GYP

Home User documentation Input Format Reference Language specification Hacking Testing GYP vs. CMake

Input Format Reference

Contents

Primitive Types

The following primitive types are found within input files:

- String values, which may be represented by enclosing them in `'single quotes'` or `"double quotes"`. By convention, single quotes are used.
- Integer values, which are represented in decimal without any special decoration. Integers are fairly rare in input files, but have a few applications in boolean contexts, where the convention is to represent true values with `1` and false with `0`.
- Lists, which are represented as a sequence of items separated by commas (`,`) within square brackets (`[` and `]`). A list may contain any other primitive types, including other lists. Generally, each item of a list must be of the same type as all other items in the list, but in some cases (such as within `conditions` sections), the list structure is more tightly specified. A trailing comma is permitted.

  This example list contains three string values.

  `[ 'Generate', 'Your', 'Projects', ]`

- Dictionaries, which map keys to values. All keys are strings. Values may be of any other primitive type, including other dictionaries. A dictionary is enclosed within curly braces (`{` and `}`). Keys precede values, separated by a colon (`:`). Successive dictionary entries are separated by commas (`,`). A trailing comma is permitted. It is an error for keys to be duplicated within a single dictionary as written in an input file, although keys may replace other keys during merging.

  This example dictionary maps each of three keys to different values.

  `{ 'inputs': ['version.c.in'], 'outputs': ['version.c'], 'process_outputs_as_sources': 1, }`

Overall Structure

A GYP input file is organized as structured data. At the root scope of each `.gyp` or `.gypi` (include) file is a dictionary. The keys and values of this dictionary, along with any descendants contained within the values, provide the data contained within the file. This data is given meaning by interpreting specific key names and their associated values in specific ways (see Settings Keys).

Comments (#)

Within an input file, a comment is introduced by a pound sign (`#`) not within a string. Any text following the pound sign, up until the end of the line, is treated as a comment.

Example

```
{
  'school_supplies': [
    'Marble composition book',
    'Sharp #2 pencil',
    'Safety scissors',  # You still shouldn't run with these
  ],
}
```

In this example, the # in `'Sharp #2 pencil'` is not taken as introducing a comment because it occurs within a string, but the text after `'Safety scissors'` is treated as a comment having no impact on the data within the file.

Merging

Merge Basics (=, ?, +)

Many operations on GYP input files occurs by merging dictionary and list items together. During merge operations, it is important to recognize the distinction between source and destination values. Items from the source value are merged into the destination, which leaves the source unchanged and the destination modified by the source. A dictionary may only be merged into another dictionary, and a list may only be merged into another list.

- When merging a dictionary, for each key in the source:
  - If the key does not exist in the destination dictionary, insert it and copy the associated value directly.
  - If the key does exist:
  - If the associated value is a dictionary, perform the dictionary merging procedure using the source's and destination's value dictionaries.
  - If the associated value is a list, perform the list merging procedure using the source's and destination's value lists.
  - If the associated value is a string or integer, the destination value is replaced by the source value.
- When merging a list, merge according to the suffix appended to the key name, if the list is a value within a dictionary.
  - If the key ends with an equals sign (`=`), the policy is for the source list to completely replace the destination list if it exists. *Mnemonic: `=` for assignment.*
  - If the key ends with a question mark (`?`), the policy is for the source list to be set as the destination list only if the key is not already present in the destination. *Mnemonic: `?` for conditional assignment*.
  - If the key ends with a plus sign (`+`), the policy is for the source list contents to be prepended to the destination list. *Mnemonic: `+` for addition or concatenation.*
  - If the list key is undecorated, the policy is for the source list contents to be appended to the destination list. This is the default list merge policy.

Example

Source dictionary:

```
{
  'include_dirs+': [
    'shared_stuff/public',
  ],
  'link_settings': {
    'libraries': [
      '-lshared_stuff',
    ],
  },
  'test': 1,
}
```

Destination dictionary:

```
{
  'target_name': 'hello',
  'sources': [
    'kitty.cc',
  ],
  'include_dirs': [
    'headers',
  ],
  'link_settings': {
    'libraries': [
      '-lm',
    ],
    'library_dirs': [
      '/usr/lib',
    ],
  },
  'test': 0,
}
```

Merged dictionary:

```
{
  'target_name': 'hello',
  'sources': [
    'kitty.cc',
  ],
  'include_dirs': [
    'shared_stuff/public',  # Merged, list item prepended due to include_dirs
    'headers',
  ],
  'link_settings': {
    'libraries': [
      '-lm',
      '-lshared_stuff',  # Merged, list item appended
    ],
    'library_dirs': [
      '/usr/lib',
    ],
  },
  'test': 1,  # Merged, int value replaced
}
```

Pathname Relativization

In a `.gyp` or `.gypi` file, many string values are treated as pathnames relative to the file in which they are defined.

String values associated with the following keys, or contained within lists associated with the following keys, are treated as pathnames:

- destination
- files
- include_dirs
- inputs
- libraries
- outputs
- sources
- mac_bundle_resources
- mac_framework_dirs
- msvs_cygwin_dirs
- msvs_props

Additionally, string values associated with keys ending in the following suffixes, or contained within lists associated with keys ending in the following suffixes, are treated as pathnames:

- `_dir`
- `_dirs`
- `_file`
- `_files`
- `_path`
- `_paths`

However, any string value beginning with any of these characters is excluded from pathname relativization:

- `/` for identifying absolute paths.
- `$` for introducing build system variable expansions.
- `-` to support specifying such items as `-llib`, meaning “library `lib` in the library search path.”
- `<`, `>`, and `!` for GYP expansions.

When merging such relative pathnames, they are adjusted so that they can remain valid relative pathnames, despite being relative to a new home.

Example

Source dictionary from `../build/common.gypi`:

```
{
  'include_dirs': ['include'],  # Treated as relative to ../build
  'libraries': ['-lz'],  # Not treated as a pathname, begins with a dash
  'defines': ['NDEBUG'],  # defines does not contain pathnames
}
```

Target dictionary, from `base.gyp`:

```
{
  'sources': ['string_util.cc'],
}
```

Merged dictionary:

```
{
  'sources': ['string_util.cc'],
  'include_dirs': ['../build/include'],
  'libraries': ['-lz'],
  'defines': ['NDEBUG'],
}
```

Because of pathname relativization, after the merge is complete, all of the pathnames in the merged dictionary are valid relative to the directory containing `base.gyp`.

List Singletons

Some list items are treated as singletons, and the list merge process will enforce special rules when merging them. At present, any string item in a list that does not begin with a dash (`-`) is treated as a singleton, although **this is subject to change.** When appending or prepending a singleton to a list, if the item is already in the list, only the earlier instance is retained in the merged list.

Example

Source dictionary:

```
{
  'defines': [
    'EXPERIMENT=1',
    'NDEBUG',
  ],
}
```

Destination dictionary:

```
{
  'defines': [
    'NDEBUG',
    'USE_THREADS',
  ],
}
```

Merged dictionary:

```
{
  'defines': [
    'NDEBUG',
    'USE_THREADS',
    'EXPERIMENT=1',  # Note that NDEBUG is not appended after this.
  ],
}
```

Including Other Files

If the `-I` (`--include`) argument was used to invoke GYP, any files specified will be implicitly merged into the root dictionary of all `.gyp` files.

An includes section may be placed anywhere within a `.gyp` or `.gypi` (include) file. `includes` sections contain lists of other files to include. They are processed sequentially and merged into the enclosing dictionary at the point that the `includes` section was found. `includes` sections at the root of a `.gyp` file dictionary are merged after any `-I` includes from the command line.

includes sections are processed immediately after a file is loaded, even before variable and conditional processing, so it is not possible to include a file based on a variable reference. While it would be useful to be able to include files based on variable expansions, it is most likely more useful to allow included files access to variables set by the files that included them.

An includes section may, however, be placed within a conditional section. The included file itself will be loaded unconditionally, but its dictionary will be discarded if the associated condition is not true.

Variables and Conditionals

Variables

There are three main types of variables within GYP.

- Predefined variables. By convention, these are named with `CAPITAL_LETTERS` . Predefined variables are set automatically by GYP. They may be overridden, but it is not advisable to do so. See Predefined Variables for a list of variables that GYP provides.
- User-defined variables. Within any dictionary, a key named `variables` can be provided, containing a mapping between variable names (keys) and their contents (values), which may be strings, integers, or lists of strings. By convention, user-defined variables are named with `lowercase_letters` .
- Automatic variables. Within any dictionary, any key with a string value has a corresponding automatic variable whose name is the same as the key name with an underscore (_) prefixed. For example, if your dictionary contains `type: 'static_library'` , an automatic variable named `_type` will be provided, and its value will be a string, `'static_library'` .

Variables are inherited from enclosing scopes.

Providing Default Values for Variables (%)

Within a `variables` section, keys named with percent sign (`%`) suffixes mean that the variable should be set only if it is undefined at the time it is processed. This can be used to provide defaults for variables that would otherwise be undefined, so that they may reliably be used in variable expansion or conditional processing.

Predefined Variables

Each GYP generator module provides defaults for the following variables:

- `OS` : The name of the operating system that the generator produces output for. Common values for values for `OS` are:

  - `'linux'`
  - `'mac'`
  - `'win'`

  But other values may be encountered and this list should not be considered exhaustive. The `gypd` (debug) generator module does not provide a predefined value for `OS` . When invoking GYP with the `gypd` module, if a value for `OS` is needed, it must be provided on the command line, such as `gyp -f gypd -DOS=mac` .

  GYP generators also provide defaults for these variables. They may be expressed in terms of variables used by the build system that they generate for, often in `$(VARIABLE)` format. For example, the GYP `PRODUCT_DIR` variable maps to the Xcode `BUILT_PRODUCTS_DIR` variable, so `PRODUCT_DIR` is defined by the Xcode generator as `$(BUILT_PRODUCTS_DIR)` .

- `EXECUTABLE_PREFIX` : A prefix, if any, applied to executable names. Usually this will be an empty string.
- `EXECUTABLE_SUFFIX` : A suffix, if any, applied to executable names. On Windows, this will be `.exe` , elsewhere, it will usually be an empty string.
- `INTERMEDIATE_DIR` : A directory that can be used to place intermediate build results in. `INTERMEDIATE_DIR` is only guaranteed to be accessible within a single target (See targets).

This variable is most useful within the context of rules and actions (See rules, See actions). Compare with `SHARED_INTERMEDIATE_DIR`.
- `PRODUCT_DIR`: The directory in which the primary output of each target, such as executables and libraries, is placed.
- `RULE_INPUT_ROOT`: The base name for the input file (e.g. "`foo`"). See Rules.
- `RULE_INPUT_EXT`: The file extension for the input file (e.g. "`.cc`"). See Rules.
- `RULE_INPUT_NAME`: Full name of the input file (e.g. "`foo.cc`"). See Rules.
- `RULE_INPUT_PATH`: Full path to the input file (e.g. "`/bar/foo.cc`"). See Rules.
- `SHARED_INTERMEDIATE_DIR`: A directory that can be used to place intermediate build results in, and have them be accessible to other targets. Unlike `INTERMEDIATE_DIR`, each target in a project, possibly spanning multiple `.gyp` files, shares the same `SHARED_INTERMEDIATE_DIR`.

The following additional predefined variables may be available under certain circumstances:

- `DEPTH`. When GYP is invoked with a `--depth` argument, when processing any `.gyp` file, `DEPTH` will be a relative path from the `.gyp` file to the directory specified by the `--depth` argument.

User-Defined Variables

A user-defined variable may be defined in terms of other variables, but not other variables that have definitions provided in the same scope.

Variable Expansions (<, >, <@, >@)

GYP provides two forms of variable expansions, "early" or "pre" expansions, and "late," "post," or "target" expansions. They have similar syntax, differing only in the character used to introduce them.

- Early expansions are introduced by a less-than (`<`) character. *Mnemonic: the arrow points to the left, earlier on a timeline.*
- Late expansions are introduced by a less-than (`>`) character. *Mnemonic: the arrow points to the right, later on a timeline.*

The difference the two phases of expansion is described in Early and Late Phases.

These characters were chosen based upon the requirement that they not conflict with the variable format used natively by build systems. While the dollar sign (`$`) is the most natural fit for variable expansions, its use was ruled out because most build systems already use that character for their own variable expansions. Using different characters means that no escaping mechanism was needed to differentiate between GYP variables and build system variables, and writing build system variables into GYP files is not cumbersome.

Variables may contain lists or strings, and variable expansions may occur in list or string context. There are variant forms of variable expansions that may be used to determine how each type of variable is to be expanded in each context.

- When a variable is referenced by `<(VAR)` or `>(VAR)`:
    - If `VAR` is a string, the variable reference within the string is replaced by variable's

string value.
- If `VAR` is a list, the variable reference within the string is replaced by a string containing the concatenation of all of the variable’s list items. Generally, the items are joined with spaces between each, but the specific behavior is generator-specific. The precise encoding used by any generator should be one that would allow each list item to be treated as a separate argument when used as program arguments on the system that the generator produces output for.

- When a variable is referenced by `<@(VAR)` or `>@(VAR)` :
  - The expansion must occur in list context.
  - The list item must be `’<@(VAR)’` or `’>@(VAR)’` exactly.
  - If `VAR` is a list, each of its elements are inserted into the list in which expansion is taking place, replacing the list item containing the variable reference.
  - If `VAR` is a string, the string is converted to a list which is inserted into the list in which expansion is taking place as above. The conversion into a list is generator-specific, but generally, spaces in the string are taken as separators between list items. The specific method of converting the string to a list should be the inverse of the encoding method used to expand list variables in string context, above.

GYP treats references to undefined variables as errors.

Command Expansions (<!, <!@)

Command expansions function similarly to variable expansions, but instead of resolving variable references, they cause GYP to execute a command at generation time and use the command’s output as the replacement. Command expansions are introduced by a less than and exclamation mark (`<!`).

In a command expansion, the entire string contained within the parentheses is passed to the system’s shell. The command’s output is assigned to a string value that may subsequently be expanded in list context in the same way as variable expansions if an `@` character is used.

In addition, command expansions (unlike other variable expansions) may include nested variable expansions. So something like this is allowed:

```
'variables' : [
  'foo': '<!(echo Build Date <!(date))',
],
```

expands to:

```
'variables' : [
  'foo': 'Build Date 02:10:38 PM Fri Jul 24, 2009 -0700 PDT',
],
```

You may also put commands into arrays in order to quote arguments (but note that you need to use a different string quoting character):

```
'variables' : [
  'files': '<!(["ls", "-1", "Filename With Spaces"])',
],
```

GYP treats command failures (as indicated by a nonzero exit status) during command expansion as errors.

Example

```
{
  'sources': [
    '!(echo filename with space.cc)',
  ],
  'libraries': [
    '!@(pkg-config --libs-only-l apr-1)',
  ],
}
```

might expand to:

```
{
  'sources': [
    'filename with space.cc',  # no @, expands into a single string
  ],
  'libraries': [  # @ was used, so there's a separate list item for each lib
    '-lapr-1',
    '-lpthread',
  ],
}
```

Conditionals

Conditionals use the same set of variables used for variable expansion. As with variable expansion, there are two phases of conditional evaluation:

- “Early” or “pre” conditional evaluation, introduced in conditions sections.
- “Late,” “post,” or “target” conditional evaluation, introduced in target_conditions sections.

The syntax for each type is identical, they differ only in the key name used to identify them and the timing of their evaluation. A more complete description of syntax and use is provided in conditions.

The difference the two phases of evaluation is described in Early and Late Phases.

Timing of Variable Expansion and Conditional Evaluation

Early and Late Phases

GYP performs two phases of variable expansion and conditional evaluation:

- The "early" or "pre" phase operates on conditions sections and the `<` form of variable expansions.
- The "late," "post," or "target" phase operates on target_conditions sections, the `>` form of variable expansions, and on the `!` form of [command expansions] (#Command_Expansions_(!,_!@)).

These two phases are provided because there are some circumstances in which each is desirable.

The "early" phase is appropriate for most expansions and evaluations. "Early" expansions and evaluations may be performed anywhere within any `.gyp` or `.gypi` file.

The "late" phase is appropriate when expansion or evaluation must be deferred until a specific section has been merged into target context. "Late" expansions and evaluations only occur within `targets` sections and their descendants. The typical use case for a late-phase expansion is to provide, in some globally-included `.gypi` file, distinct behaviors depending on the specifics of a target.

Example

Given this input:

```
{
  'target_defaults': {
    'target_conditions': [
      ['_type=="shared_library"', {'cflags': ['-fPIC']}],
    ],
  },
  'targets': [
    {
      'target_name': 'sharing_is_caring',
      'type': 'shared_library',
    },
    {
      'target_name': 'static_in_the_attic',
      'type': 'static_library',
    },
  ]
}
```

The conditional needs to be evaluated only in target context; it is nonsense outside of target context because no `_type` variable is defined. target_conditions allows evaluation to be deferred until after the targets sections are merged into their copies of target_defaults. The resulting targets, after "late" phase processing:

```
{
  'targets': [
    {
      'target_name': 'sharing_is_caring',
      'type': 'shared_library',
      'cflags': ['-fPIC'],
    },
    {
      'target_name': 'static_in_the_attic',
      'type': 'static_library',
    },
  ]
}
```

Expansion and Evaluation Performed Simultaneously

During any expansion and evaluation phase, both expansion and evaluation are performed simultaneously. The process for handling variable expansions and conditional evaluation within a dictionary is:

- Load automatic variables (those with leading underscores).
- If a variables section is present, recurse into its dictionary. This allows conditionals to be present within the `variables` dictionary.
- Load Variables user-defined variables from the variables section.
- For each string value in the dictionary, perform variable expansion and, if operating during the "late" phase, command expansions.
- Reload automatic variables and Variables user-defined variables because the variable expansion step may have resulted in changes to the automatic variables.
- If a conditions or target_conditions section (depending on phase) is present, recurse into its dictionary. This is done after variable expansion so that conditionals may take advantage of expanded automatic variables.
- Evaluate conditionals.
- Reload automatic variables and Variables user-defined variables because the conditional evaluation step may have resulted in changes to the automatic variables.
- Recurse into child dictionaries or lists that have not yet been processed.

One quirk of this ordering is that you cannot expect a variables section within a dictionary's conditional to be effective in the dictionary itself, but the added variables will be effective in any child dictionaries or lists. It is thought to be far more worthwhile to provide resolved automatic variables to conditional sections, though. As a workaround, to conditionalize variable values, place a conditions or target_conditions section within the variables section.

Dependencies and Dependents

In GYP, "dependents" are targets that rely on other targets, called "dependencies." Dependents declare their reliance with a special section within their target dictionary, dependencies.

Dependent Settings

It is useful for targets to “advertise” settings to their dependents. For example, a target might require that all of its dependents add certain directories to their include paths, link against special libraries, or define certain preprocessor macros. GYP allows these cases to be handled gracefully with “dependent settings” sections. There are three types of such sections:

- direct_dependent_settings, which advertises settings to a target’s direct dependents only.
- all_dependent_settings, which advertises settings to all of a target’s dependents, both direct and indirect.
- link_settings, which contains settings that should be applied when a target’s object files are used as linker input.

Furthermore, in some cases, a target needs to pass its dependencies’ settings on to its own dependents. This might happen when a target’s own public header files include header files provided by its dependency. export_dependent_settings allows a target to declare dependencies for which direct_dependent_settings should be passed through to its own dependents.

Dependent settings processing merges a copy of the relevant dependent settings dictionary from a dependency into its relevant dependent targets.

In most instances, direct_dependent_settings will be used. There are very few cases where all_dependent_settings is actually correct; in most of the cases where it is tempting to use, it would be preferable to declare export_dependent_settings. Most libraries and library_dirs sections should be placed within link_settings sections.

Example

Given:

```
{
  'targets': [
    {
      'target_name': 'cruncher',
      'type': 'static_library',
      'sources': ['cruncher.cc'],
      'direct_dependent_settings': {
        'include_dirs': ['.'],  # dependents need to find cruncher.h.
      },
      'link_settings': {
        'libraries': ['-lm'],  # cruncher.cc does math.
      },
    },
    {
      'target_name': 'cruncher_test',
      'type': 'executable',
      'dependencies': ['cruncher'],
      'sources': ['cruncher_test.cc'],
    },
  ],
}
```

After dependent settings processing, the dictionary for `cruncher_test` will be:

```
{
  'target_name': 'cruncher_test',
  'type': 'executable',
  'dependencies': ['cruncher'],  # implies linking against cruncher
  'sources': ['cruncher_test.cc'],
  'include_dirs': ['.']
  'libraries': ['-lm'],
},
```

If `cruncher` was declared as a `shared_library` instead of a `static_library`, the `cruncher_test` target would not contain `-lm`, but instead, `cruncher` itself would link against `-lm`.

Linking Dependencies

The precise meaning of a dependency relationship varies with the types of the targets at either end of the relationship. In GYP, a dependency relationship can indicate two things about how targets relate to each other:

- Whether the dependent target needs to link against the dependency.
- Whether the dependency target needs to be built prior to the dependent. If the former case is true, this case must be true as well.

The analysis of the first item is complicated by the differences between static and shared libraries.

- Static libraries are simply collections of object files (`.o` or `.obj`) that are used as inputs to a linker (`ld` or `link.exe`). Static libraries don't link against other libraries, they're collected together and used when eventually linking a shared library or executable.
- Shared libraries are linker output and must undergo symbol resolution. They must link against other libraries (static or shared) in order to facilitate symbol resolution. They may be used as libraries in subsequent link steps.
- Executables are also linker output, and also undergo symbol resolution. Like shared libraries, they must link against static and shared libraries to facilitate symbol resolution. They may not be reused as linker inputs in subsequent link steps.

Accordingly, GYP performs an operation referred to as "static library dependency adjustment," in which it makes each linker output target (shared libraries and executables) link against the static libraries it depends on, either directly or indirectly. Because the linkable targets link against these static libraries, they are also made direct dependents of the static libraries.

As part of this process, GYP is also able to remove the direct dependency relationships between two static library targets, as a dependent static library does not actually need to link against a dependency static library. This removal facilitates speedier builds under some build systems, as they are now free to build the two targets in parallel. The removal of this dependency is incorrect in some cases, such as when the dependency target contains rules or actions that generate header files required by the dependent target. In such cases, the dependency target, the one providing the side-effect files, must declare itself as a hard_dependency. This setting instructs GYP to not remove the dependency link between two static library targets in its generated output.

Loading Files to Resolve Dependencies

When GYP runs, it loads all `.gyp` files needed to resolve dependencies found in dependencies sections. These files are not merged into the files that reference them, but they may contain special sections that are merged into dependent target dictionaries.

Build Configurations

Explain this.

List Filters

GYP allows list items to be filtered by "exclusions" and "patterns." Any list containing string values in a dictionary may have this filtering applied. For the purposes of this section, a list modified by exclusions or patterns is referred to as a "base list", in contrast to the "exclusion list" and "pattern list" that operates on it.

- For a base list identified by key name `key`, the `key!` list provides exclusions.
- For a base list identified by key name `key`, the `key/` list provides regular expression pattern-based filtering.

Both `key!` and `key/` may be present. The `key!` exclusion list will be processed first, followed by the `key/` pattern list.

Exclusion lists are most powerful when used in conjunction with conditionals.

Exclusion Lists (!)

An exclusion list provides a way to remove items from the related list based on exact matching. Any item found in an exclusion list will be removed from the corresponding base list.

Example

This example excludes files from the `sources` based on the setting of the `OS` variable.

```
{
  'sources:' [
    'mac_util.mm',
    'win_util.cc',
  ],
  'conditions': [
    ['OS=="mac"', {'sources!': ['win_util.cc']}],
    ['OS=="win"', {'sources!': ['mac_util.cc']}],
  ],
}
```

Pattern Lists (/)

Pattern lists are similar to, but more powerful than, [exclusion lists](#Exclusion_Lists_(!)). Each item in a pattern list is itself a two-element list. The first item is a string, either ’include’ or ’exclude’, specifying the action to take. The second item is a string specifying a regular expression. Any item in the base list matching the regular expression pattern will either be included or excluded, based on the action specified.

Items in a pattern list are processed in sequence, and an excluded item that is later included will not be removed from the list (unless it is subsequently excluded again.)

Pattern lists are processed after [exclusion lists](#Exclusion_Lists_(!)), so it is possible for a pattern list to re-include items previously excluded by an exclusion list.

Nothing is actually removed from a base list until all items in an [exclusion list](#Exclusion_Lists_(!)) and pattern list have been evaluated. This allows items to retain their correct position relative to one another even after being excluded and subsequently included.

Example

In this example, a uniform naming scheme is adopted for platform-specific files.

```
{
  'sources': [
    'io_posix.cc',
    'io_win.cc',
    'launcher_mac.cc',
    'main.cc',
    'platform_util_linux.cc',
    'platform_util_mac.mm',
  ],
  'sources/': [
    ['exclude', '_win\\.cc$'],
  ],
  'conditions': [
    ['OS!="linux"', {'sources/': [['exclude', '_linux\\.cc$']]}],
    ['OS!="mac"', {'sources/': [['exclude', '_mac\\.cc|mm?$']]}],
    ['OS=="win"', {'sources/': [
      ['include', '_win\\.cc$'],
      ['exclude', '_posix\\.cc$'],
    ]}],
  ],
}
```

After the pattern list is applied, `sources` will have the following values, depending on the setting of `OS`:

- When `OS` is `linux`: `['io_posix.cc', 'main.cc', 'platform_util_linux.cc']`
- When `OS` is `mac`: `['io_posix.cc', 'launcher_mac.cc', 'main.cc', 'platform_util_mac.mm']`
- When `OS` is `win`: `['io_win.cc', 'main.cc', 'platform_util_win.cc']`

Note that when `OS` is `win`, the `include` for `_win.cc` files is processed after the `exclude` matching the same pattern, because the `sources/` list participates in merging during conditional evaluation just like any other list would. This guarantees that the `_win.cc` files, previously unconditionally excluded, will be re-included when `OS` is `win`.

Locating Excluded Items

In some cases, a GYP generator needs to access to items that were excluded by an [exclusion list](#Exclusion_Lists_(!)) or [pattern list](#Pattern_Lists_(/)). When GYP excludes items during processing of either of these list types, it places the results in an `_excluded` list. In the example above, when `OS` is `mac`, `sources_excluded` would be set to `['io_win.cc', 'platform_util_linux.cc']`. Some GYP generators use this feature to display excluded files in the project files they generate for the convenience of users, who may wish to refer to other implementations.

Processing Order

GYP uses a defined and predictable order to execute the various steps performed between loading files and generating output.

- Load files.
  - Load `.gyp` files. Merge any command-line includes into each `.gyp` file's root dictionary. As includes are found, load them as well and merge them into the scope in which the includes section was found.
  - Perform "early" or "pre" variable expansion and conditional evaluation.
  - Merge each target's dictionary into the `.gyp` file's root target_defaults dictionary.
  - Scan each target for dependencies, and repeat the above steps for any newly-referenced `.gyp` files not yet loaded.
- Scan each target for wildcard dependencies, expanding the wildcards.
- Process dependent settings. These sections are processed, in order:
  - all_dependent_settings
  - direct_dependent_settings
  - link_dependent_settings
- Perform static library dependency adjustment.
- Perform "late," "post," or "target" variable expansion and conditional evaluation on target dictionaries.

- Merge target settings into configurations as appropriate.
- Process exclusion and pattern lists.

Settings Keys

Settings that may appear anywhere

conditions

List of `condition` *items*

A `conditions` section introduces a subdictionary that is only merged into the enclosing scope based on the evaluation of a conditional expression. Each `condition` within a `conditions` list is itself a list of at least two items:

1. A string containing the conditional expression itself. Conditional expressions may take the following forms:
   - For string values, `var=="value"` and `var!="value"` to test equality and inequality. For example, `'OS=="linux"'` is true when the `OS` variable is set to `"linux"`.
   - For integer values, `var==value`, `var!=value`, `var<value`, `var<=value`, `var>=value`, and `var>value`, to test equality and several common forms of inequality. For example, `'chromium_code==0'` is true when the `chromium_code` variable is set to `0`.
   - It is an error for a conditional expression to reference any undefined variable.
2. A dictionary containing the subdictionary to be merged into the enclosing scope if the conditional expression evaluates to true.

These two items can be followed by any number of similar two items that will be evaluated if the previous conditional expression does not evaluate to true.

An additional optional dictionary can be appended to this sequence of two items. This optional dictionary will be merged into the enclosing scope if none of the conditional expressions evaluate to true.

Within a `conditions` section, each item is processed sequentially, so it is possible to predict the order in which operations will occur.

There is no restriction on nesting `conditions` sections.

`conditions` sections are very similar to `target_conditions` sections. See target_conditions.

Example

```
{
  'sources': [
    'common.cc',
  ],
  'conditions': [
    ['OS=="mac"', {'sources': ['mac_util.mm']}],
    ['OS=="win"', {'sources': ['win_main.cc']}, {'sources': ['posix_main.cc']]
    ['OS=="mac"', {'sources': ['mac_impl.mm']},
     'OS=="win"', {'sources': ['win_impl.cc']},
     {'sources': ['default_impl.cc']}
    ],
  ],
}
```

Given this input, the `sources` list will take on different values based on the `OS` variable.

- If `OS` is `"mac"`, `sources` will contain `['common.cc', 'mac_util.mm', 'posix_main.cc', 'mac_impl.mm']`.
- If `OS` is `"win"`, `sources` will contain `['common.cc', 'win_main.cc', 'win_impl.cc']`.
- If `OS` is any other value such as `"linux"`, `sources` will contain `['common.cc', 'posix_main.cc', 'default_impl.cc']`.