template search directory list
                                - may appear multiple times
   -T Str override-tpl   Override template file
   -l Str lib-template   Library template file
```

```
                                - may appear multiple times
   -b Str base-name      Base name for output file(s)
      Str definitions    Definitions input file
   -S Str load-scheme    Scheme code file to load
   -F Str load-functions Load scheme callout library
   -s Str skip-suffix    Omit the file with this suffix
                                - may appear multiple times
   -o opt select-suffix  specify this output suffix
                                - may appear multiple times
      no  source-time    set mod times to latest source
      Str equate         characters considered equivalent
      no  writable       Allow output files to be writable
                                - disabled as --not-writable
      Num loop-limit     Limit on increment loops
                                  it must lie in one of the ranges:
                                  -1 exactly, or
                                  1 to 16777216
   -t Num timeout        Time limit for servers
                                  it must lie in the range: 0 to 3600
      KWd trace          tracing level of detail
      Str trace-out      tracing output file or filter
      no  show-defs      Show the definition tree
      no  show-shell     Show shell commands
   -D Str define         name to add to definition list
                                - may appear multiple times
   -U Str undefine       definition list removal pattern
                                - an alternate for define
   -v opt version        Output version information and exit
   -? no  help           Display usage information and exit
   -! no  more-help      Extended usage information passed thru pager

Options are specified by doubled hyphens and their name
or by a single hyphen and the flag character.

This program will convert any arbitrary XML file into equivalent
AutoGen definitions, and invoke AutoGen.

The valid trace option keywords are:
        nothing
        templates
        block-macros
        expressions
        everything

The template will be derived from either:
*  the ``--override-tpl'' command line option
*  a top level XML attribute named, "template"

The ``base-name'' for the output will similarly be either:
```

```
*  the ‘‘--base-name’’ command line option
*  the base name of the .xml file

please send bug reports to:  autogen-bugs@lists.sf.net
```

8.6.2 output option (-O)

This is the “output file in lieu of autogen processing” option. By default, the output is handed to an AutoGen for processing. However, you may save the definitions to a file instead.

8.6.3 templ-dirs option (-L)

This is the “template search directory list” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

Pass-through AutoGen argument

8.6.4 override-tpl option (-T)

This is the “override template file” option. Pass-through AutoGen argument

8.6.5 lib-template option (-l)

This is the “library template file” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

Pass-through AutoGen argument

8.6.6 base-name option (-b)

This is the “base name for output file(s)” option. Pass-through AutoGen argument

8.6.7 definitions option

This is the “definitions input file” option. Pass-through AutoGen argument

8.6.8 load-scheme option (-S)

This is the “scheme code file to load” option. Pass-through AutoGen argument

8.6.9 load-functions option (-F)

This is the “load scheme callout library” option. Pass-through AutoGen argument

8.6.10 skip-suffix option (-s)

This is the “omit the file with this suffix” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

Pass-through AutoGen argument

8.6.11 select-suffix option (-o)

This is the “specify this output suffix” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.

Pass-through AutoGen argument

8.6.12 source-time option

This is the “set mod times to latest source” option. Pass-through AutoGen argument

8.6.13 equate option

This is the “characters considered equivalent” option. Pass-through AutoGen argument

8.6.14 writable option

This is the “allow output files to be writable” option. Pass-through AutoGen argument

8.6.15 loop-limit option

This is the “limit on increment loops” option. Pass-through AutoGen argument

8.6.16 timeout option (-t)

This is the “time limit for servers” option. Pass-through AutoGen argument

8.6.17 trace option

This is the “tracing level of detail” option. Pass-through AutoGen argument

8.6.18 trace-out option

This is the “tracing output file or filter” option. Pass-through AutoGen argument

8.6.19 show-defs option

This is the “show the definition tree” option. Pass-through AutoGen argument

8.6.20 show-shell option

This is the “show shell commands” option. Pass-through AutoGen argument

8.6.21 define option (-D)

This is the “name to add to definition list” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.
- is a member of the define class of options.

Pass-through AutoGen argument

8.6.22 undefine option (-U)

This is the “definition list removal pattern” option.

This option has some usage constraints. It:

- may appear an unlimited number of times.
- is a member of the define class of options.

Pass-through AutoGen argument

8.7 Replacement for Stdio Formatting Library

Using the ‘printf’ formatting routines in a portable fashion has always been a pain, and this package has been way more pain than anyone ever imagined. Hopefully, with this release of snprintfv, the pain is now over for all time.

The issues with portable usage are these:

1. Argument number specifiers are often either not implemented or are buggy. Even GNU libc, version 1 got it wrong.
2. ANSI/ISO "forgot" to provide a mechanism for computing argument lists for vararg procedures.
3. The argument array version of printf (‘printfv()’) is not generally available, does not work with the native printf, and does not have a working argument number specifier in the format specification. (Last I knew, anyway.)
4. You cannot fake varargs by calling ‘vprintf()’ with an array of arguments, because ANSI does not require such an implementation and some vendors play funny tricks because they are allowed to.

These four issues made it impossible for AutoGen to ship without its own implementation of the ‘printf’ formatting routines. Since we were forced to do this, we decided to make the formatting routines both better and more complete :-). We addressed these issues and added the following features to the common printf API:

5. The formatted output can be written to
   - a string allocated by the formatting function (‘asprintf()’).
   - a file descriptor instead of a file stream (‘dprintf()’).
   - a user specified stream (‘stream_printf()’).
6. The formatting functions can be augmented with your own functions. These functions are allowed to consume more than one character from the format, but must commence with a unique character. For example,

   ```
   "%{struct stat}\n"
   ```

   might be used with ’`{`’ registered to a procedure that would look up "struct stat" in a symbol table and do appropriate things, consuming the format string through the ’`}`’ character.

Gary V. Vaughan was generous enough to supply this implementation. Many thanks!!

For further details, the reader is referred to the snprintfv documentation. These functions are also available in the template processing as ‘sprintf’ (see Section 3.5.25 [SCM sprintf], page 38), ‘printf’ (see Section 3.5.20 [SCM printf], page 36), ‘fprintf’ (see Section 3.5.6 [SCM fprintf], page 33), and ‘shellf’ (see Section 3.5.24 [SCM shellf], page 38).

9 Some ideas for the future.

Here are some things that might happen in the distant future.

- Write code for `"AutoGetopts"` (GNU getopt), or possibly the new glibc argp parser.
- Fix up current tools that contain miserably complex perl, shell, sed, awk and m4 scripts to instead use this tool.

Concept Index

Function Index

Table of Contents

8 Add-on packages for AutoGen 102