Universal Ctags Documentation

Release 0.3.0

Universal Ctags Team

25 December 2018

Contents

Version Draft

Authors universal-ctags developers

Web Page https://ctags.io/

Universal-ctags was created to continue the development of Darren Hiebert's Exuberant-ctags after activity on that project unfortunately stalled.

Reza Jelveh <reza.jelveh@gmail.com> initially created a personal fork on GitHub and as interest and participation grew it was decided to move development to a dedicated GitHub organization.

The goal of this project is to maintain a common/unified space where people interested in improving ctags can work together.

This guide is primarily intended for developers. Users should first consult the ctags.1 man page.

This is a draft document. Proofreading and pull-requests are welcome!

CHAPTER 1

Introduced changes

Maintainer Masatake YAMATO <yamato@redhat.com>

Table of contents

Many changes have been introduced in Universal-ctags. Use git-log to review changes not enumerated here, especially in language parsers.

1.1 Importing changes from Exuberant-ctags

See “Exuberant-ctags” in “Tracking other projects” for detailed information regarding imported changes.

Some changes have also been imported from Fedora and Debian.

1.2 Parser related changes

1.2.1 Fully rewritten parsers

- C (see *The new C/C++ parser*)
- C++ (see *The new C/C++ parser*)
- Python (see *The new Python parser*)
- HTML (see *The new HTML parser*)
- Tcl (see *The new Tcl parser*)
- ITcl (see *The new Tcl parser*)

1.2.2 New parsers

The following parsers have been added:

- Ada
- AnsiblePlaybook *libyaml*
- Asciidoc
- Autoconf
- Automake
- AutoIt
- Clojure
- CMake *optlib*
- CSS
- Ctags option library *optlib*
- CUDA
- D
- DBusIntrospect *libxml*
- Diff
- DTD
- DTS
- Elm *optlib*
- Falcon
- Gdbinit script *optlib*
- Glade *libxml*
- Go
- JavaProperties
- JSON
- GNU linker script(LdScript)
- Man page *optlib*
- Markdown *optlib*

- Maven2 *libxml*
- M4
- ObjectiveC
- Passwd *optlib*
- PuppetManifest *optlib*
- Perl6
- Pod *optlib*
- PropertyList(plist) *libxml*
- Protobuf
- PythonLoggingConfig
- QemuHX *optlib*
- QtMoc
- R
- RelaxNG *libxml*
- ReStructuredText
- Robot
- RpmSpec
- Rust
- SystemdUnit
- SystemVerilog
- SVG *libxml*
- TclOO (see *The new Tcl parser*)
- TTCN
- WindRes
- XSLT v1.0 *libxml*
- Yacc
- Yaml *libyaml*
- YumRepo
- Zephir
- Myrddin
- RSpec *optlib*

See "Option library" for details on *optlib*. Libxml2 is required to use the parser(s) marked with *libxml*. Libyaml is required to use the parser(s) marked with *libyaml*.

TIPS: you can list newly introduced parsers if you also have Exuberant-ctags installed with following command line:

```
$ diff -ruN <(universal-ctags --list-languages) <(exuberant-ctags --list-
↪languages)  | grep '^[-+]'
```

1.2.3 Heavily improved parsers

- Ant (rewritten with *libxml*)
- PHP
- Verilog

1.2.4 *F* kind usage

`F` is used as a kind letter for file kind in Exuberant-ctags; the `F` was hard-coded in ctags internal. However, we found some built-in parsers including Ruby uses `F` for their own purpose. So if you find a tag having `F` as a kind letter, you cannot say what it is well: a file name or something peculiar in the language. Long kind description strings may help you but we are not sure all tools utilizing `tags` file refer the long kind description strings.

Universal-ctags disallows parsers to use `F` their own purpose in both built-in and optlib parsers.

`F` in built-in parsers are replaced as follows:

Language	Long description	Replacement
ObjectiveC	field	E
Ruby	singletonMethod	S
Rust	method	P
SQL	field	E

1.3 New and extended options

1.3.1 Wildcard in options

For the purpose of gathering as much as information as possible from source code the "wildcard"(`*`) option value has been introduced.

`--extras=*`

Enables all extra tags.

`--fields=*`

Enables all available fields.

`--<LANG>-kinds=*`

Enables all available kinds for `LANG`.

`--kinds-<LANG>=*`

Alternative representation of `--<LANG>-kinds=*`.

`--all-kinds=SPEC`

Applies SPEC as kinds to all available language parsers.

`--all-kinds=*`

Enables all available kinds for all available language parsers.

1.3.2 Long names in kinds, fields, and extra options

A letter is used for specifying a kind, a field, or an extra entry. In Universal-ctags a name can also be used.

Surround the name with braces (*{* and *}*) in values assigned to the options, `--kind-<LANG>=`, `--fields=`, or `--extras=`.

```
$ ./ctags --kinds-C=+L-d ...
```

This command line uses the letters, *L* for enabling the label kind and *d* for disabling the macro kind of C. The command line can be rewritten with the associated names.

```
$ ./ctags --kinds-C='+{label}-{macro}' ...
```

The quotes are needed because braces are interpreted as meta characters by the shell.

The available names can be listed with `--list-kinds-full`, `--list-fields`, or `--list-extras`.

1.3.3 Notice messages and `--quiet`

There were 3 classes of message in ctags:

fatal

A critical error has occurred and ctags aborts the execution.

warning

An error has occurred but ctags continues the execution.

verbose

Mainly used for debugging purposes.

notice is a new class of message. It is less important than *warning* but more important for users than *verbose*.

Generally the user can ignore *notice* class messages and `--quiet` can be used to disable them.

1.3.4 `--input-encoding=ENCODING` and `--output-encoding=ENCODING`

Japanese programmers sometimes use the Japanese language in source code comments. Of course, it is not limited to Japanese. People may use their own native language and in such cases encoding becomes an issue.

ctags doesn't consider the input encoding; it just reads input as a sequence of bytes and uses them as is when writing tags entries.

On the other hand Vim does consider input encoding. When loading a file, Vim converts the file contents into an internal format with one of the encodings specified in its *fileencodings* option.

As a result of this difference, Vim cannot always move the cursor to the definition of a tag as users expect when attempting to match the patterns in a tags file.

The good news is that there is a way to notify Vim of the encoding used in a tags file with the `TAG_FILE_ENCODING` pseudo tag.

Two new options have been introduced (`--input-encoding=IN` and `--output-encoding=OUT`).

Using the encoding specified with these options ctags converts input from `IN` to `OUT`. ctags uses the converted strings when writing the pattern parts of each tag line. As a result the tags output is encoded in `OUT` encoding.

In addition `OUT` is specified at the top the tags file as the value for the `TAG_FILE_ENCODING` pseudo tag. The default value of `OUT` is UTF-8.

NOTE: Converted input is NOT passed to language parsers. The parsers still deal with input as a byte sequence.

With `--input-encoding-<LANG>=IN`, you can specify a specific input encoding for `LANG`. It overrides the global default value given with `--input-encoding`.

The example usage can be found in *Tmain/{input,output}-encoding-option.d*.

Acceptable `IN` and `OUT` values can be listed with *iconv -l* or *iconv –list*. It is platform dependant.

To enable the option, libiconv is needed on your platform. In addition `--enable-iconv` must be given to configure before making ctags. On Windows mingw32, you must specify `WITH_ICONV=yes` like this:

```
C:\dev\ctags>mingw32-make -f mk_mingw.mak WITH_ICONV=yes
```

`--list-features` helps you to know whether your ctags executable links to libiconv or not. You will find `iconv` in the output if it links to.

1.3.5 Extra tag entries (`--extras`)

`--extra` option in Exuberant-ctags is renamed to `--extras` (plural) in Universal-ctags for making consistent with `--kinds-<LANG>` and `--fields`.

These extra tag entries are newly introduced.

`F`

Equivalent to –file-scope.

`p`

Include pseudo tags.

1.3.6 Options for inspecting ctags internals

Exuberant-ctags provides a way to inspect its internals via `--list-kinds`, `--list-languages`, and `--list-maps`.

This idea has been expanded in Universal-ctags with `--list-kinds-full`, `--list-map-extensions`, `--list-extras`, `--list-features`, `--list-fields`, `--list-map-patterns`, and `--list-pseudo-tags` being added.

The original three `--list-` options are not changed for compatibility reasons, however, the newly introduced options are recommended for all future use.

By default, interactive use is assumed and ctags tries aligning the list output in columns for easier reading.

When `--machinable` is given before a `--list-` option, ctags outputs the list in a format more suitable for processing by scripts. Tab characters are used as separators between columns. The alignment of columns is never considered when `--machinable` is given.

Currently only `--list-extras`, `--list-fields` and `--list-kinds-full` support `--machinable` output.

These new `--list-` options also print a column header, a line representing the name of each column. The header may help users and scripts to understand and recognize the columns. Ignoring the column header is easy because it starts with a # character.

`--with-list-header=no` suppresses output of the column header.

1.3.7 Kinds synchronization

In Universal-ctags, as in Exuberant-ctags, most kinds are parser local; enabling (or disabling) a kind in a parser has no effect on kinds in any other parsers even those with the same name and/or letter.

However, there are exceptions, such as C and C++ for example. C++ can be considered a language extended from C. Therefore it is natural that all kinds defined in the C parser are also defined in the C++ parser. Enabling a kind in the C parser also enables a kind having the same name in the C++ parser, and vice versa.

A kind group is a group of kinds satisfying the following conditions:

1. Having the same name and letter, and
2. Being synchronized with each other

A master parser manages the synchronization of a kind group. The *MASTER* column of `--list-kinds-full` shows the master parser of the kind.

Internally, a state change (enabled or disabled with `--kind-<LANG>=[+|-]...`) of a kind in a kind group is reported to its master parser as an event. Then the master parser updates the state of all kinds in the kind group as specified with the option.

```
$ ./ctags --list-kinds-full=C++
#LETTER NAME            ENABLED  REFONLY NROLES MASTER     DESCRIPTION
d       macro           on       FALSE   1      C          macro definitions
...
$ ./ctags --list-kinds-full=C
#LETTER NAME            ENABLED  REFONLY NROLES MASTER     DESCRIPTION
d       macro           on       FALSE   1      C          macro definitions
...
```

The example output indicates that the *d* kinds of both the C++ and C parsers are in the same group and that the *C* parser manages the group.

```
$ ./ctags --kinds-C++=-d --list-kinds-full=C | head -2
#LETTER NAME            ENABLED  REFONLY NROLES MASTER     DESCRIPTION
d       macro           off      FALSE   1      C          macro definitions
$ ./ctags --kinds-C=-d --list-kinds-full=C | head -2
#LETTER NAME            ENABLED  REFONLY NROLES MASTER     DESCRIPTION
d       macro           off      FALSE   1      C          macro definitions
$ ./ctags --kinds-C++=-d --list-kinds-full=C++ | head -2
#LETTER NAME            ENABLED  REFONLY NROLES MASTER     DESCRIPTION
d       macro           off      FALSE   1      C          macro definitions
$ ./ctags --kinds-C=-d --list-kinds-full=C++ | head -2
#LETTER NAME            ENABLED  REFONLY NROLES MASTER     DESCRIPTION
d       macro           off      FALSE   1      C          macro definitions
```

In the above example, the *d* kind is disabled via C or C++. Disabling a *d* kind via one language disables the *d* kind for the other parser, too.

1.3.8 `--put-field-prefix` options

Some fields are newly introduced in Universal-ctags and more will be introduced in the future. Other tags generators may also introduce their own fields.

In such a situation there is a concern about conflicting field names; mixing tags files generated by multiple tags generators including Universal-ctags is difficult.

`--put-field-prefix` provides a workaround for this use case. When `--put-field-prefix` is given, ctags adds "UCTAGS" as a prefix to newly introduced fields.

```
$ cat /tmp/foo.h
#include <stdio.h>
$ ./ctags -o - --extras=+r --fields=+r /tmp/foo.h
stdio.h     /tmp/foo.h      /^#include <stdio.h>/;" h       roles:system
$ ./ctags --put-field-prefix -o - --extras=+r --fields=+r /tmp/foo.h
stdio.h     /tmp/foo.h      /^#include <stdio.h>/;" h       UCTAGSroles:system
```

In this example, `roles` is prefixed.

1.3.9 `--maxdepth` option

`--maxdepth` limits the depth of directory recursion enabled with the `-R` option.

1.3.10 `--map-<LANG>` option

`--map-<LANG>` is newly introduced to control the file name to language mappings (langmap) with finer granularity than `--langmap` allows.

A langmap entry is defined as a pair; the name of the language and a file name extension (or pattern).

Here we use "spec" as a generic term representing both file name extensions and patterns.

`--langmap` maps specs to languages exclusively:

```
$ ./ctags --langdef=FOO --langmap=FOO:+.ABC \
          --langdef=BAR --langmap=BAR:+.ABC  \
          --list-maps | grep '\*.ABC$'
BAR       *.ABC
```

Though language *FOO* is added before *BAR*, only *BAR* is set as a handler for the spec **.ABC*.

Universal-ctags enables multiple parsers to be configured for a spec. The appropriate parser for a given input file can then be chosen by a variety of internal guessing strategies (see "Choosing a proper parser in ctags").

Let's see how specs can be mapped non-exclusively with `--map-<LANG>`:

```
% ./ctags --langdef=FOO --map-FOO=+.ABC \
          --langdef=BAR --map-BAR=+.ABC \
          --list-maps | grep '\*.ABC$'
FOO       *.ABC
BAR       *.ABC
```

Both *FOO* and *BAR* are registered as handlers for the spec **.ABC*.

`--map-<LANG>` can also be used for removing a langmap entry.:

```
$ ./ctags --langdef=FOO --map-FOO=+.ABC \
          --langdef=BAR --map-BAR=+.ABC \
          --map-FOO=-.ABC --list-maps | grep '\*.ABC$'
BAR       *.ABC

$ ./ctags --langdef=FOO --map-FOO=+.ABC \
          --langdef=BAR --map-BAR=+.ABC \
          --map-BAR=-.ABC --list-maps | grep '\*.ABC$'
FOO       *.ABC

$./ctags --langdef=FOO --map-FOO=+.ABC \
         --langdef=BAR --map-BAR=+.ABC \
         --map-BAR=-.ABC --map-FOO=-.ABC  --list-maps | grep '\*.ABC$'
(NOTHING)
```

`--langmap` provides a way to manipulate the langmap in a spec-centric manner and `--map-<LANG>` provides a way to manipulate the langmap in a parser-centric manner.

1.3.11 Guessing parser from file contents (`-G` option)

See "Choosing a proper parser in ctags" section.

1.3.12 Enabling/disabling pseudo tags (`--pseudo-tags` option)

Each pseudo tag can be enabled/disabled with `--pseudo-tags`.

```
--pseudo-tags=+ptag
--pseudo-tags=-ptag
```

When prefixed with +, the pseudo tag specified as `ptag` is enabled. When prefixed with -, the pseudo tag is disabled. `--list-pseudo-tags` shows all recognized ptag names.

All pseudo tags are enabled if * is given as the value of ptag like:

```
--pseudo-tags='*'
```

All pseudo tags are disabled if no option value is given to `--pseudo-tags` like:

```
--pseudo-tags=
```

To specify only a single pseudo tag, omit the sign:

```
--pseudo-tags=ptag
```

1.3.13 JSON output

Experimental JSON output has been added. `--output-format` can be used to enable it.

```
$ ./ctags --output-format=json --fields=-s /tmp/foo.py
{"_type": "tag", "name": "Foo", "path": "/tmp/foo.py", "pattern": "/^class Foo:$/",
↪ "kind": "class"}
{"_type": "tag", "name": "doIt", "path": "/tmp/foo.py", "pattern": "/^    def␣
↪doIt():$/", "kind": "member"}
```

See *JSON output* for more details.

1.3.14 “always” and “never” as an argument for –tag-relative

Even if “yes” is specified as an option argument for –tag-relative, absolute paths are used in tags output if an input is given as an absolute path. This behavior is expected in exuberant-ctags as written in its man-page.

In addition to “yes” and “no”, universal-ctags takes “never” and “always”.

If “never” is given, absolute paths are used in tags output regardless of the path representation for input file(s). If “always” is given, relative paths are used always.

1.3.15 Defining a macro in CPreProcessor input

Newly introduced `-D` option extends the function provided by `-I` option.

`-D` emulates the behaviour of the corresponding gcc option: it defines a C preprocessor macro. All types of macros are supported, including the ones with parameters and variable arguments. Stringification, token pasting and recursive macro expansion are also supported.

`-I` is now simply a backward-compatible syntax to define a macro with no replacement.

Some examples follow.

```
$ ctags ... -D IGNORE_THIS ...
```

With this commandline the following C/C++ input

```
int IGNORE_THIS a;
```

will be processed as if it was

```
int a;
```

Defining a macro with parameters uses the following syntax:

```
$ ctags ... -D "foreach(arg)=for(arg;;)" ...
```

This example defines *for(arg;;)* as the replacement *foreach(arg)*. So the following C/C++ input

```
foreach(char * p,pointers)
{

}
```

is processed in new C/C++ parser as:

```
for(char * p;;)
{

}
```

and the p local variable can be extracted.

The previous commandline includes quotes since the macros generally contain characters that are treated specially by the shells. You may need some escaping.

Token pasting is performed by the ## operator, just like in the normal C preprocessor.

```
$ ctags ... -D "DECLARE_FUNCTION(prefix)=int prefix ## Call();"
```

So the following code

```
DECLARE_FUNCTION(a)
DECLARE_FUNCTION(b)
```

will be processed as

```
int aCall();
int bCall();
```

Macros with variable arguments use the gcc __VA_ARGS__ syntax.

```
$ ctags ... -D "DECLARE_FUNCTION(name,...)=int name(__VA_ARGS__);"
```

So the following code

```
DECLARE_FUNCTION(x,int a,int b)
```

will be processed as

```
int x(int a,int b);
```

1.3.16 `--_interactive` Mode

A new `--_interactive` option launches a JSON based command REPL which can be used to control ctags generation programmatically.

See *–_interactive Mode* for more details.

`--_interactive=sandbox` adds up seccomp filter. See *sandbox submode* for more details.

1.3.17 Defining a kind

A new `--kinddef-<LANG>=letter,name,description` option reduces the typing defining a regex pattern with `--regex-<LANG>=`, and keeps the consistency of dynamically defined kinds in a language.

A kind letter defined with `--kinddef-<LANG>` can be referred in `--kinddef-<LANG>`.

Previously you had to write in your optlib:

```
--regex-elm=/^([[:lower:]_][[:alnum:]_]*)[^=]*=$/\1/f,function,Functions/
↪{scope=set}
--regex-elm=/^[[:blank:]]+([[:lower:]_][[:alnum:]_]*)[^=]*=$/\1/f,function,
↪Functions/{scope=ref}
```

With new `--kinddef-<LANG>` you can write the same things like:

```
--kinddef-elm=f,function,Functions
--regex-elm=/^([[:lower:]_][[:alnum:]_]*)[^=]*=$/\1/f/{scope=set}
--regex-elm=/^[[:blank:]]+([[:lower:]_][[:alnum:]_]*)[^=]*=$/\1/f/{scope=ref}
```

We can say now "kind" is a first class object in Universal-ctags.

1.3.18 Defining an extra

A new `--_extradef-<LANG>=name,description` option allows you to defining a parser own extra which turning on and off can be referred from a regex based parser for `<LANG>`.

See *Conditional tagging with extras* for more details.

1.3.19 Defining a subparser

Basic

About the concept of subparser, see *Tagging definitions of higher(upper) level language (sub/base)*.

With `base` long flag of *–langdef=<LANG>* option, you can define a subparser for a specified base parser. Combining with `--kinddef-<LANG>` and `--regex-<KIND>` options, you can extend an existing parser without risk of kind confliction.

Let's see an example.

input.c

```
static int set_one_prio(struct task_struct *p, int niceval, int error)
{
}

SYSCALL_DEFINE3(setpriority, int, which, int, who, int, niceval)
{
        ...;
}
```

```
$./ctags --options=NONE  -x --_xformat="%20N %10K %10l"  -o - input.c
ctags: Notice: No options will be read from files or environment
        set_one_prio    function          C
     SYSCALL_DEFINE3    function          C
```

C parser doesn't understand that *SYSCALL_DEFINE3* is a macro for defining an entry point for a system.

Let's define *linux* subparser which using C parser as a base parser:

```
$ cat linux.ctags
--langdef=linux{base=C}
--kinddef-linux=s,syscall,system calls
--regex-linux=/SYSCALL_DEFINE[0-9]\(([^, )]+)[\),]*/\1/s/
```

The output is change as follows with *linux* parser:

```
$ ./ctags --options=NONE --options=./linux.ctags -x --_xformat="%20N %10K %10l"  -
↪o - input.c
ctags: Notice: No options will be read from files or environment
         setpriority    syscall      linux
        set_one_prio   function          C
     SYSCALL_DEFINE3   function          C
```

setpriority is recognized as a *syscall* of *linux*.

Using only *–regex-C=...* you can capture *setpriority*. However, there were concerns about kind confliction; when introducing a new kind with *–regex-C=...*, you cannot use a letter and name already used in C parser and *–regex-C=...* options specified in the other places.

You can use a newly defined subparser as a new namespace of kinds. In addition you can enable/disable with the subparser usable *–languages=[+|-]* option:

Directions

As explained in *Tagging definitions of higher(upper) level language (sub/base)*, you can choose direction(s) how a base parser and a guest parser work together with long flags putting after *–langdef=Foo{base=Bar}*.

C level notation	Command line long flag
SUBPARSER_BASE_RUNS_SUB	shared
SUBPARSER_SUB_RUNS_BASE	dedicated
SUBPARSER_BASE_RUNS_SUB	bidirectional

Let's see actual difference of behaviors.

The examples are taken from #1409 submitted by @sgraham on github Universal-ctags repository.

input.cc and *input.mojom* are input files, and have the same contents:

```
 ABC();
int main(void)
{
}
```

C++ parser can capture *main* as a function. Mojom subparser defined in the later runs on C++ parser and is for capturing *ABC*.

shared combination

{shared} is specified, for *input.cc*, both tags capture by C++ parser and mojom parser are recorded to tags file. For *input.mojom*, only tags captured by mojom parser are recorded to tags file.

mojom-shared.ctags:

```
--langdef=mojom{base=C++}{shared}
--map-mojom=+.mojom
--kinddef-mojom=f,function,functions
--regex-mojom=/^[ ]+([a-zA-Z]+)\(/\1/f/
```

tags for *input.cc*:

```
ABC input.cc        /^ ABC();$/;"    f       language:mojom
main        input.cc        /^int main(void)$/;"    f       language:C++    ␣
↪typeref:typename:int
```

tags for *input.mojom*:

```
ABC    input.mojom     /^ ABC();$/;"    f       language:mojom
```

Mojom parser uses C++ parser internally but tags captured by C++ parser are dropped in the output.

{shared} is the default behavior. If none of *{shared}*, *{dedicated}*, nor *{bidirectional}* is specified, it implies *{shared}*.

dedicated combination

{dedicated} is specified, for *input.cc*, only tags capture by C++ parser are recorded to tags file. For *input.mojom*, both tags capture by C++ parser and mojom parser are recorded to tags file.

mojom-dedicated.ctags:

```
--langdef=mojom{base=C++}{dedicated}
--map-mojom=+.mojom
--kinddef-mojom=f,function,functions
--regex-mojom=/^[ ]+([a-zA-Z]+)\(/\1/f/
```

tags for *input.cc*:

```
main        input.cc        /^int main(void)$/;"    f       language:C++    ␣
↪typeref:typename:int
```

tags for *input.mojom*:

```
ABC input.mojom     /^ ABC();$/;"    f       language:mojom
main        input.mojom     /^int main(void)$/;"    f       language:C++    ␣
↪typeref:typename:int
```

Mojom parser works only when *.mojom* file is given as input.

bidirectional combination

{bidirectional} is specified, both tags capture by C++ parser and mojom parser are recorded to tags file for either input *input.cc* and *input.mojom*.

mojom-bidirectional.ctags:

```
--langdef=mojom{base=C++}{bidirectional}
--map-mojom=+.mojom
--kinddef-mojom=f,function,functions
--regex-mojom=/^[ ]+([a-zA-Z]+)\(/\1/f/
```

tags for *input.cc*:

```
ABC input.cc        /^ ABC();$/;"    f       language:mojom
main        input.cc        /^int main(void)$/;"    f       language:C++    ␣
↪typeref:typename:int
```

tags for *input.mojom*:

```
ABC input.cc        /^ ABC();$/;"    f       language:mojom
main        input.cc        /^int main(void)$/;"    f       language:C++    ␣
↪typeref:typename:int
```

Listing subparsers

Subparsers can be listed with `--list-subparser`:

```
$ ./ctags --options=NONE --options=./linux.ctags --list-subparsers=C
ctags: Notice: No options will be read from files or environment
#NAME                          BASEPARSER           DIRECTION
linux                          C                    base => sub {shared}
```

1.4 Changes to the tags file format

1.4.1 Truncating the pattern for long input lines

To prevent generating overly large tags files, a pattern field is truncated, by default, when its size exceeds 96 bytes. A different limit can be specified with `--pattern-length-limit=N`.

The truncation avoids cutting in the middle of a UTF-8 code point spanning multiple bytes to prevent writing invalid byte sequences from valid input files. This handling allows for an extra 3 bytes above the configured limit in the worse case of a 4 byte code point starting right before the limit. Please also note that this handling is fairly naive and fast, and although it is resistant against any input, it requires a valid input to work properly; it is not guaranteed to work as the user expects when dealing with partially invalid UTF-8 input. This also partially affect non-UTF-8 input, if the byte sequence at the truncation length looks like a multibyte UTF-8 sequence. This should however be rare, and in the worse case will lead to including up to an extra 3 bytes above the limit.

An input source file with long lines and multiple tag matches per line can generate an excessively large tags file with an unconstrained pattern length. For example, running ctags on a minified JavaScript source file often exhibits this behaviour.

1.5 Reference tags

Traditionally ctags collects the information for locating where a language object is DEFINED.

In addition Universal-ctags supports reference tags. If the extra-tag `r` is enabled, Universal-ctags also collects the information for locating where a language object is REFERENCED. This feature was proposed by @shigio in #569 for GNU GLOBAL.

Here are some examples. Here is the target input file named reftag.c.

```
#include <stdio.h>
#include "foo.h"
#define TYPE point
struct TYPE { int x, y; };
TYPE p;
#undef TYPE
```

Traditional output:

```
$ ./ctags -o - reftag.c
TYPE        reftag.c        /^#define TYPE /;"      d       file:
TYPE        reftag.c        /^struct TYPE { int x, y; };$/;"        s       file:
p   reftag.c        /^TYPE p;$/;"   v       typeref:typename:TYPE
x   reftag.c        /^struct TYPE { int x, y; };$/;"        m       struct:TYPE
↪ typeref:typename:int    file:
y   reftag.c        /^struct TYPE { int x, y; };$/;"        m       struct:TYPE
↪ typeref:typename:int    file:
```

Output with the extra-tag `r` enabled:

```
$ ./ctags --list-extras | grep ^r
r   Include reference tags  off
$ ./ctags -o - --extras=+r reftag.c
```

(continues on next page)

(continued from previous page)

```
TYPE        reftag.c        /^#define TYPE /;"      d       file:
TYPE        reftag.c        /^#undef TYPE$/;"       d       file:
TYPE        reftag.c        /^struct TYPE { int x, y; };$/;"        s       file:
foo.h       reftag.c        /^#include "foo.h"/;"   h
p   reftag.c        /^TYPE p;$/;"   v       typeref:typename:TYPE
stdio.h     reftag.c        /^#include <stdio.h>/;" h
x   reftag.c        /^struct TYPE { int x, y; };$/;"        m       struct:TYPE
↪ typeref:typename:int    file:
y   reftag.c        /^struct TYPE { int x, y; };$/;"        m       struct:TYPE
↪ typeref:typename:int    file:
```

#undef X and two *#include* are newly collected.

"roles" is a newly introduced field in Universal-ctags. The field named is for recording how a tag is referenced. If a tag is definition tag, the roles field has "def" as its value.

Universal-ctags prints the role information when the *r* field is enabled with `--fields=+r`.

```
$  ./ctags -o - --extras=+r --fields=+r reftag.c
TYPE        reftag.c        /^#define TYPE /;"      d       file:
TYPE        reftag.c        /^#undef TYPE$/;"       d       file:   roles:undef
TYPE        reftag.c        /^struct TYPE { int x, y; };$/;"        s       file:
↪ roles:def
foo.h       reftag.c        /^#include "foo.h"/;"   h       roles:local
p   reftag.c        /^TYPE p;$/;"   v       typeref:typename:TYPE   roles:def
stdio.h     reftag.c        /^#include <stdio.h>/;" h       roles:system
x   reftag.c        /^struct TYPE { int x, y; };$/;"        m       struct:TYPE
↪ typeref:typename:int    file:   roles:def
y   reftag.c        /^struct TYPE { int x, y; };$/;"        m       struct:TYPE
↪ typeref:typename:int    file:   roles:def
```

The *Reference tag marker* field, `R`, is a specialized GNU global requirement; D is used for the traditional definition tags, and R is used for the new reference tags. The field can be used only with `--_xformat`.

```
$ ./ctags -x --_xformat="%R %-16N %4n %-16F %C" --extras=+r reftag.c
D TYPE                3 reftag.c          #define TYPE point
D TYPE                4 reftag.c          struct TYPE { int x, y; };
D p                   5 reftag.c          TYPE p;
D x                   4 reftag.c          struct TYPE { int x, y; };
D y                   4 reftag.c          struct TYPE { int x, y; };
R TYPE                6 reftag.c          #undef TYPE
R foo.h               2 reftag.c          #include "foo.h"
R stdio.h             1 reftag.c          #include <stdio.h>
```

See *Customizing xref output* for more details about this option.

Although the facility for collecting reference tags is implemented, only a few parsers currently utilize it. All available roles can be listed with `--list-roles`:

```
$ ./ctags --list-roles
#LANGUAGE      KIND(L/N)         NAME                ENABLED DESCRIPTION
SystemdUnit    u/unit            Requires            on      referred in Requires
↪key
SystemdUnit    u/unit            Wants               on      referred in Wants key
SystemdUnit    u/unit            After               on      referred in After key
SystemdUnit    u/unit            Before              on      referred in Before key
SystemdUnit    u/unit            RequiredBy          on      referred in
↪RequiredBy key
SystemdUnit    u/unit            WantedBy            on      referred in WantedBy
↪key
Yaml           a/anchor          alias               on      alias
DTD            e/element         attOwner            on      attributes owner
```

(continues on next page)

(continued from previous page)

```
Automake        c/condition       branched            on      used for branching
Cobol           S/sourcefile      copied              on      copied in source file
Maven2          g/groupId         dependency          on      dependency
DTD             p/parameterEntity elementName         on      element names
DTD             p/parameterEntity condition           on      conditions
LdScript        s/symbol          entrypoint          on      entry points
LdScript        i/inputSection    discarded           on      discarded when linking
...
```

The first column shows the name of the parser. The second column shows the letter/name of the kind. The third column shows the name of the role. The fourth column shows whether the role is enabled or not. The fifth column shows the description of the role.

You can define a role in an optlib parser for capturing reference tags. See *Capturing reference tags* for more details.

Currently ctags doesn't provide the way for disabling a specified role.

1.6 Automatic parser selection

See "Choosing a proper parser in ctags" section.

1.6.1 Incompatible changes to file name pattern and extension handling

When guessing a proper parser for a given input file, Exuberant-ctags tests file name patterns AFTER file extensions (e-order). Universal-ctags does this differently; it tests file name patterns BEFORE file extensions (u-order).

This incompatible change is introduced to deal with the following situation: "build.xml" is an input file. The Ant parser declares it handles a file name pattern "build.xml" and another parser, Foo, declares it handles a file extension "xml".

Which parser should be used for parsing the input? The user may want to use the Ant parser because the pattern it declares is more specific than the extension Foo declares. However, in e-order, the other parser, Foo, is chosen.

So Universal-ctags uses the u-order even though it introduces an incompatibility.

1.7 Pseudo tags

Pseudo tags are used to add meta data to a tags file. Universal-ctags will utilize pseudo tags aggressively.

Universal-ctags is not mature yet; there is a possibility that incompatible changes will be introduced. As a result tools reading a tags file may not work as expected.

To mitigate this issue pseudo tags are employed to make a tags file more self-descriptive. We hope some of the incompatibilities can be overcome in client tools by utilizing this approach.

Example output:

```
$ ./ctags -o - --extras=p --pseudo-tags='TAG_KIND_DESCRIPTION' foo.c
!_TAG_KIND_DESCRIPTION!C    L,label /goto label/
!_TAG_KIND_DESCRIPTION!C    c,class /classes/
!_TAG_KIND_DESCRIPTION!C    d,macro /macro definitions/
!_TAG_KIND_DESCRIPTION!C    e,enumerator    /enumerators (values inside an␣
↪enumeration)/
!_TAG_KIND_DESCRIPTION!C    f,function      /function definitions/
!_TAG_KIND_DESCRIPTION!C    g,enum  /enumeration names/
!_TAG_KIND_DESCRIPTION!C    h,header        /included header files/
```

(continues on next page)

(continued from previous page)

```
!_TAG_KIND_DESCRIPTION!C    l,local /local variables/
!_TAG_KIND_DESCRIPTION!C    m,member        /class, struct, and union members/
!_TAG_KIND_DESCRIPTION!C    n,namespace     /namespaces/
!_TAG_KIND_DESCRIPTION!C    p,prototype     /function prototypes/
!_TAG_KIND_DESCRIPTION!C    s,struct        /structure names/
!_TAG_KIND_DESCRIPTION!C    t,typedef       /typedefs/
!_TAG_KIND_DESCRIPTION!C    u,union /union names/
!_TAG_KIND_DESCRIPTION!C    v,variable      /variable definitions/
!_TAG_KIND_DESCRIPTION!C    x,externvar     /external and forward variable␣
↪declarations/
foo foo.c   /^foo (int i, int j)$/;"        f
main        foo.c   /^main (void)$/;"       f
```

1.7.1 TAG_KIND_DESCRIPTION

This is a newly introduced pseudo tag. It is not emitted by default. It is emitted only when `--pseudo-tags=+TAG_KIND_DESCRIPTION` is given.

This is for describing kinds; their letter, name, and description are enumerated in the tag.

ctags emits `TAG_KIND_DESCRIPTION` with following format:

```
!_TAG_KIND_SEPARATOR!{parser}   {letter},{name} /{description}/
```

A backslash and a slash in {description} is escaped with a backslash.

1.7.2 TAG_KIND_SEPARATOR

This is a newly introduced pseudo tag. It is not emitted by default. It is emitted only when `--pseudo-tags=+TAG_KIND_SEPARATOR` is given.

This is for describing separators placed between two kinds in a language.

Tag entries including the separators are emitted when `--extras=+q` is given; fully qualified tags contain the separators. The separators are used in scope information, too.

ctags emits `TAG_KIND_SEPARATOR` with following format:

```
!_TAG_KIND_SEPARATOR!{parser}   {sep}   /{upper}{lower}/
```

or

```
!_TAG_KIND_SEPARATOR!{parser}   {sep}   /{lower}/
```

Here {parser} is the name of language. e.g. PHP. {lower} is the letter representing the kind of the lower item. {upper} is the letter representing the kind of the upper item. {sep} is the separator placed between the upper item and the lower item.

The format without {upper} is for representing a root separator. The root separator is used as prefix for an item which has no upper scope.

* given as {upper} is a fallback wild card; if it is given, the {sep} is used in combination with any upper item and the item specified with {lower}.

Each backslash character used in {sep} is escaped with an extra backslash character.

Example output:

```
$ ./ctags -o - --extras=+p --pseudo-tags=  --pseudo-tags=+TAG_KIND_SEPARATOR input.
↪php
!_TAG_KIND_SEPARATOR!PHP    ::      /*c/
...
!_TAG_KIND_SEPARATOR!PHP    \\      /c/
...
!_TAG_KIND_SEPARATOR!PHP    \\      /nc/
...
```

The first line means *::* is used when combining something with an item of the class kind.

The second line means \ is used when a class item is at the top level; no upper item is specified.

The third line means \ is used when for combining a namespace item (upper) and a class item (lower).

Of course, ctags uses the more specific line when choosing a separator; the third line has higher priority than the first.

1.7.3 TAG_OUTPUT_MODE

This pseudo tag represents output mode: u-ctags or e-ctags.

See also *Compatible output and weakness*.

1.8 Parser own fields

A tag has a *name*, an *input* file name, and a *pattern* as basic information. Some fields like *language:*, *signature:*, etc are attached to the tag as optional information.

In Exuberant-ctags, fields are common to all languages. Universal-ctags extends the concept of fields; a parser can define its own field. This extension was proposed by @pragmaware in #857.

For implementing the parser own fields, the options for listing and enabling/disabling fields are also extended.

In the output of `--list-fields`, the owner of the field is printed in the *LANGUAGE* column:

```
$ ./ctags --list-fields
#LETTER NAME            ENABLED LANGUAGE         XFMT  DESCRIPTION
...
-       end             off     C                TRUE   end lines of various␣
↪constructs
-       properties      off     C                TRUE   properties (static, inline,
↪ mutable,...)
-       end             off     C++              TRUE   end lines of various␣
↪constructs
-       template        off     C++              TRUE   template parameters
-       captures        off     C++              TRUE   lambda capture list
-       properties      off     C++              TRUE   properties (static,␣
↪virtual, inline, mutable,...)
-       sectionMarker   off     reStructuredText TRUE   character used for␣
↪declaring section
-       version         off     Maven2           TRUE   version of artifact
```

e.g. reStructuredText is the owner of the sectionMarker field and both C and C++ own the end field.

`--list-fields` takes one optional argument, *LANGUAGE*. If it is given, `--list-fields` prints only the fields for that parser:

```
$ ./ctags --list-fields=Maven2
#LETTER NAME            ENABLED LANGUAGE        XFMT  DESCRIPTION
-       version         off     Maven2          TRUE  version of artifact
```

A parser own field only has a long name, no letter. For enabling/disabling such fields, the name must be passed to `--fields-<LANG>`.

e.g. for enabling the *sectionMarker* field owned by the *reStructuredText* parser, use the following command line:

```
$ ./ctags --fields-reStructuredText=+{sectionMarker} ...
```

The wild card notation can be used for enabling/disabling parser own fields, too. The following example enables all fields owned by the *C++* parser.

```
$ ./ctags --fields-C++='*' ...
```

* can also be used for specifying languages.

The next example is for enabling *end* fields for all languages which have such a field.

```
$ ./ctags --fields-'*'=+'{end}' ...
...
```

In this case, using wild card notation to specify the language, not only fields owned by parsers but also common fields having the name specified (*end* in this example) are enabled/disabled.

Using the wild card notation to specify the language is helpful to avoid incompatibilities between versions of Universal-ctags itself (SELF INCOMPATIBLY).

In Universal-ctags development, a parser developer may add a new parser own field for a certain language. Sometimes other developers then recognize it is meaningful not only for the original language but also other languages. In this case the field may be promoted to a common field. Such a promotion will break the command line compatibility for `--fields-<LANG>` usage. The wild card for <*LANG*> will help in avoiding this unwanted effect of the promotion.

With respect to the tags file format, nothing is changed when introducing parser own fields; <*fieldname*>:<*value*> is used as before and the name of field owner is never prefixed. The *language:* field of the tag identifies the owner.

1.9 Parser own extras

As man page of Exuberant-ctags says, `--extras` option specifies whether to include extra tag entries for certain kinds of information. This option is available in Universal-ctags, too.

In Universal-ctags it is extended; a parser can define its own extra flags. They can be controlled with `--extras-<LANG>=[+|-]{...}`.

See some examples:

```
$ ./ctags --list-extras
#LETTER NAME                ENABLED LANGUAGE        DESCRIPTION
F       fileScope           TRUE    NONE            Include tags ...
f       inputFile           FALSE   NONE            Include an entry ...
p       pseudo              FALSE   NONE            Include pseudo tags
q       qualified           FALSE   NONE            Include an extra ...
r       reference           FALSE   NONE            Include reference tags
g       guest               FALSE   NONE            Include tags ...
-       whitespaceSwapped   TRUE    Robot           Include tags swapping ...
```

See the *LANGUAGE* column. NONE means the extra flags are language independent (common). They can be enabled or disabled with *–extras=* as before.

Look at *whitespaceSwapped*. Its language is *Robot*. This flag is enabled by default but can be disabled with *–extras-Robot=-{whitespaceSwapped}*.

```
$ cat input.robot
*** Keywords ***
it's ok to be correct
    Python_keyword_2

$ ./ctags -o - input.robot
it's ok to be correct       input.robot     /^it's ok to be correct$/;"     k
it's_ok_to_be_correct       input.robot     /^it's ok to be correct$/;"     k

$ ./ctags -o - --extras-Robot=-'{whitespaceSwapped}' input.robot
it's ok to be correct       input.robot     /^it's ok to be correct$/;"     k
```

When disabled the name *it's_ok_to_be_correct* is not included in the tags output. In other words, the name *it's_ok_to_be_correct* is derived from the name *it's ok to be correct* when the extra flag is enabled.

1.9.1 Discussion

(This subsection should move to somewhere for developers.)

The question is what are extra tag entries. As far as I know none has answered explicitly. I have two ideas in Universal-ctags. I write "ideas", not "definitions" here because existing parsers don't follow the ideas. They are kept as is in variety reasons but the ideas may be good guide for people who wants to write a new parser or extend an exiting parser.

The first idea is that a tag entry whose name is appeared in the input file as is, the entry is NOT an extra. (If you want to control the inclusion of such entries, the classical `--kind-<LANG>=[+|-]...` is what you want.)

Qualified tags, whose inclusion is controlled by `--extras=+q`, is explained well with this idea. Let's see an example:

```
$ cat input.py
class Foo:
    def func (self):
        pass

$ ./ctags -o - --extras=+q --fields=+E input.py
Foo input.py        /^class Foo:$/;"        c
Foo.func    input.py        /^    def func (self):$/;"      m       class:Foo
↪ extra:qualified
func        input.py        /^    def func (self):$/;"      m       class:Foo
```

Foo and *func* are in *input.py*. So they are no extra tags. In other hand, *Foo.func* is not in *input.py* as is. The name is generated by ctags as a qualified extra tag entry. *whitespaceSwapped* extra flag of *Robot* parser is also aligned well on the idea.

I don't say all parsers follows this idea.

```
$ cat input.cc
class A
{
  A operator+ (int);
};

$ ./ctags --kinds-all='*' --fields= -o - input.cc
A   input.cc        /^class A$/
operator +  input.cc        /^  A operator+ (int);$/
```

In this example *operator+* is in *input.cc*. In other hand, *operator +* is in the ctags output as non extra tag entry. See a whitespace between the keyword *operator* and + operator. This is an exception of the first idea.

The second idea is that if the *inclusion* of a tag cannot be controlled well with `--kind-<LANG>=[+|-]...`, the tag may be an extra.

```
$ cat input.c
static int foo (void)
{
        return 0;
}
int bar (void)
{
        return 1;
}

$ ./ctags --sort=no -o - --extras=+F input.c
foo input.c /^static int foo (void)$/;"     f       typeref:typename:int    file:
bar input.c /^int bar (void)$/;"    f       typeref:typename:int

$ ./ctags -o - --extras=-F input.c
foo input.c /^static int foo (void)$/;"     f       typeref:typename:int    file:

$
```

Function *foo* of C language is included only when *F* extra flag is enabled. Both *foo* and *bar* are functions. Their inclusions can be controlled with *f* kind of C language: `--kind-C=[+|-]f`.

The difference between static modifier or implicit extern modifier in a function definition is handled by *F* extra flag.

Basically the concept kind is for handling the kinds of language objects: functions, variables, macros, types, etc. The concept extra can handle the other aspects like scope (static or extern).

However, a parser developer can take another approach instead of introducing parser own extra; one can prepare *staticFunction* and *exportedFunction* as kinds of one's parser. The second idea is a just guide; the parser developer must decide suitable approach for the target language.

Anyway, in the second idea, `--extra` is for controlling inclusion of tags. If what you want is not about inclusion, `--param-<LANG>` can be used as the last resort.

1.10 Parser own parameter

To control the detail of a parser, `--param-<LANG>` option is introduced. `--kinds-<LANG>`, `--fields-<LANG>`, `--extras-<LANG>` can be used for customizing the behavior of a parser specified with `<LANG>`.

`--param-<LANG>` should be used for aspects of the parser that the options(kinds, fields, extras) cannot handle well.

A parser defines a set of parameters. Each parameter has name and takes an argument. A user can set a parameter with following notation

```
--param-<LANG>:name=arg
```

An example of specifying a parameter

```
--param-CPreProcessor:if0=true
```

Here *if0* is a name of parameter of CPreProcessor parser and *true* is the value of it.

All available parameters can be listed with `--list-params` option.

```
$ ./ctags --list-params
#PARSER         NAME     DESCRIPTION
CPreProcessor   if0      examine code within "#if 0" branch (true or [false])
CPreProcessor   ignore   a token to be specially handled
```

(At this time only CPreProcessor parser has parameters.)

1.11 Customizing xref output

`--_xformat` option allows a user to customize the cross reference (xref) output enabled with `-x`.

```
--_xformat=FORMAT
```

The notation for FORMAT is similar to that employed by *printf(3)* in the C language; *%* represents a slot which is substituted with a field value when printing. You can specify multiple slots in FORMAT. Here field means an item listed with `--list-fields` option.

The notation of a slot:

```
%[-][.][WIDTH-AND-ADJUSTMENT]FIELD-SPECIFIER
```

`FIELD-SPECIFIER` specifies a field whose value is printed. Short notation and long notation are available. They can be mixed in a FORMAT. Specifying a field with either notation, one or more fields are activated internally.

The short notation is just a letter listed in the LETTER column of the `--list-fields` output.

The long notation is a name string surrounded by braces(*{* and *}*). The name string is listed in the NAME column of the output of the same option. To specify a field owned by a parser, prepend the parser name to the name string with . as a separator.

Wild card (*) can be used where a parser name is specified. In this case both common and parser own fields are activated and printed. If a common field and a parser own field have the same name, the common field has higher priority.

WIDTH-AND-ADJUSTMENT is a positive number. The value of the number is used as the width of the column where a field is printed. The printing is right adjusted by default, and left adjusted when - is given as prefix. The output is not truncated by default even if its field width is specified and smaller than width of output value. For truncating the output to the specified width, use . as prefix.

An example of specifying common fields:

```
$  ./ctags -x --_xformat="%-20N %4n %-16{input}|" main/main.c | head
CLOCKS_PER_SEC         360 main/main.c     |
CLOCKS_PER_SEC         364 main/main.c     |
CLOCK_AVAILABLE        358 main/main.c     |
CLOCK_AVAILABLE        363 main/main.c     |
Totals                  87 main/main.c     |
__anonae81ef0f0108      87 main/main.c     |
addTotals              100 main/main.c     |
batchMakeTags          436 main/main.c     |
bytes                   87 main/main.c     |
clock                  365 main/main.c     |
```

Here *%-20N %4n %-16{input}|* is a format string. Let's look at the elements of the format.

%-20N

> The short notation is used here. The element means filling the slot with the name of the tag. The width of the column is 20 characters and left adjusted.

%4n

> The short notation is used here. The element means filling the slot with the line number of the tag. The width of the column is 4 characters and right adjusted.

%-16{input}

> The long notation is used here. The element means filling the slot with the input file name where the tag is defined. The width of column is 16 characters and left adjusted.

|

Printed as is.

Another example of specifying parser own fields:

```
$  ./ctags -x --_xformat="%-20N [%10{C.properties}]" main/main.c
CLOCKS_PER_SEC       [          ]
CLOCK_AVAILABLE      [          ]
Totals               [          ]
__anonae81ef0f0108   [          ]
addTotals            [    extern]
batchMakeTags        [    static]
bytes                [          ]
clock                [          ]
clock                [    static]
...
```

Here *"%-20N [%10{C.properties}]"* is a format string. Let's look at the elements of the format.

%-20N

Already explained in the first example.

[and *]*

Printed as is.

%10{C.properties}

The long notation is used here. The element means filling the slot with the value of the properties field of the C parser. The width of the column is 10 characters and right adjusted.

1.12 Incompatible changes in command line

1.12.1 `-D` option

For a ctags binary that had debugging output enabled in the build config stage, `-D` was used for specifying the level of debugging output. It is changed to `-d`. This change is not critical because `-D` option was not described in ctags.1 man page.

Instead `-D` is used for defining a macro in CPreProcessor parser.

1.13 Skipping utf-8 BOM

The three bytes sequence('xEFxBBxBF') at the head of an input file is skipped when parsing.

TODO:

- Do the same in guessing and selecting parser stage.
- Refect the BOM detection to encoding option

1.14 Readtags

1.14.1 Printing line numbers with `-n`

If both `-e` and `-n` are given, readtags prints the *line:* field.

1.14.2 Filtering in readtags command

readtags has ability to find tag entries by name.

The concept of filtering is inspired by the display filter of Wireshark. You can specify more complex conditions for searching. Currently this feature is available only on platforms where *fmemopen* is available as part of libc. Filtering in readtags is an experimental feature.

The syntax of filtering rules is based on the Scheme language, a variant of Lisp. The language has prefix notation and parentheses.

Before printing an entry from the tags file, readtags evaluates an expression (S expression or sexp) given as an option argument to `-Q`. As the result of the evaluation, readtags gets a value. false represented as *#f*, indicates rejection: readtags doesn't print it.

```
SEXP =
     LIST
     INTEGER
     BOOLEAN
     STRING
     SYMBOL

     LIST = ( SEXP... ) | ()
     INTEGER = [0-9]+
     BOOLEAN = #t | #f
     STRING  = "..."
     SYMBOL  = null?
                 and
                  or
                 not
                 eq?
                   <
                   >
                  <=
                  >=
               prefix?
               suffix?
               substr?
                member
                     $
              $name
             $input
            $access
              $file
          $language
     $implementation
               $line
               $kind
               $role
            $pattern
           $inherits
         $scope-kind
         $scope-name
                $end
```

All symbols starting with *$* represent a field of a tag entry which is being tested against the S expression. Most will evaluate as a string or *#f*. It evaluates to *#f* when the field doesn't exist. *$inherits* is evaluated to a list of strings if the entry has an *inherits* field. The *scope* field holds structured data: the kind and name of the upper scope combined with *:*. The kind part is mapped to *$scope-kind*, and the name part to *$scope-name*.

$scope-kind and *$scope-name* can only be used if the input tags file is generated by ctags with `--fields=+Z`.

All symbols not prefixed with *$* are operators. When using these, put them at the head(car) of list. The rest(cdr) of the list is passed to the operator as arguments. Many of them are also available in the Scheme language; see the

other documents.

prefix?, suffix?, and substr? may only be available in this implementation. All of them take two strings. The first one is called the target.

The exception in the above naming convention is the *$* operator. *$* is a generic accessor for accessing extension fields. *$* takes one argument: the name of an extension field. It returns the value of the field as a string if a value is given, or *#f*.

```
(prefix? "TARGET" "TA")
=> #t

(prefix? "TARGET" "RGET")
=> #f

(prefix? "TARGET" "RGE")
=> #f

(suffix? "TARGET" "TA")
=> #f

(suffix? "TARGET" "RGET")
=> #t

(suffix? "TARGET" "RGE")
=> #f

(substr? "TARGET" "TA")
=> #t

(suffix? "TARGET" "RGET")
=> #t

(suffix? "TARGET" "RGE")
=> #t

(and (suffix? "TARGET" "TARGET")
     (prefix? "TARGET" "TARGET")
     (substr? "TARGET" "TARGET")
=> #t
```

Let's see examples.

1.14.3 Examples of input

Create the tags file (*foo.tags*) with following command line

```
$ ./ctags --fields='*' --extras='*' -o foo.tags foo.py
```

for following input (*foo.py*)

```
class Foo:
    def aq ():
        pass
    def aw ():
        pass
    def ae ():
        pass
    class A:
        pass
class Bar (Foo):
    def bq ():
```

(continues on next page)

(continued from previous page)

```
        pass
    def bw ():
        pass
    class B:
        pass

class Baz (Foo):
    def bq ():
        pass
    def bw ():
        pass
    class C:
        pass
```

1.14.4 Examples of filter expressions

- Print entries ending with "q"

```
$ ./readtags -e -t foo.tags -Q '(suffix? $name "q")' -l
Bar.bq  foo.py  /^    def bq ():$/;"    kind:member     language:Python␣
↪scope:class:Bar access:public   signature:()
Baz.bq  foo.py  /^    def bq ():$/;"    kind:member     language:Python␣
↪scope:class:Baz access:public   signature:()
Foo.aq  foo.py  /^    def aq ():$/;"    kind:member     language:Python␣
↪scope:class:Foo access:public   signature:()
aq      foo.py  /^    def aq ():$/;"    kind:member     language:Python␣
↪scope:class:Foo access:public   signature:()
bq      foo.py  /^    def bq ():$/;"    kind:member     language:Python␣
↪scope:class:Bar access:public   signature:()
bq      foo.py  /^    def bq ():$/;"    kind:member     language:Python␣
↪scope:class:Baz access:public   signature:()
```

- Print members of Baz

```
$ ./readtags -e -t foo.tags -Q '(and (eq? $kind "member") (eq? "Baz" $scope-
↪name))' -l
Baz.bq  foo.py  /^    def bq ():$/;"    kind:member     language:Python␣
↪scope:class:Baz access:public   signature:()
Baz.bw  foo.py  /^    def bw ():$/;"    kind:member     language:Python␣
↪scope:class:Baz access:public   signature:()
bq      foo.py  /^    def bq ():$/;"    kind:member     language:Python␣
↪scope:class:Baz access:public   signature:()
bw      foo.py  /^    def bw ():$/;"    kind:member     language:Python␣
↪scope:class:Baz access:public   signature:()
```

- Print only fully qualified entries (assuming "." is used as the separator)

```
$ ./readtags -e -t foo.tags -Q '(and (eq? $kind "member") (substr? $name "."))
↪' -l
Bar.bq  foo.py  /^    def bq ():$/;"    kind:member     language:Python␣
↪scope:class:Bar access:public   signature:()
Bar.bw  foo.py  /^    def bw ():$/;"    kind:member     language:Python␣
↪scope:class:Bar access:public   signature:()
Baz.bq  foo.py  /^    def bq ():$/;"    kind:member     language:Python␣
↪scope:class:Baz access:public   signature:()
Baz.bw  foo.py  /^    def bw ():$/;"    kind:member     language:Python␣
↪scope:class:Baz access:public   signature:()
Foo.ae  foo.py  /^    def ae ():$/;"    kind:member     language:Python␣
↪scope:class:Foo access:public   signature:()
```

(continues on next page)

(continued from previous page)

```
Foo.aq  foo.py  /^    def aq ():$/;"    kind:member     language:Python␣
↪scope:class:Foo access:public   signature:()
Foo.aw  foo.py  /^    def aw ():$/;"    kind:member     language:Python␣
↪scope:class:Foo access:public   signature:()
```

- Print only classes inheriting Foo

```
$ ./readtags  -e -t foo.tags -Q '(and (member "Foo" $inherits) (eq? $kind
↪"class"))' -l
Bar     foo.py  /^class Bar (Foo):$/;"  kind:class      language:Python␣
↪inherits:Foo    access:public
Baz     foo.py  /^class Baz (Foo): $/;" kind:class      language:Python␣
↪inherits:Foo    access:public
```

CHAPTER 2

Request for extending a parser (or Reporting a bug of parser)

Maintainer Masatake YAMATO <yamato@redhat.com>

Table of contents

Sometimes you will find u-ctags doesn't make a tag for a language object unexpectedly. Writing a patch for making the tag is appreciate. However, you may not have time to do so. In that case, you can open an issue on the GitHub page of u-ctags.

This section tells you how to drive u-ctags developers effectively.

2.1 Before reporting

U-Ctags just captures the definitions of language objects. U-ctags has an infrastructure for capturing references for language objects. However, we implement reference tagging limited area. We will not work on writing new code for capturing references for your favorite language. About requests for capturing reference tags, we will say "patches are welcome.".

What kind of language objects u-ctags captures is controlled by *–kind-<LANG>* option. Some kinds are disabled by default because we assume people don't want too large *tags* file. When you cannot find a language object you want in a tags file, it is worth for checking the status of kinds. *–list-kinds=<LANG>* or (*–list-kinds-full=<LANG>*) option lists the status of the given language.

Let's see an example.

Consider following input (foo.h):

```
struct point {
  int x, y;
};
```

(continues on next page)

(continued from previous page)

```
struct point *make_point(int x0, int y0);
```

tags output generated with *u-ctags -o - /tmp/foo.h* is as following.

```
point    foo.h    /^struct point {$/;"    s
x    foo.h    /^  int x, y;$/;"    m    struct:point    typeref:typename:int
y    foo.h    /^  int x, y;$/;"    m    struct:point    typeref:typename:int
```

Though *point*, *x* and *y* are tagged, the declaration *make_point* is not tagged because *prototype* kind of C++ is disabled by default. You can know it from the output of *ctags –list-kinds-full=C++*.

```
#LETTER NAME        ENABLED REFONLY NROLES MASTER DESCRIPTION
A       alias       no      no      0      NONE   namespace aliases
L       label       no      no      0      C      goto labels
N       name        no      no      0      NONE   names imported via using␣
↪scope::symbol
...
p       prototype   no      no      0      C      function prototypes
```

By turning on the kind with *–kinds-C++=+p*, u-ctags tags *make_point*:

```
make_point    foo.h    /^struct point *make_point(int x0, int y0);$/;"    p    ␣
↪typeref:struct:point *
point    foo.h    /^struct point {$/;"    s
x    foo.h    /^  int x, y;$/;"    m    struct:point    typeref:typename:int
y    foo.h    /^  int x, y;$/;"    m    struct:point    typeref:typename:int
```

Wildcard * is for enabling all kinds of a language at once. *–kinds-C++=** option enables all kinds of C++ parser. If you specify *all* as the name of <*LANG*>, you can enable all kinds of all languages at once.

2.2 The content of report

Don't assume following three things.

U-ctags developers know vi.

> If you explain the expectation about how tags related functions of vi and its plugins, U-ctags developers don't understand it. The answer from them can be "it can be a bug of vi."

U-ctags developers know the programming language that you are talking.

> U-ctags developers need your help to understand the meaning of language object you asked tagging especially about kind. A person extending a parser have to decide a kind of newly tagging language object: reusing an existing kind or introducing a new kind. U-ctags developers expect a report know the concent kind, field, and extra. ctags.1 man page of u-ctags explains them.

English is the native language of the head maintainer.

> I don't want to write this but I have to write this. Following are my private request for reporters.
>
> Instead of long explanation, showing code or output examples make me understand what you want.
>
> Don't omit sentences. Please, write your sentence in full.
>
> Use pronounce fewer.

U-ctags can generate something meaningful from a broken input.

> From garbage, u-ctags generates garbage. For a syntactically broken input source file, U-ctags does not work well. U-ctags developers will not work on improving u-ctags for handing such input. The exception is that macro related input. Well known one is C and C++.

Following a tuple with three items helps us to understand what you want.

1. Input file

   A shorter input file is better. However, it must be syntactically valid. Show the URL (or something) where you get the input file. It is needed to incorporate the input file to the u-ctags source tree as a test case.

2. Command line running u-ctags
3. Expected output

These three items should be rendered preformatted form on an issue page of GitHub. Use triple backquotes notation of GitHub's markdown notation. I will close an issue with a bad notation like this issue.

2.3 An example of good report

For the following input file(input.f90), u-ctags reports incomplete pattern for fuction *f* at the line 23.

```
! input.f90, taken from https://github.com/universal-ctags/ctags/issues/1616
module example_mod

 ! This module contains two interfaces:
 !   1. f_interface, which is an interface to the local f function
 !   2. g, which is implemented in the example_smod submodule

    interface f_interface
       ! The function `f` is defined below, within the `contains` statement
        module function f(x) result(y)
           integer :: x, y
        end function f
     end interface f_interface

    interface
       ! The function `g` is implemented in example_smod.f90
        module function g(x) result(y)
           integer :: x,y
        end function g
    end interface

    contains
     function f(x) result(y)
        integer :: x, y

        y = x * 2
     end function f
end module example_mod
```

I run ctags with following command line:

```
u-ctags --fields=+n -o - /tmp/input.f90
```

What I got:

```
example_mod     /tmp/input.f90  /^module example_mod$/;"        m       line:2
f       /tmp/input.f90  /^     fu/;"    f       line:23
f_interface     /tmp/input.f90  /^    interface f_interface$/;" i       line:8 ␣
↪module:example_mod
```

I think this should be:

```
example_mod     /tmp/input.f90  /^module example_mod$/;"        m       line:2
f       /tmp/input.f90  /^     function f/;"    f       line:23
f_interface     /tmp/input.f90  /^    interface f_interface$/;" i       line:8 ␣
↪module:example_mod
```

or:

```
example_mod     /tmp/input.f90  /^module example_mod$/;"        m       line:2
f       /tmp/input.f90  /^     function f(x) result(y)/;"       f       line:23
f_interface     /tmp/input.f90  /^    interface f_interface$/;" i       line:8 ␣
↪module:example_mod
```

Either way, */^ fu/* is too short as a pattern.

The version of u-ctags is *83b0d1f6*:

```
$ u-ctags --version
Universal Ctags 0.0.0(83b0d1f6), Copyright (C) 2015 Universal Ctags Team
Universal Ctags is derived from Exuberant Ctags.
Exuberant Ctags 5.8, Copyright (C) 1996-2009 Darren Hiebert
  Compiled: Dec 15 2017, 08:07:36
  URL: https://ctags.io/
  Optional compiled features: +wildcards, +regex, +multibyte, +debug, +option-
↪directory, +xpath, +json, +interactive, +sandbox, +yaml, +aspell
```

CHAPTER 3

Contributions

Maintainer Masatake YAMATO <yamato@redhat.com>

Table of contents

- *Build script*

You are welcome.

This is what we would like potential contributors to know. In this section "you" means a contributor, and "we" means reviewers. "I" means Masatake YAMATO, the author of this section.

3.1 General topics

3.1.1 Origin of changes and license

Make clear where the patches come from and who wrote them.

If you backport patches from Geany or some other project, their commit IDs should be logged, too.

Include a copyright notice when adding a new `{parsers,main}/*.[ch]` file.

A new file also requires a license notice at the head of the file.

We expect your change (or new code) to be provided under the terms of the General Public License version 2 or any later version. We would like you to express "version 2 or any later version".

3.1.2 Commit log

(For new parsers the following criteria is not applicable.)

Make clear the original motivation for the change and/or the impact on the tags file.

If you fix a bug reported somewhere on the web, its URL should be logged, too.

If the bug is reported in the Exuberant-ctags tracker on the SourceForge web site, log it as sf-bugs:N, sf-patches:N, sf-support-requests:N, or sf-feature-requests:N. docs/tracking.rst also should be updated.

3.1.3 NEWS file

Update docs/news.rst especially if you add a new parser.

3.1.4 Testing

Add test cases, and run both existing cases and your new cases.

If you add a new parser or modify an existing parser, add new test cases to "Units". If you modify the core, add new test cases to "Tmain". The way to write and run test cases is described in the "Testing ctags" section of this guide.

With the exception of the tmain test harness, you can specify VG=1 for running test cases under the Valgrind memory debugger.

A parse should not enter an infinite loop for bad input. A parse should not crash for bad input. A parse should return control to its caller for bad input.

Describe what kind of tests are passed in the commit message. e.g.

```
make units LANGUAGES=TTCN VG=1 is passed.
make fuzz LANGUAGES=TTCN VG=1  is passed.
make chop LANGUAGES=TTCN VG=1  is passed.
```

3.1.5 C language

Don't forget to use *static* modifiers. Don't introduce unnecessary global variables.

Remove unused variables and types. If you want to keep them in your source code, include a descriptive comment.

Use the available facilities provided by the ctags core. If the facilities are not enough for writing a parser, consider extending the core first.

Use underscores in names only in file scope objects. Don't use them in function declarations, variable declarations or macro names in header files.

Basic whitespace settings are specified in the EditorConfig configuration file (*.editorconfig*). There are plugins available for most popular editors to automatically configure these settings.

Style guidelines are largely captured in the Uncrustify configuration file (*.uncrustify.cfg*). Formatting can be checked with:

```
$ uncrustify -c .uncrustify.cfg -f parsers/awk.c | diff -u parsers/awk.c -
```

Don't mix *whitespace cleanup* fixes and other improvements in one commit when changing the existing code. Style fixes, including *whitespace cleanup*, should be in a separate commit. Mixing functional changes with style fixes makes reviewing harder.

If possible, don't use file static variables. Find an alternative way that uses parameters.

Notes for GNU emacs users

If you use GNU emacs, utilize the *.editorconfig* configuration based on non-GNU C style. Here non-GNU C style means "align a keyword for control flow and *{* of the block start".

GNU style:

```
if (...)
    {
        ...
```

non-GNU style:

```
if (...)
{
        ...
```

For combining the style and *.editorconfig* configuration, put following code snippet to your .emacs:

```
(add-hook 'hack-local-variables-hook
        (lambda () (editorconfig-apply)))
```

.dir-locals.el in ctags source tree applies "linux" style of *cc-mode*. Above code snippet applies the *.editorconfig* configuration AFTER installing the "linux" style to the current buffer.

I like GNU style, but for keeping consistency in existing code of Exuberant-ctags, the origin of Universal-ctags, I introduced the style and configuration to my .emacs. Please, do the same.

3.1.6 Command line options

Don't introduce *–<LANG>-foo=...* style options. They are less suitable for command-line completion by the zsh/bash completion engines. Instead, introduce *–foo-<LANG>=...* style options.

Add an entry to docs/news.rst if you change the behavior of an option or introduce a new option. If you think the option is stable enough, add it to ctags.1.in, too.

Use underscore as a prefix for experimental options. Once an option is introduced, it must be maintained. We don't want to remove it later. If you are not sure of the usefulness of the option, use an underscore at the start of a long option name like: *–_echo*.

Write a test case for Tmain or Units.

Don't remove an option, especially if it exists in Exuberant-ctags. We want to maintain compatibility as much as possible.

3.1.7 Test cases

Add a test case to Unit when creating or modifying a parser.

Add a test case to Tmain when modifying the core.

Add a test case to Tinst when modifying the install target in the Makefile.

3.1.8 Compatibility

We are trying to maintain compatibility with Exuberant-ctags in the following two areas.

3.1.9 Tag file compatibility with Exuberant-ctags

We will not accept a patch that breaks the tags file format described in "Proposal for extended Vi tags file format" a.k.a. FORMAT file.

TBW.

3.1.10 Command line option with Exuberant-ctags

TBW.

3.2 Specific to add new parser and/or new kind/role

When working on ctags I take into account the following uses for tags:

1. inserting the name with completion,
2. jumping to the definition of the name (in an editor or similar tool),
3. navigating the source code tree,
4. summarizing the source code tree, and
5. answering a query about the source code tree.

When I review new parser code, I expect the parser to contribute to these purposes.

3.2.1 What should be tagged?

There are two classes of tags. The primary class is a definition tag. If a name is defined in a file, the name and the line and the file where the name is defined should be tagged (recorded). However, in some languages answering, "What is a definition?" is not so obvious. You may have to decide what is tagged in your parser thoughtfully. The purposes listed at the top of this subsection should help you decide.

The secondary class is a reference tag. This is newly introduced in Universal-ctags and is not available in Exuberant-ctags. If a name is used (or referenced) in a file, it can be tagged as a reference tag.

Don't be confused by the two tag classes.

3.2.2 Defining kinds and roles

Defining kinds is the most important task in writing a new parser. Once a kind is introduced, we cannot change it because it breaks tags file compatibility.

If you are not interested in designing kinds because you are an emacs user and use just TAGS output, there are two choices: TBW.

3.2.3 Scope information and full qualified tags

Optional. TBW.

3.2.4 Adding a new field

TBW.

3.2.5 Reference

In the comment at the head of your source file, include a URL for a web page that explains the language your parser deals with. Especially if the language is not well known.

Here is an example.

```
/*
*
*   Copyright (c) 2016, Masatake YAMATO
*   Copyright (c) 2016, Red Hat, K.K.
*
*   This source code is released for free distribution under the terms of the
*   GNU General Public License version 2 or (at your option) any later version.
*
*   This module contains functions for generating tags for property list defined
*   in http://www.apple.com/DTDs/PropertyList-1.0.dtd.
*/
```

3.2.6 Testing your parser

If possible, prepare a simple test and a complex one. The simple one for helping us, the maintainers, understand the intent of the modification.

If there are more than 3 test cases for a parser, a parser specific test case directory should be prepared like *Units/parser-c.r*.

3.2.7 Writing parser in regex

You can write a parser with regex patterns.

optlib2c, a part of the Universal-ctags build system can translate a parser written in regex patterns into C source code.

The *man* parser is one example described in regex patterns. See the output of the following command line for details:

> *git show 0a9e78a8a40e8595b3899e2ad249c8f2c3819c8a^..89aa548*

Translated C code is also committed to our git repository. The translated code is useful for building ctags on the platforms where optlib2c doesn't run.

The regex approach is also suitable for prototyping.

3.2.8 Squashing commits

When you submit a pull request you might receive some comments from a reviewer and, in response, update your patches. After updating, we would like you to squash your patches into logical units of work before we merge them to keep the repository history as simple as possible.

Quoted from @steveno in #393:

> You can check out this page for a good example of how to squash commits http://gitready.com/advanced/2009/02/10/squashing-commits-with-rebase.html
>
> Once you've squashed all your commits, simply do a git push -f to your fork, and GitHub will update the pull request for you automatically.

3.2.9 Build script

Add your *.c* file to *source.mak*.

In addition, update *win32/ctags_vs2013.vcxproj* and *win32/ctags_vs2013.vcxproj.filters*. Otherwise our CI process run on Appveyor will fail.

CHAPTER 4

Parsers

This section deals with individual parser topics.

4.1 Asm parser

Maintainer Masatake YAMATO <yamato@redhat.com>

The original (Exuberant-ctags) parser handles #define C preprocessor directive and C style comments by itself. In Universal-ctags Asm parser utilizes CPreProcessor meta parser for handling them. So a language object defined with #define is tagged as "defines" of CPreProcessor language, not Asm language.

```
$ cat input.S
#define S 1

$ e-ctags --fields=+l  -o - input.S
S    input.S /^#define S 1$/;"       d       language:Asm

$ u-ctags --fields=+l  -o - input.S
S    input.S /^#define S /;" d       language:CPreProcessor  file:
```

4.2 CMake parser

The CMake parser is used for `.cmake` and `CMakeLists.txt` files. It generates tags for the following items:

- User-defined functions
- User-defined macros
- User-defined options created by `option()`
- Variables defined by `set()`
- Targets created by `add_custom_target()`, `add_executable()` and `add_library()`

The parser uses the experimental multi-table regex `ctags` options to perform the parsing and tag generation.

Caveats:

Names that are `${}` references to variables are not tagged.

For example, given the following:

```
set(PROJECT_NAME_STR ${PROJECT_NAME})
add_executable( ${PROJECT_NAME_STR} ... )
add_custom_target( ${PROJECT_NAME_STR}_tests ... )
add_library( sharedlib ... )
```

...the variable `PROJECT_NAME_STR` and target `sharedlib` will both be tagged, but the other targets will not be.

References:

https://cmake.org/cmake/help/latest/manual/cmake-language.7.html

4.3 The new C/C++ parser

Maintainer Szymon Tomasz Stefanek <s.stefanek@gmail.com>

4.3.1 Introduction

The C++ language has strongly evolved since the old C/C++ parser was written. The old parser was struggling with some of the new features of the language and has shown signs of reaching its limits. For this reason in February/March 2016 the C/C++ parser was rewritten from scratch.

In the first release several outstanding bugs were fixed and some new features were added. Among them:

- Tagging of "using namespace" declarations
- Tagging of function parameters
- Extraction of function parameter types
- Tagging of anonymous structures/unions/classes/enums
- Support for C++11 lambdas (as anonymous functions)
- Support for function-level scopes (for local variables and parameters)
- Extraction of local variables which include calls to constructors
- Extraction of local variables from within the for(), while(), if() and switch() parentheses.
- Support for function prototypes/declarations with trailing return type

At the time of writing (March 2016) more features are planned.

4.3.2 Notable New Features

Some of the notable new features are described below.

Properties

Several properties of functions and variables can be extracted and placed in a new field called `properties`. The syntax to enable it is:

```
$ ctags ... --fields-c++=+{properties} ...
```

At the time of writing the following properties are reported:

- `virtual`: a function is marked as virtual

- `static`: a function/variable is marked as static
- `inline`: a function implementation is marked as inline
- `explicit`: a function is marked as explicit
- `extern`: a function/variable is marked as extern
- `const`: a function is marked as const
- `pure`: a virtual function is pure (i.e = 0)
- `override`: a function is marked as override
- `default`: a function is marked as default
- `final`: a function is marked as final
- `delete`: a function is marked as delete
- `mutable`: a variable is marked as mutable
- `volatile`: a function is marked as volatile
- `specialization`: a function is a template specialization
- `scopespecialization`: template specialization of scope `a<x>::b()`
- `deprecated`: a function is marked as deprecated via `__attribute__`
- `scopedenum`: a scoped enumeration (C++11)

Preprocessor macros

The new parser supports the definition of real preprocessor macros via the `-D` option. All types of macros are supported, including the ones with parameters and variable arguments. Stringification, token pasting and recursive macro expansion are also supported.

Option `-I` is now simply a backward-compatible syntax to define a macro with no replacement.

The syntax is similar to the corresponding gcc `-D` option.

Some examples follow.

```
$ ctags ... -D IGNORE_THIS ...
```

With this commandline the following C/C++ input

```
int IGNORE_THIS a;
```

will be processed as if it was

```
int a;
```

Defining a macro with parameters uses the following syntax:

```
$ ctags ... -D "foreach(arg)=for(arg;;)" ...
```

This example defines `for(arg;;)` as the replacement `foreach(arg)`. So the following C/C++ input

```
foreach(char * p,pointers)
{

}
```

is processed in new C/C++ parser as:

```
for(char * p;;)
{

}
```

and the p local variable can be extracted.

The previous commandline includes quotes since the macros generally contain characters that are treated specially by the shells. You may need some escaping.

Token pasting is performed by the ## operator, just like in the normal C preprocessor.

```
$ ctags ... -D "DECLARE_FUNCTION(prefix)=int prefix ## Call();"
```

So the following code

```
DECLARE_FUNCTION(a)
DECLARE_FUNCTION(b)
```

will be processed as

```
int aCall();
int bCall();
```

Macros with variable arguments use the gcc __VA_ARGS__ syntax.

```
$ ctags ... -D "DECLARE_FUNCTION(name,...)=int name(__VA_ARGS__);"
```

So the following code

```
DECLARE_FUNCTION(x,int a,int b)
```

will be processed as

```
int x(int a,int b);
```

4.3.3 Incompatible Changes

The parser is mostly compatible with the old one. There are some minor incompatible changes which are described below.

Anonymous structure names

The old parser produced structure names in the form __anonN where N was a number starting at 1 in each file and increasing at each new structure. This caused collisions in symbol names when ctags was run on multiple files.

In the new parser the anonymous structure names depend on the file name being processed and on the type of the structure itself. Collisions are far less likely (though not impossible as hash functions are unavoidably imperfect).

Pitfall: the file name used for hashing includes the path as passed to the ctags executable. So the same file "seen" from different paths will produce different structure names. This is unavoidable and is up to the user to ensure that multiple ctags runs are started from a common directory root.

File scope

The file scope information is not 100% reliable. It never was. There are several cases in that compiler, linker or even source code tricks can "unhide" file scope symbols (for instance *.c files can be included into each other)

and several other cases in that the limitation of the scope of a symbol to a single file simply cannot be determined with a single pass or without looking at a program as a whole.

The new parser defines a simple policy for file scope association that tries to be as compatible as possible with the old parser and should reflect the most common usages. The policy is the following:

- Namespaces are in file scope if declared inside a .c or .cpp file
- Function prototypes are in file scope if declared inside a .c or .cpp file
- K&R style function definitions are in file scope if declared static inside a .c file.
- Function definitions appearing inside a namespace are in file scope only if declared static inside a .c or .cpp file. Note that this rule includes both global functions (global namespace) and class/struct/union members defined outside of the class/struct/union declaration.
- Function definitions appearing inside a class/struct/union declaration are in file scope only if declared static inside a .cpp file
- Function parameters are always in file scope
- Local variables are always in file scope
- Variables appearing inside a namespace are in file scope only if they are declared static inside a .c or .cpp file
- Variables that are members of a class/struct/union are in file scope only if declared in a .c or .cpp file
- Typedefs are in file scope if appearing inside a .c or .cpp file

Most of these rules are debatable in one way or the other. Just keep in mind that this is not 100% reliable.

Inheritance information

The new parser does not strip template names from base classes. For a declaration like

```
template<typename A> class B : public C<A>
```

the old parser reported `C` as base class while the new one reports `C<A>`.

Typeref

The syntax of the typeref field (`typeref:A:B`) was designed with only struct/class/union/enum types in mind. Generic types don't have `A` information and the keywords became entirely optional in C++: you just can't tell. Furthermore, struct/class/union/enum types share the same namespace and their names can't collide, so the `A` information is redundant for most purposes.

To accommodate generic types and preserve some degree of backward compatibility the new parser uses struct/class/union/enum in place of `A` where such keyword can be inferred. Where the information is not available it uses the 'typename' keyword.

Generally, you should ignore the information in field `A` and use only information in field `B`.

4.4 The new HTML parser

Maintainer Jiri Techet <techet@gmail.com>

4.4.1 Introduction

The old HTML parser was line-oriented based on regular expression matching. This brought several limitations like the inability of the parser to deal with tags spanning multiple lines and not respecting HTML comments. In addition, the speed of the parser depended on the number of regular expressions - the more tag types were extracted, the more regular expressions were needed and the slower the parser became. Finally, parsing of embedded JavaScript was very limited, based on regular expressions and detecting only function declarations.

The new parser is hand-written, using separated lexical analysis (dividing the input into tokens) and syntax analysis. The parser has been profiled and optimized for speed so it is one of the fastest parsers in universal-ctags. It handles HTML comments correctly and in addition to existing tags it extracts also <h1>, <h2> and <h3> headings. It should be reasonably simple to add new tag types.

Finally, the parser uses the new functionality of universal-ctags to use another parser for parsing other languages within a host language. This is used for parsing JavaScript within <script> tags and CSS within <style> tags. This simplifies the parser and generates much better results than having a simplified JavaScript or CSS parser within the HTML parser. To run JavaScript and CSS parsers from HTML parser, use *–extras=+g* option.

4.5 puppetManifest parser

Maintainer Masatake YAMATO <yamato@redhat.com>

puppetManifest is an experimental parser for testing multi tables regex meta parser defined with `--_mtable-<LANG>` option.

The parser has some bugs derived from the limit of the multi tables regex meta parser.

Here document

> The parser cannot ignore the contents inside the area of here document. The end marker of here document is defined in the source code. Currently, ctags has no way to add a regex pattern for detecting the end maker.

4.6 The new Python parser

Maintainer Colomban Wendling <ban@herbesfolles.org>

4.6.1 Introduction

The old Python parser was a line-oriented parser that grew way beyond its capabilities, and ended up riddled with hacks and easily fooled by perfectly valid input. By design, it especially had problems dealing with constructs spanning multiple lines, like triple-quoted strings or implicitly continued lines; but several less tricky constructs were also mishandled, and handling of lexical constructs was duplicated and each clone evolved in its own direction, supporting different features and having different bugs depending on the location.

All this made it very hard to fix some existing bugs, or add new features. To fix this regrettable state of things, the parser has been rewritten from scratch separating lexical analysis (generating tokens) from syntactical analysis (understanding what the lexemes mean). This moves understanding lexemes to a single location, making it consistent and easier to extend with new lexemes, and lightens the burden on the parsing code making it more concise, robust and clear.

This rewrite allowed to quite easily fix all known bugs of the old parser, and add many new features, including:

- Tagging function parameters
- Extraction of decorators
- Proper handling of semicolons
- Extracting multiple variables in a combined declaration

- More accurate support of mixed indentation
- Tagging local variables

The parser should be compatible with the old one.

4.7 The new Tcl parser

Maintainer Masatake YAMATO <yamato@redhat.com>

Tcl parser is rewritten as a token oriented parser to support namespace. It was line oriented parser. Some incompatibility between Exuberant-ctags is introduced in the rewriting.

The line oriented parser captures *class*, *public|protected|private method*. They are definitions in ITcl and TclOO. The new token oriented Tcl parser ignores them. Instead ITcl and TclOO subparser running on Tcl base parser capture them.

4.7.1 Known bugs

Full qualified tags

The separator used in full qualified tags should be *::* but *.* is used.

A ITcl or TclOO class *C* can be defined in a Tcl namespace *N*:

```
namespace eval N {
    oo::class create C {
    }
}
```

When `--extras=+q` is given, currently ctags reports:

```
N.C ...
```

This should be:

```
N::C ...
```

Much work is needed to fix this.

Nested procs

proc defined in a *proc* cannot be captured well. This is a regression.

4.8 The Vim parser

4.8.1 Incompatible change

Quoted from `:help script-variable` in the Vim documentation:

```
                        *script-variable* *s:var*
In a Vim script variables starting with "s:" can be used. They
cannot be accessed from outside of the scripts, thus are local to
the script.
```

Exuberant-ctags records the prefix *s:* as part of a script-local variable's name. However, it is omitted from function names. As requested in issue #852 on GitHub, Universal-ctags now also includes the prefix in script-local function names.

4.9 XSLT parser

Maintainer Masatake YAMATO <yamato@redhat.com>

This parser only supports XSLT 1.0. If a newer version (2.0 and 3.0) is specified in an input file, ctags just skips the input. With `--verbose`, ctags prints the detected version of the input file.

Scope information generated by the XSLT parser is a bit broken. Currently a period (.) is used as the separator in nested scopes. This is the default separator value in ctags.

When the XSLT parser captures a node *<xsl:template match="...">* the value of the *match* attribute is tagged with kind *matchedTemplate*. When a *matchedTemplate* name is stored as part of the scope information, client tools may be confused because . is used both as the scope separator and in the XPath match expression.

CHAPTER 5

Output formats

This section deals with individual output-format topics.

`--output-format=` can be used for choosing an output format.

5.1 JSON output

5.1.1 Format

JSON output goes to standard output by default. Each generated tag line is represented as an object.

```
$./ctags --output-format=json /tmp/foo.py
{"_type": "tag", "name": "Foo", "path": "/tmp/foo.py", "pattern": "/^class Foo:$/",
↪ "kind": "class"}
```

Object keys which do not start with _ are normal fields and map directly to the fields of the default tags file format.

Keys that have names starting with _ are a JSON format meta field. Currently only *_type* is used and it can have the values *tag* for a normal tag or *ptag* for a pseudo tag.

JSON output is still under development and it is expected the format will change in the future. To give applications a chance to handle these changes ctags uses a pseudo tag, *JSON_OUTPUT_VERSION*, for specifying the format version.

```
$ ./ctags --extras='p' --pseudo-tags=JSON_OUTPUT_VERSION  --output-format=json /
↪tmp/foo.py
{"_type": "ptag", "name": "JSON_OUTPUT_VERSION", "path": "0.0", "pattern": "in␣
↪development"}
{"_type": "tag", "name": "Foo", "path": "/tmp/foo.py", "pattern": "/^class Foo:$/",
↪ "kind": "class"}
...
```

The JSON output format is newly designed and does not need to support the historical quirks of the default tags file format.

Kind long names are always used instead of kind letters. Enabling the *k* and/or *K* fields enables the *z* {kind} field internally.

Scope information is always split into scope kinds and scope names. Enabling the *s* field enables the *Z* {kind} and *p* {scopeKind} fields internally. As for all kinds, long names are used for printing ; kind letters are never used.

If you need kind letters, open an issue at the GitHub site of Universal-ctags.

5.1.2 Field introspection

Values for the most of all fields are represented in JSON string type. However, some of them are represented in integer type and/or boolean type. What kind of JSON data types used in a field can be known with the output of `--list-fields` option:

> $ ./ctags –list-fields #LETTER NAME ENABLED LANGUAGE XFMT JSTYPE DESCRIPTION N name on NONE TRUE s– tag name (fixed field) .. f file on NONE TRUE –b File-restricted scoping i inherits off NONE TRUE s-b Inheritance information . . . n line off NONE TRUE -i- Line number of tag definition . . .

JSTYPE column tells the data type of fields.

s string

i integer

b boolean

For example, The value for "inherits" field is represented in the string or boolean type.

5.2 Xref output

- Printing *z'{kind} field in xref format doesn't include 'kind:* prefix.
- Printing *Z'{scope} field in xref format doesn't include 'scope:* prefix.

CHAPTER 6

`--_interactive` Mode

Universal ctags can be run with `--_interactive`, which enters a REPL that can be used programmatically to control ctags generation. In this mode, json commands are received over stdin, and corresponding responses are emitted over stdout.

This feature needs ctags to be built with json support and this requires libjansson to be installed at build-time. If it's supported it will be listed in the output of `--list-features`:

```
$ ./ctags --list-features | grep json
json
```

Communication with Universal ctags over stdio uses the json lines format, where each json object appears on a single line and is terminated with a newline.

When `ctags --_interactive` is invoked, it will emit a single json object to stdout announcing its name and version. This signals the start of the interactive loop, and the user can begin sending commands over stdin.

```
$ ctags --_interactive
{"_type": "program", "name": "Universal Ctags", "version": "0.0.0"}
```

The following commands are currently supported in interactive mode:

- *generate-tags*

6.1 generate-tags

The `generate-tags` command takes two arguments:

- `filename`: name of the file to generate tags for (required)
- `size`: size in bytes of the file, if the contents will be received over stdin (optional)

The simplest way to generate tags for a file is by passing its path on filesystem(`file request`). The response will include one json object per line representing each tag, followed by a single json object with the `completed` field emitted once the file has been fully processed.

```
$ echo '{"command":"generate-tags", "filename":"test.rb"}' | ctags --_interactive
{"_type": "program", "name": "Universal Ctags", "version": "0.0.0"}
```

(continues on next page)

(continued from previous page)

```
{"_type": "tag", "name": "foobar", "path": "test.rb", "pattern": "/^  def foobar$/
↪", "kind": "method", "scope": "Test", "scopeKind": "class"}
{"_type":"completed", "command": "generate-tags"}
```

The `generate-tags` command can also be used to generate tags for code which is not present on filesystem(`inline request`). For example, an IDE might want to generate ctags for an unsaved buffer while the user is editing code. When `size` is specified, the corresponding number of bytes are read over stdin after the json object and newline.

```
$ (
  echo '{"command":"generate-tags", "filename":"test.rb", "size": 17}'
  echo 'def foobaz() end'
) | ctags --_interactive
{"_type": "program", "name": "Universal Ctags", "version": "0.0.0"}
{"_type": "tag", "name": "foobaz", "path": "test.rb", "pattern": "/^def foobaz()␣
↪end$/", "kind": "method"}
{"_type": "completed", "command": "generate-tags"}
```

6.2 sandbox submode

`sandbox` submode can be used with `--_interactive=sandbox`. This submode will activate a sandbox, to this limits the damage that the can be achieved when exploiting a buffer overflow in Universal-ctags.

In the sandbox submode ctags can generate tags only for inline requests because ctags has to use open system call to handle file requests. The open system call is not allowed in the sandbox.

This feature uses seccomp-bpf, and is only supported on Linux. To use the submode libseccomp is needed at build-time. If ctags was built with seccomp support, `sandbox` is listed in the output of `--list-features` option.

```
$ ./ctags --list-features | grep sandbox
sandbox
```

```
$ (
  echo '{"command":"generate-tags", "filename":"test.rb", "size": 17}'
  echo 'def foobaz() end'
) | ctags --_interactive=sandbox
{"_type": "program", "name": "Universal Ctags", "version": "0.0.0"}
{"_type": "tag", "name": "foobaz", "path": "test.rb", "pattern": "/^def foobaz()␣
↪end$/", "kind": "method"}
{"_type": "completed", "command": "generate-tags"}
```

CHAPTER 7

Choosing a proper parser in ctags

See ctags(1) within the source tree.

CHAPTER 8

Running multiple parsers on an input file

Universal-ctags provides parser developers ways(guest/host and sub/base) to run multiple parsers for an input file.

This section shows concepts behind the running multiple parsers, real examples, and APIs.

8.1 Applying a parser to specified areas of input file (guest/host)

guest/host combination considers the case that an input file has areas written in languages different from the language for the input file.

host parser parses the input file and detects the areas. *host parser* schedules *guest parsers* parsing the areas. *guest parsers* parses the areas.

guest parsers are run only when *–extras=+g* is given. If *–fields=+E* is given, all tags generated by a guest parser is marked *guest* in their *extras:* fields.

8.1.1 Examples of guest/host combinations

{CSS,JavaScript}/HTML parser combination

For an html file, you may want to run HTML parser, of course. The html file may have CSS areas and JavaScript areas. In other hand Universal-ctags has both CSS and JavaScript parsers. Don't you think it is useful if you can apply these parsers to the areas?

In this case, HTML has responsible to detect the CSS and JavaScript areas and record the positions of the areas. The HTML parser schedules delayed invocations of CSS and JavaScript parsers on the area with promise API.

Here HTML parser is a host parser. CSS and JavaScript parsers are guest parsers.

See *The new HTML parser* and parsers/html.c.

C/Yacc parser combination

A yacc file has some areas written in C. Universal-ctags has both YACC and C parsers. You may want to run C parser for the areas from YACC parser.

Here YACC parser is a host parser. C parser is a guest parser. See *promise API* and parsers/yacc.c.

Pod/Perl parser combination

Pod (Plain Old Documentation) is a language for documentation. The language can be used not only in a stand alone file but also it can be used inside a Perl script.

Universal-ctags has both parsers for Perl and Pod. The Perl parser recognizes the area where Pod document is embedded in a Perl script and schedules applying pod parser as a guest parser on the area.

8.1.2 API for running a parser in an area

promise API can be used. A host parser using the interface has responsibility to detect areas from input stream and record them with name of guest parsers that will be applied to the areas.

8.2 Tagging definitions of higher(upper) level language (sub/base)

8.2.1 Background

Consider an application written in language X. The application has its domain own concepts. Developers of the application may try to express the concepts in the syntax of language X.

In language X level, the developer can define functions, variables, types, and so on. Further more, if the syntax of X allows, the developers want to define higher level(= application level) things for implementing the domain own concepts.

Let me show the part of source code of SPY-WARS, an imaginary game application. It is written in scheme language, a dialect of lisp. (Here gauche is considered as the implementation of scheme interpreter).

```
(define agent-tables (make-hash-table))
(define-class <agent> ()
  ((rights :init-keyword :rights)
   (responsibilities :init-keyword :responsibilities)))

(define-macro (define-agent name rights responsibilities)
  `(hash-table-put! agent-tables ',name
                    (make <agent>
                      :rights ',rights
                      :responsibilities ',responsibilities)))

(define-agent Bond (kill ...) ...)
(define-agent Bourne ...)

...
```

define, *define-class*, and *define-macro* are keywords of scheme for defining a variable, class and macro. Therefore scheme parser of ctags should make tags for *agent-tables* with variable kind, <*agent*> with class kind, and *define-agent* with macro kind. There is no discussion here.

> NOTE: To be exactly *define-class* and *define-macro* are not the part of scheme language. They are part of gauche. That means three parsers are stacked: scheme, gosh, and SPY-WARS.

The interesting things here are *Bond* and *Bourne*.

```
(define-agent Bond (kill ...) ...)
(define-agent Bourne ...)
```

In scheme parser level, the two expressions define nothing; the two expressions are just macro(*define-agent*) expansions.

However, in the application level, they define agents as the macro name shown. In this level Universal-ctags should capture *Bond* and *Bourne*. The question is which parser should capture them? scheme parser should not; define-agent is not part of scheme language. Newly defined SPY-WARS parser is the answer.

Though *define-agent* is just a macro in scheme parser level, it is keyword in SPY-WARS parser. SPY-WARS parser makes a tag for a token next to *define-agent*.

The above example illustrates levels of language in an input file. scheme is used as the base language. With the base language we can assume an imaginary higher level language named SPY-WARS is used to write the application. To parse the source code of the application written in two stacked language, ctags uses the two stacked parsers.

Making higher level language is very popular technique in the languages of lisp family (see On Lisp for more details). However, it is not special to lisp.

Following code is taken from linux kernel written in C:

```
DEFINE_EVENT(mac80211_msg_event, mac80211_info,
        TP_PROTO(struct va_format *vaf),
        TP_ARGS(vaf)
);
```

There is no concept EVENT in C language, however it make sense in the source tree of linux kernel. So we can consider linux parser, based on C parser, which tags *mac80211_msg_event* as *event* kind.

8.2.2 Terms

Base parser and subparser

In the context of the SPY-WARS example, scheme parser is called a *base parser*. The SPY-WARS is called a *subparser*. A base parser tags definitions found in lower level view. A subparser on the base parser tags definitions found in higher level view. This relationship can be nested. A subparser can be a base parser for another sub parsers.

At a glance the relationship between two parsers are similar to the relationship guest parser and host parser description in *Applying a parser to specified areas of input file*. However, they are different. Though a guest parser can run stand-alone, a subparser cannot; a subparser needs help from base parser to work.

Top down parser choice and bottom up parser choice

There are two ways to run a subparser: top down or bottom up parser choices.

Universal-ctags can chose a subparser *automatically*. Matching file name patterns and extensions are the typical ways for choosing. A user can choose a subparser with *–language-force=* option. Choosing a parser in these deterministic way is called *top down*. When a parser is chosen as a subparser in the top down way, the subparser must call its base parser. The base parser may call methods defined in the sub parser.

Universal-ctags uses *bottom up* choice when the top down way doesn't work; a given file name doesn't match any patterns and extensions of subparsers and the user doesn't specify *–language-force=* explicitly. In choosing a subparser bottom up way it is assumed that a base parser for the subparser can be chosen by top down way. During a base parser running, the base parser tries to detect use of higher level languages in the input file. As shown later in this section, the base parser utilizes methods defined in its subparsers for the detection. If the base parser detects the use of a higher level language, a subparser for the higher level language is chosen. Choosing a parser in this non-deterministic way(dynamic way) is called *bottom up*.

Here is an example. Universal-ctags has both m4 parser and Autoconf parser. The m4 parser is a base parser. The Autoconf parser is a subparser based on the m4 parser. If *configure.ac* is given as an input file, Autoconf parser

is chosen automatically because the Autoconf parser has *configure.ac* in its patterns list. Based on the pattern matching, Universal-ctags chooses the Autoconf parser automatically(top down choice).

If *input.m4* is given as an input file, the Autoconf parser is not chosen. Instead the m4 parser is chosen automatically because the m4 parser has *.m4* in its extension list. The m4 parser passes every token finding in the input file to the Autoconf parser. The Autoconf parser gets the chance to probe whether the Autoconf parser itself can handle the input or not; if a token name is started with *AC_*, the Autoconf parser reports "this is Autoconf input though its file extension is *m4*" to the m4 parser. As the result the Autoconf parser is chosen(bottom up choice).

Some subparsers can be chosen both top down and bottom up ways. Some subparser can be chosen only top down way or bottom up ways.

Exclusive subparser and coexisting subparser

TBW. This must be filled when I implement python-celery parser.

8.2.3 API for making a combination of base parser and subparsers

Outline

You have to work on both sides: a base parser and subparsers.

A base parser must define a data structure type(*baseMethodTable*) for its subparsers by extending *struct subparser* defined in *main/subparser.h*. A subparser defines a variable(*subparser var*) having type *baseMethodTable* by filling its fields and registers *subparser var* to the base parser using dependency API.

The base parser calls functions pointed by *baseMethodTable* of subparsers during parsing. A function for probing a higher level language may be included in *baseMethodTable*. What kind of fields should be included in *baseMethodTable* is up to the design of a base parser and the requirements of its subparsers. A method for probing is one of them.

Registering a *subparser var* to a base parser is enough for the bottom up choice. For handling the top down choice (e.g. specifying *–language-force=subparser* in a command line), more code is needed.

call *scheduleRunningBaseparser* function from a function(*parser* method) assigned to *parser* member in *parserDefinition* of the subparser, *scheduleRunningBaseparser'is declared in *main/subparser.h*. 'scheduleRunningBaseparser* takes an integer argument that specifies the dependency used for registering the *subparser var*.

By extending *struct subparser* you can define a type for your subparser. Then make a variable for the type and declare a dependency on the base parser.

Details

Fields of *subparser* type

Here the source code of Autoconf/m4 parsers is referred as an example.

main/types.h:

```
struct sSubparser;
typedef struct sSubparser subparser;
```

main/subparser.h:

```
typedef enum eSubparserRunDirection {
        SUBPARSER_BASE_RUNS_SUB = 1 << 0,
        SUBPARSER_SUB_RUNS_BASE = 1 << 1,
        SUBPARSER_BI_DIRECTION  = SUBPARSER_BASE_RUNS_SUB|SUBPARSER_SUB_RUNS_BASE,
} subparserRunDirection;
```

(continues on next page)

(continued from previous page)

```
struct sSubparser {
        ...

        /* public to the parser */
        subparserRunDirection direction;

        void (* inputStart) (subparser *s);
        void (* inputEnd) (subparser *s);
        void (* exclusiveSubparserChosenNotify) (subparser *s, void *data);
};
```

A subparser must fill the fields of *subparser*.

direction field specifies how the subparser is called. If a subparser runs exclusively and is chosen in top down way, set *SUBPARSER_SUB_RUNS_BASE* flag. If a subparser runs coexisting way and is chosen in bottom up way, set *SUBPARSER_BASE_RUNS_SUB*. Use *SUBPARSER_BI_DIRECTION* if both cases can be considered.

SystemdUnit parser runs as a subparser of iniconf base parser. SystemdUnit parser specifies *SUBPARSER_SUB_RUNS_BASE* because unit files of systemd have very specific file extensions though they are written in iniconf syntax. Therefore we expect SystemdUnit parser is chosen in top down way. The same logic is applicable to YumRepo parser.

Autoconf parser specifies *SUBPARSER_BI_DIRECTION*. For input file having name *configure.ac*, by pattern matching, Autoconf parser is chosen in top down way. In other hand, for file name *foo.m4*, Autoconf parser can be chosen in bottom up way.

inputStart is called before the base parser starting parsing a new input file. *inputEnd* is called after the base parser finishing parsing the input file. Universal-ctags main part calls these methods. Therefore, a base parser doesn't have to call them.

exclusiveSubparserChosenNotify is called when a parser is chosen as an exclusive parser. Calling this method is a job of a base parser.

Extending *subparser* type

The m4 parser extends *subparser* type like following:

parsers/m4.h:

```
typedef struct sM4Subparser m4Subparser;
struct sM4Subparser {
        subparser subparser;

        bool (* probeLanguage) (m4Subparser *m4, const char* token);

        /* return value: Cork index */
        int  (* newMacroNotify) (m4Subparser *m4, const char* token);

        bool (* doesLineCommentStart)   (m4Subparser *m4, int c, const char␣
↪*token);
        bool (* doesStringLiteralStart) (m4Subparser *m4, int c);
};
```

Put *subparser* as the first member of the extended struct(here sM4Subparser). In addition the first field, 4 methods are defined in the extended struct.

Till choosing a subparser for the current input file, the m4 parser calls *probeLanguage* method of its subparsers each time when find a token in the input file. A subparser returns *true* if it recognizes the input file is for the itself by analyzing tokens passed from the base parser.

parsers/autoconf.c:

```
extern parserDefinition* AutoconfParser (void)
{
        static const char *const patterns [] = { "configure.in", NULL };
        static const char *const extensions [] = { "ac", NULL };
        parserDefinition* const def = parserNew("Autoconf");

        static m4Subparser autoconfSubparser = {
                .subparser = {
                        .direction = SUBPARSER_BI_DIRECTION,
                        .exclusiveSubparserChosenNotify =␣
↪exclusiveSubparserChosenCallback,
                },
                .probeLanguage  = probeLanguage,
                .newMacroNotify = newMacroCallback,
                .doesLineCommentStart = doesLineCommentStart,
                .doesStringLiteralStart = doesStringLiteralStart,
        };
```

probeLanguage function defined in *autoconf.c* is connected to the *probeLanguage* member of *autoconfSubparser*. The *probeLanguage* function of Autoconf is very simple:

parsers/autoconf.c:

```
static bool probeLanguage (m4Subparser *m4, const char* token)
{
        return strncmp (token, "m4_", 3) == 0
                || strncmp (token, "AC_", 3) == 0
                || strncmp (token, "AM_", 3) == 0
                || strncmp (token, "AS_", 3) == 0
                || strncmp (token, "AH_", 3) == 0
                ;
}
```

This function checks the prefix of passed tokens. If known prefix is found, Autoconf assumes this is an Autoconf input and returns *true*.

parsers/m4.c:

```
if (m4tmp->probeLanguage
        && m4tmp->probeLanguage (m4tmp, token))
{
        chooseExclusiveSubparser ((m4Subparser *)tmp, NULL);
        m4found = m4tmp;
}
```

The m4 parsers calls probeLanguage function of a subparser. If *true* is returned *chooseExclusiveSubparser* function which is defined in the main part. *chooseExclusiveSubparser* calls *exclusiveSubparserChosenNotify* method of the chosen subparser.

The method is implemented in Autoconf subparser like following:

parsers/autoconf.c:

```
static void exclusiveSubparserChosenCallback (subparser *s, void *data)
{
        setM4Quotes ('[', ']');
}
```

It changes quote characters of the m4 parser.

Making a tag in a subparser

Via calling callback functions defined in subparsers, their base parser gives chance to them making tag entries.

The m4 parser calls *newMacroNotify* method when it finds an m4 macro is used. The Autoconf parser connects *newMacroCallback* function defined in *parser/autoconf.c*.

parsers/autoconf.c:

```
static int newMacroCallback (m4Subparser *m4, const char* token)
{
        int keyword;
        int index = CORK_NIL;

        keyword = lookupKeyword (token, getInputLanguage ());

        /* TODO:
           AH_VERBATIM
         */
        switch (keyword)
        {
        case KEYWORD_NONE:
                break;
        case KEYWORD_init:
                index = makeAutoconfTag (PACKAGE_KIND);
                break;

...

extern parserDefinition* AutoconfParser (void)
{
        ...
        static m4Subparser autoconfSubparser = {
                .subparser = {
                        .direction = SUBPARSER_BI_DIRECTION,
                        .exclusiveSubparserChosenNotify =␣
↪exclusiveSubparserChosenCallback,
                },
                .probeLanguage  = probeLanguage,
                .newMacroNotify = newMacroCallback,
```

In *newMacroCallback* function, the Autoconf parser receives the name of macro found by the base parser and analysis weather the macro is interesting in the context of Autoconf language or not. If it is interesting name, the Autoconf parser makes a tag for it.

Calling methods of subparsers from a base parser

A base parser can use *foreachSubparser* macro for accessing its subparsers. A base should call *enterSubparser* before calling a method of a subparser, and call *leaveSubparser* after calling the method. The macro and functions are declare in *main/subparser.h* .

parsers/m4.c:

```
static m4Subparser * maySwitchLanguage (const char* token)
{
        subparser *tmp;
        m4Subparser *m4found = NULL;

        foreachSubparser (tmp, false)
        {
                m4Subparser *m4tmp = (m4Subparser *)tmp;
```

(continues on next page)

(continued from previous page)

```
                enterSubparser(tmp);
                if (m4tmp->probeLanguage
                        && m4tmp->probeLanguage (m4tmp, token))
                {
                        chooseExclusiveSubparser (tmp, NULL);
                        m4found = m4tmp;
                }
                leaveSubparser();

                if (m4found)
                        break;
        }

        return m4found;
}
```

foreachSubparser takes a variable having type *subparser*. For each iteration, the value for the variable is updated.

enterSubparser takes a variable having type *subparser*. With the calling *enterSubparser*, the current language(the value returned from *getInputLanguage*) can be temporary switched to the language specified with the variable. One of the effect of switching is that *language* field of tags made in the callback function called between *enterSubparser* and *leaveSubparser* is adjusted.

Registering a subparser to its base parser

Use *DEPTYPE_SUBPARSER* dependency in a subparser for registration.

parsers/autoconf.c:

```
extern parserDefinition* AutoconfParser (void)
{
        parserDefinition* const def = parserNew("Autoconf");

        static m4Subparser autoconfSubparser = {
                .subparser = {
                        .direction = SUBPARSER_BI_DIRECTION,
                        .exclusiveSubparserChosenNotify =␣
↪exclusiveSubparserChosenCallback,
                },
                .probeLanguage  = probeLanguage,
                .newMacroNotify = newMacroCallback,
                .doesLineCommentStart = doesLineCommentStart,
                .doesStringLiteralStart = doesStringLiteralStart,
        };
        static parserDependency dependencies [] = {
                [0] = { DEPTYPE_SUBPARSER, "M4", &autoconfSubparser },
        };

        def->dependencies = dependencies;
        def->dependencyCount = ARRAY_SIZE (dependencies);
```

DEPTYPE_SUBPARSER is specified in the 0th element of'dependencies' function static variable. In the next a literal string "M4" is specified and *autoconfSubparser* follows. The intent of the code is registering *autoconfSubparser* subparser definition to a base parser named "M4".

dependencies function static variable must be assigned to *dependencies* fields of a variable of *parserDefinition*. The main part of Universal-ctags refers the field when initializing parsers.

[0] emphasizes this is "the 0th element". The subparser may refer the index of the array when the subparser calls *scheduleRunningBaseparser*.

Scheduling running the base parser

For the case that a subparser is chosen in top down, the subparser must call *scheduleRunningBaseparser* in the main *parser* method.

parsers/autoconf.c:

```
static void findAutoconfTags(void)
{
        scheduleRunningBaseparser (0);
}

extern parserDefinition* AutoconfParser (void)
{
        ...
        parserDefinition* const def = parserNew("Autoconf");
        ...
        static parserDependency dependencies [] = {
                [0] = { DEPTYPE_SUBPARSER, "M4", &autoconfSubparser },
        };

        def->dependencies = dependencies;
        ...
        def->parser = findAutoconfTags;
        ...
        return def;
}
```

A subparser can do nothing actively. A base parser makes its subparser work b calling methods of the subparser. Therefor a subparser must run its base parser when the subparser is chosen in a top down way, The main part prepares *scheduleRunningBaseparser* function for the purpose.

A subparser should call the function from *parser* method of *parserDefinition* of the subparser. *scheduleRunningBaseparser* takes an integer. It specifies an index of the dependency which is used for registering the subparser.

Command line interface

Running subparser can be controlled with *s* extras flag. By default it is enabled. To turning off the feature running subparser, specify *–extras=-s*.

When *–extras=+E* option given, a tag entry recorded by a subparser is marked as follows:

```
TMPDIR  input.ac        /^AH_TEMPLATE([TMPDIR],$/;"     template        ␣
↪extras:subparser         end:4
```

See also *Defining a subparser*.

8.2.4 Examples of sub/base combinations

Automake/Make parser combination

Simply to say the syntax of Automake is the subset of Make. However, the Automake parser has interests in Make macros having special suffixes: "_PROGRAMS", "_LTLIBRARIES", and "_SCRIPTS" so on.

Here is an example of input for Automake:

```
bin_PROGRAMS = ctags
ctags_CPPFLAGS =    \
        -I.         \
```

(continues on next page)

(continued from previous page)

```
        -I$(srcdir) \
        -I$(srcdir)/main
```

From the point of the view of the Make parser, bin_PROGRAMS is a just a macro; the Make parser tags bin_PROGRAMS as a macro. The Make parser doesn't tag "ctags" being right side of = because it is not a new name: just a value assigned to bin_PROGRAMS. However, for the Automake parser "ctags" is a new name; the Automake parser tags "ctags" with kind "Program". The Automake parser can tag it with getting help from the Make parser.

The Automake parser is an exclusive subparser. It is chosen in top down way; an input file name "Makefile.am" gives enough information for choosing the Automake parser.

To give chances to the Automake parser to capture Automake own definitions, The Make parser provides following interface in *parsers/make.h*:

```
struct sMakeSubparser {
        subparser subparser;

        void (* valueNotify) (makeSubparser *s, char* name);
        void (* directiveNotify) (makeSubparser *s, char* name);
        void (* newMacroNotify) (makeSubparser *s,
                                 char* name,
                                 bool withDefineDirective,
                                 bool appending);
};
```

The Automake parser defines methods for tagging Automake own definitions in a *struct sMakeSubparser* type variable, and runs the Make parser by calling *scheduleRunningBaseparser* function.

The Make parser tags Make own definitions in an input file. In addition Make parser calls the methods during parsing the input file.

```
$ ./ctags --fields=+lK  --extras=+r -o - Makefile.am
bin  Makefile.am     /^bin_PROGRAMS = ctags$/;"      directory
↪language:Automake
bin_PROGRAMS Makefile.am     /^bin_PROGRAMS = ctags$/;"      macro   language:Make
ctags        Makefile.am     /^bin_PROGRAMS = ctags$/;"      program
↪language:Automake       directory:bin
ctags_CPPFLAGS       Makefile.am     /^ctags_CPPFLAGS =    \\$/;"    macro
↪language:Make
```

bin_PROGRAMS and *ctags_CPPFLAGS* are tagged as macros of Make. In addition *bin* is tagged as directory, and *ctags* as program of Automake.

bin is tagged in a callback function assigned to *newMacroFound* method. *ctags* is tagged in a callback function assigned to *valuesFound* method.

–extras=+r is used in the example. *r* extra is needed to tag *bin*. *bin* is not defined in the line, *bin_PROGRAMS =*. *bin* is referenced as a name of directory where programs are stored. Therefore *r* is needed.

For tagging *ctags*, the Automake parser must recognize *bin* in *bin_PROGRAMS* first. *ctags* is tagged because it is specified as a value for *bin_PROGRAMS*. As the result *r* is also needed to tag *ctags*.

Only Automake related tags are emitted if Make parser is disabled.

```
$ ./ctags --languages=-Make --fields=+lKr --extras=+r -o - Makefile.am
bin     Makefile.am     /^bin_PROGRAMS = ctags$/;"      directory
↪language:Automake       roles:program
ctags   Makefile.am     /^bin_PROGRAMS = ctags$/;"      program language:Automake
↪     directory:bin
```

Autoconf/M4 parser combination

Universal-ctags uses m4 parser as a base parser and Autoconf parse as a sub parser for *configure.ac* input file.

```
AC_DEFUN([PRETTY_VAR_EXPAND],
         [$(eval "$as_echo_n" $(eval "$as_echo_n" "${$1}"))])
```

The m4 parser finds no definition here. However, Autoconf parser finds *PRETTY_VAR_EXPAND* as a macro definition. Syntax like *(...)* is part of M4 language. So Autoconf parser is implemented as a sub parser of m4 parser. The most parts of tokens in input files are handled by M4. Autoconf parser gives hints for parsing *configure.ac* and registers callback functions to Autoconf parser.

CHAPTER 9

Building ctags

9.1 Building with configure (*nix including GNU/Linux)

Like most Autotools-based projects, you need to do:

```
$ ./autogen.sh
$ ./configure --prefix=/where/you/want # defaults to /usr/local
$ make
$ make install # may require extra privileges depending on where to install
```

After installation the *ctags* executable will be in *$prefix/bin/*.

autogen.sh runs *autoreconf* internally. If you use a (binary oriented) GNU/Linux distribution, *autoreconf* may be part of the *autoconf* package. In addition you may have to install *automake* and/or *pkg-config*, too.

9.1.1 Changing the executable's name

On some systems, like certain BSDs, there is already a 'ctags' program in the base system, so it is somewhat inconvenient to have the same name for Universal-ctags. During the `configure` stage you can now change the name of the created executable.

To add a prefix 'ex' which will result in 'ctags' being renamed to 'exctags':

```
$ ./configure --program-prefix=ex
```

To completely change the program's name run the following:

```
$ ./configure --program-transform-name='s/ctags/my_ctags/; s/etags/myemacs_tags/'
```

Please remember there is also an 'etags' installed alongside 'ctags' which you may also want to rename as shown above.

9.2 Building/hacking/using on MS-Windows

Maintainer Frank Fesevur <ffes@users.sourceforge.net>

This part of the documentation is written by Frank Fesevur, co-maintainer of universal-ctags and the maintainer of the Windows port of this project. It is still very much a work in progress. Things still need to be written down, tested or even investigated. When building for Windows you should be aware that there are many compilers and build environments available. This is a summary of available options and things that have been tested so far.

9.2.1 Compilers

There are many compilers for Windows. Compilers not mentioned here may work but are not tested.

Microsoft Visual Studio

http://www.visualstudio.com/

Obviously there is Microsoft Visual Studio 2013. Many professional developers targeting Windows use Visual Studio. Visual Studio comes in a couple of different editions. Their Express and Community editions are free to use, but a Microsoft-account is required to download the .iso and when you want to continue using it after a 30-days trial period. Other editions of Visual Studio must be purchased.

Installing Visual Studio will give you the IDE, the command line compilers and the MS-version of make named nmake.

Note that ctags cannot be built with Visual Studio older than 2013 anymore. There is C99 (or C11) coding used that generates syntax errors with VS2012 and older. This could affect compilers from other vendors as well.

GCC

There are three flavors of GCC for Windows:

- MinGW http://www.mingw.org
- MinGW-w64 http://mingw-w64.sourceforge.net
- TDM-GCC http://tdm-gcc.tdragon.net

MinGW started it all, but development stalled for a while and no x64 was available. Then the MinGW-w64 fork emerged. It started as a 64-bit compiler, but soon they included both a 32-bit and a 64-bit compiler. But the name remained, a bit confusing. Another fork of MinGW is TDM-GCC. It also provides both 32-bit and 64-bit compilers. All have at least GCC 4.8. MinGW-w64 appears to be the most used flavor of MinGW at this moment. Many well known programs that originate from GNU/Linux use MinGW-w64 to compile their Windows port.

9.2.2 Building ctags from the command line

Microsoft Visual Studio

Most users of Visual Studio will use the IDE and not the command line to compile a project. But by default a shortcut to the command prompt that sets the proper path is installed in the Start Menu. When this command prompt is used `nmake -f mk_mvc.mak` will compile ctags. You can also go into the `win32` subdirectory and run `msbuild ctags_vs2013.sln` for the default build. Use `msbuild ctags_vs2013.sln /p:Configuration=Release` to specifically build a release build. MSBuild is what the IDE uses internally and therefore will produce the same files as the IDE.

If you want to build an iconv enabled version, you must specify `WITH_ICONV=yes` and `ICONV_DIR` like below:

```
nmake -f mk_mvc.mak WITH_ICONV=yes ICONV_DIR=path/to/iconvlib
```

If you want to build a debug version using `mk_mvc.mak`, you must specify `DEBUG=1` like below:

```
nmake -f mk_mvc.mak DEBUG=1
```

If you want to create PDB files for debugging even for a release version, you must specify `PDB=1` like below:

```
nmake -f mk_mvc.mak PDB=1
```

GCC

General

All the GCC's come with installers or with zipped archives. Install or extract them in a directory without spaces.

GNU Make builds for Win32 are available as well, and sometimes are included with the compilers. Make sure it is in your path, for instance by copying the make.exe in the bin directory of your compiler.

Native win32 versions of the GNU/Linux commands cp, rm and mv can be useful. rm is almost always used in by the `clean` target of a makefile.

CMD

Any Windows includes a command prompt. Not the most advanced, but it is enough to do the build tasks. Make sure the path is set properly and `make -f mk_mingw.mak` should do the trick.

If you want to build an iconv enabled version, you must specify `WITH_ICONV=yes` like below:

```
make -f mk_mingw.mak WITH_ICONV=yes
```

If you want to build a debug version, you must specify `DEBUG=1` like below:

```
make -f mk_mingw.mak DEBUG=1
```

MSYS / MSYS2

From their site: MSYS is a collection of GNU utilities such as bash, make, gawk and grep to allow building of applications and programs which depend on traditional UNIX tools to be present. It is intended to supplement MinGW and the deficiencies of the cmd shell.

MSYS comes in two flavors; the original from MinGW and MSYS2. See http://www.msys2.org/ about MSYS2.

MSYS is old but still works. You can build ctags with it using `make -f mk_mingw.mak`. The Autotools are too old on MSYS so you cannot use them.

MSYS2 is a more maintained version of MSYS, but specially geared towards MinGW-w64. You can also use Autotools to build ctags. If you use Autotools you can enable parsers which require jansson, libxml2 or libyaml, and can also do the Units testing with `make units`.

The following packages are needed to build a full-featured version:

- base-devel (make, autoconf)
- mingw-w64-{i686,x86_64}-toolchain (mingw-w64-{i686,x86_64}-gcc, mingw-w64-{i686,x86_64}-pkg-config)
- mingw-w64-{i686,x86_64}-jansson
- mingw-w64-{i686,x86_64}-libxml2
- mingw-w64-{i686,x86_64}-libyaml
- mingw-w64-{i686,x86_64}-xz

If you want to build a single static-linked binary, you can use the following command:

```
./autogen.sh
./configure --disable-external-sort EXTRA_CFLAGS=-DLIBXML_STATIC LDFLAGS=-static␣
↪LIBS='-lz -llzma -lws2_32'
make
```

`--disable-external-sort` is a recommended option for Windows builds.

Cygwin

Cygwin provides ports of many GNU/Linux tools and a POSIX API layer. This is the most complete way to get the GNU/Linux terminal feel under Windows. Cygwin has a setup that helps you install all the tools you need. One drawback of Cygwin is that it has poor performance.

It is easy to build a Cygwin version of ctags using the normal GNU/Linux build steps. This ctags.exe will depend on cygwin1.dll and should only be used within the Cygwin ecosystem.

Cygwin has packages with a recent version of MinGW-w64 as well. This way it is easy to cross-compile a native Windows application with `make -f mk_mingw.mak CC=i686-w64-mingw32-gcc`.

You can also build a native Windows version using Autotools.

```
./autogen.sh
./configure --host=i686-w64-mingw32 --disable-external-sort
make
```

If you use Autotools you can also do the Units testing with `make units`.

Some anti-virus software slows down the build and test process significantly, especially when `./configure` is running and during the Units tests. In that case it could help to temporarily disable them. But be aware of the risks when you disable your anti-virus software.

Cross-compile from GNU/Linux

All major distributions have both MinGW and MinGW-w64 packages. Cross-compiling works the same way as with Cygwin. You cannot do the Windows based Units tests on GNU/Linux.

9.2.3 Building ctags with IDEs

I have no idea how things work for most GNU/Linux developers, but most Windows developers are used to IDEs. Not many use a command prompt and running the debugger from the command line is not a thing a Windows developers would normally do. Many IDEs exist for Windows, I use the two below.

Microsoft Visual Studio

As already mentioned Microsoft Visual Studio 2013 has the free Express and Community editions. For ctags the Windows Desktop Express Edition is enough to get the job done. The IDE has a proper debugger. Project files for VS2013 can be found in the win32 directory.

Please know that when files are added to the sources.mak, these files need to be added to the .vcxproj and .vcxproj.filters files as well. The XML of these files should not be a problem.

Code::Blocks

http://www.codeblocks.org/

Code::Blocks is a decent GPL-licensed IDE that has good gcc and gdb integration. The TDM-GCC that can be installed together with Code::Blocks works fine and I can provide a project file. This is an easy way to have a free - free as in beer as well as in speech - solution and to have the debugger within the GUI as well.

9.2.4 Other differences between Microsoft Windows and GNU/Linux

There other things where building ctags on Microsoft Windows differs from building on GNU/Linux.

- Filenames on Windows file systems are case-preserving, but not case-sensitive.
- Windows file systems use backslashes "\" as path separators, but paths with forward slashes "/" are no problem for a Windows program to recognize, even when a full path (include drive letter) is used.
- The default line-ending on Windows is CRLF. A tags file generated by the Windows build of ctags will contain CRLF.
- The tools used to build ctags do understand Unix-line endings without problems. There is no need to convert the line-ending of existing files in the repository.
- Due to the differences between the GNU/Linux and Windows C runtime library there are some things that need to be added to ctags to make the program as powerful as it is on GNU/Linux. At this moment regex and fnmatch are borrowed from glibc.
- Because there is no default scandir() for Windows, the optlib feature is not yet available for Windows. Various implementations of scandir() for Windows do exist, but still have to be investigated.
- Units testing needs a decent `bash` shell. It is only tested using Cygwin or MSYS2.

9.3 Building on Mac OS

Maintainer Cameron Eagans <me@cweagans.net>

This part of the documentation is written by Cameron Eagans, a co-maintainer of Universal-ctags and the maintainer of the OSX packaging of this project.

9.3.1 Build Prerequisites

Building ctags on OSX should be no different than building on GNU/Linux. The same toolchains are used, and the Mac OS packaging scripts use autotools and make (as you'd expect).

You may need to install the xcode command line tools. You can install the entire xcode distribution from the App Store, or for a lighter install, you can simply run `xcode-select --install` to *only* install the compilers and such. See http://stackoverflow.com/a/9329325 for more information. Once your build toolchain is installed, proceed to the next section.

At this point, if you'd like to build from an IDE, you'll have to figure it out. Building ctags is a pretty straightforward process that matches many other projects and most decent IDEs should be able to handle it.

Building Manually (i.e. for development)

You can simply run the build instructions in README.md.

Building with Homebrew

Homebrew (http://brew.sh/) is the preferred method for installing Universal-ctags for end users. Currently, the process for installing with Homebrew looks like this:

```
brew tap universal-ctags/universal-ctags
brew install --HEAD universal-ctags
```

Eventually, we hope to move the Universal-ctags formula to the main Homebrew repository, but since we don't have any tagged releases at this point, it's a head-only formula and wouldn't be accepted. When we have a tagged release, we'll submit a PR to Homebrew.

If you'd like to help with the Homebrew formula, you can find the repository here: https://github.com/universal-ctags/homebrew-universal-ctags

9.3.2 Differences between OSX and GNU/Linux

There other things where building ctags on OSX differs from building on GNU/Linux.

- Filenames on HFS+ (the Mac OS filesystem) are case-preserving, but not case-sensitive in 99% of configurations. If a user manually formats their disk with a case sensitive version of HFS+, then the filesystem will behave like normal GNU/Linux systems. Depending on users doing this is not a good thing.

9.3.3 Contributing

This documentation is very much a work in progress. If you'd like to contribute, submit a PR and mention @cweagans for review.

CHAPTER 10

Testing ctags

It it difficult for us to know syntax of all languages supported in ctags. Test facility and test cases are quite import for maintaining ctags in limited resources.

10.1 *Units* test facility

Maintainer Masatake YAMATO <yamato@redhat.com>

Exuberant ctags has a test facility. The test case were *Test* directory. So Here I call it *Test*.

Main aim of the facility is detecting regression. All files under Test directory are given as input for old and new version of ctags commands. The output tags files of both versions are compared. If any difference is found the check fails. *Test* expects the older ctags binary to be correct.

This expectation is not always met. Consider that a parser for a new language is added. You may want to add a sample source code for that language to *Test*. An older ctags version is unable to generate a tags file for that sample code, but the newer ctags version does. At this point a difference is found and *Test* reports failure.

The units test facility(*Units*) I describe here takes a different approach. An input file and an expected output file are given by a contributor of a language parser. The units test facility runs ctags command with the input file and compares its output and the expected output file. The expected output doesn't depend on ctags.

If a contributor sends a patch which may improve a language parser, and if a reviewer is not familiar with that language, s/he cannot evaluate it.

Unit test files, the pair of input file and expected output file may be able to explain the intent of patch well; and may help the reviewer.

10.1.1 How to write a test case

The test facility recognizes an input file and an expected output file by patterns of file name. Each test case should have its own directory under Units directory.

*Units/TEST/input.** **requisite**

Input file name must have a *input* as basename. *TEST* part should explain the test case well.

Units/TEST/input[-_][0-9]. Units/TEST/input[-_][0-9][-_]*.** **optional**

Optional input file names. They are put next to *input.** in testing command line.

Units/TEST/expected.tags **optional**

Expected output file must have a name *expected.tags*. It should be the same directory of the input file.

If this file is not given, the exit status of ctags process is just checked; the output is ignored.

If you want to test etags output (specified with `-e`), Use **.tags-e** as suffix instead of **.tags**. In such a case you don't have to write `-e` to `args.ctags`. The test facility sets `-e` automatically.

If you want to test cross reference output (specified with `-x`), Use **.tags-x** as suffix instead of **.tags**. In such a case you don't have to write `-x` to `args.ctags`. The test facility sets `-x` automatically.

If you want to test json output (specified with `--output-format=json`), Use **.tags-json** as suffix instead of **.tags**. In such a case you don't have to write `--output-format=json` to `args.ctags`, and `json` to `features`. The test facility sets the option and the feature automatically.

Units/TEST/args.ctags **optional**

`-o -` is used as default optional argument when running a unit test ctags. If you want to add more options, enumerate options in **args.ctags** file. This file is an optional.

Remember you have to put one option in one line; don't put multiple options to one line. Multiple options in one line doesn't work.

Units/TEST/filter-.** **optional**

You can rearrange the output of ctags with this command before comparing with *executed.tags*. This command is invoked with no argument. The output ctags is given via stdin. Rearrange data should be written to stdout.

Units/TEST/features **optional**

If a unit test case requires special features of ctags, enumerate them in this file line by line. If a target ctags doesn't have one of the features, the test is skipped.

If a file line is started with `!`, the effect is inverted; if a target ctags has the feature specified with `!`, the test is skipped.

All features built-in can be listed with passing `--list-features` to ctags.

Units/TEST/languages **optional**

If a unit test case requires that language parsers are enabled/available, enumerate them in this file line by line. If one of them is disabled/unavailable, the test is skipped.

language parsers enabled/available can be checked with passing `--list-languages` to ctags.

Units/TEST/dictfile **optional**

Used in spell checking. See *cspell* for more details.

10.1.2 Note for importing a test case from Test directory

I think all test cases under Test directory should be converted to Units.

If you convert use following TEST name convention.

- use *.t* instead of *.d* as suffix for the name

Here is an example:

```
Test/simple.sh
```

This should be:

```
Units/simple.sh.t
```

With this name convention we can track which test case is converted or not.

10.1.3 Example of files

See *Units/c-sample/input.c* and *Units/c-sample/expected*.

10.1.4 How to run unit tests

test make target:

```
$ make units
```

The result of unit tests is reported by lines. You can specify test cases with `UNITS=`.

An example to run *vim-command.d* only:

```
$ make units UNITS=vim-command
```

Another example to run *vim-command.d* and *parser-python.r/bug1856363.py.d*:

```
$ make units UNITS=vim-command,bug1856363.py
```

During testing *OUTPUT.tmp*, *EXPECTED.tmp* and *DIFF.tmp* files are generated for each test case directory. These are removed when the unit test is **passed**. If the result is **FAILED**, it is kept for debugging. Following command line can clean up these generated files at once:

```
$ make clean-units
```

Other than **FAILED** and **passed** two types of result are defined.

skipped

means running the test case is skipped in some reason.

failed (KNOWN bug)

mean the result if failed but the failure is expected. See "Gathering test cases for known bugs".

10.1.5 Example of running

```
$ make units
Category: ROOT
------------------------------------------------------------------------
Testing 1795612.js as JavaScript                              passed
Testing 1850914.js as JavaScript                              passed
Testing 1878155.js as JavaScript                              passed
Testing 1880687.js as JavaScript                              passed
Testing 2023624.js as JavaScript                              passed
Testing 3184782.sql as SQL                                    passed
...
```

10.1.6 Running unit tests for specific languages

You can run only the tests for specific languages by setting `LANGUAGES` to parsers as reported by `ctags --list-languages`:

```
make units LANGUAGES=PHP,C
```

Multiple languages can be selected using a comma separated list.

10.1.7 Gathering test cases for known bugs

When we met a bug, making a small test case that triggers the bug is important development activity. Even the bug cannot be fixed in soon, the test case is an important result of work. Such result should be merged to the source tree. However, we don't love **FAILED** message, too. What we should do?

In such a case, merge as usually but use *.b* as suffix for the directory of test case instead of *.d*.

Unix/css-singlequote-in-comment-issue2.b is an example of *.b* suffix usage.

When you run test.units target, you will see:

```
Testing c-sample as C                                    passed
Testing css-singlequote-in-comment as CSS                failed (KNOWN bug)
Testing ctags-simple as ctags                            passed
```

Suffix *.i* is a variant of *.b*. *.i* is for merging/gathering input which lets ctags process enter an infinite loop. Different from *.b*, test cases marked as *.i* are never executed. They are just skipped but reported the skips:

```
Testing ada-ads as Ada                                   passed
Testing ada-function as Ada                              skipped (may cause an␣
↪infinite loop)
Testing ada-protected as Ada                             passed
...

Summary (see CMDLINE.tmp to reproduce without test harness)
------------------------------------------------------------
  #passed:                                  347
  #FIXED:                                   0
  #FAILED (unexpected-exit-status):         0
  #FAILED (unexpected-output):              0
  #skipped (features):                      0
  #skipped (languages):                     0
  #skipped (infinite-loop):                 1
    ada-protected
  ...
```

10.1.8 Running under valgrind and timeout

If `VG=1` is given, each test cases are run under valgrind. If valgrind detects an error, it is reported as:

```
$ make units VG=1
Testing css-singlequote-in-comment as CSS                failed (valgrind-error)
...
Summary (see CMDLINE.tmp to reproduce without test harness)
------------------------------------------------------------
...
#valgrind-error:                          1
  css-singlequote-in-comment
...
```

In this case the report of valgrind is recorded to `Units/css-singlequote-in-comment/VALGRIND-CSS.tmp`.

NOTE: `/bin/bash` is needed to report the result. You can specify a shell running test with SHELL macro like:

```
$ make units VG=1 SHELL=/bin/bash
```

If `TIMEOUT=N` is given, each test cases are run under timeout command. If ctags doesn't stop in `N` second, it is stopped by timeout command and reported as:

```
$ make units TIMEOUT=1
Testing css-singlequote-in-comment as CSS             failed (TIMED OUT)
...
Summary (see CMDLINE.tmp to reproduce without test harness)
-------------------------------------------------------------
...
#TIMED-OUT:                              1
  css-singlequote-in-comment
...
```

If `TIMEOUT=N` is given, *.i* test cases are run. They will be reported as *TIMED-OUT*.

10.1.9 Categories

With *.r* suffix, you can put test cases under a sub directory of *Units*. `Units/parser-ada.r` is an example. If *misc/units* test harness, the sub directory is called a category. `parser-ada.r` is the name category in the above example.

CATEGORIES macro of make is for running units in specified categories. Following command line is for running units in `Units/parser-sh.r` and `Units/parser-ada.r`:

```
$ make units CATEGORIES='parser-sh,parser-ada'
```

10.1.10 Finding minimal bad input

When a test case is failed, the input causing `FAILED` result is passed to *misc/units shrink*. *misc/units shrink* tries to make the shortest input which makes ctags exits with non-zero status. The result is reported to `Units/\*/SHRINK-${language}.tmp`. Maybe useful to debug.

10.1.11 Acknowledgments

The file name rule is suggested by Maxime Coste <frrrwww@gmail.com>.

10.2 Semi-fuzz(*Fuzz*) testing

Maintainer Masatake YAMATO <yamato@redhat.com>

Unexpected input can lead ctags to enter an infinite loop. The fuzz target tries to identify these conditions by passing semi-random (semi-broken) input to ctags.

```
$ make fuzz LANGUAGES=LANG1[,LANG2,...]
```

With this command line, ctags is run for random variations of all test inputs under *Units/*/input.** of languages defined by `LANGUAGES` macro variable. In this target, the output of ctags is ignored and only the exit status is analyzed. The ctags binary is also run under timeout command, such that if an infinite loop is found it will exit with a non-zero status. The timeout will be reported as following:

```
[timeout C]                Units/test.vhd.t/input.vhd
```

This means that if C parser doesn't stop within N seconds when *Units/test.vhd.t/input.vhd* is given as an input, timeout will interrupt ctags. The default duration can be changed using `TIMEOUT=N` argument in *make* command. If there is no timeout but the exit status is non-zero, the target reports it as following:

```
[unexpected-status(N) C]                Units/test.vhd.t/input.vhd
```

The list of parsers which can be used as a value for `LANGUAGES` can be obtained with following command line

```
$ ./ctags --list-languages
```

Besides `LANGUAGES` and `TIMEOUT`, fuzz target also takes the following parameters:

`VG=1`

Run ctags under valgrind. If valgrind finds a memory error it is reported as:

```
[valgrind-error Verilog]                Units/array_spec.f90.t/
↪input.f90
```

The valgrind report is recorded at `Units/\*/VALGRIND-${language}.tmp`.

As the same as units target, this semi-fuzz test target also calls *misc/units shrink* when a test case is failed. See "*Units* test facility" about the shrunk result.

10.3 *Noise* testing

Maintainer Masatake YAMATO <yamato@redhat.com>

After enjoying developing Semi-fuzz testing, I'm looking for a more unfair approach. Run

```
$ make noise LANGUAGES=LANG1[,LANG2,...]
```

It takes a long time, especially with `VG=1`, so this cannot be run under Travis CI. However, it is a good idea to run it locally.

The noise target generates test cases by inserting or deleting one character to the test cases of *Units*.

TBW

10.4 *Chop* and *slap* testing

Maintainer Masatake YAMATO <yamato@redhat.com>

After reviving many bug reports, we recognized some of them spot unexpected EOF. The chop target was developed based on this recognition.

The chop target generates many input files from an existing input file under *Units* by truncating the existing input file at variety file positions.

```
$ make chop  LANGUAGES=LANG1[,LANG2,...]
```

It takes a long time, especially with `VG=1`, so this cannot be run under Travis CI. However, it is a good idea to run it locally.

slap target is derived from chop target. While chop target truncates the existing input files from tail, the slap target does the same from head.

10.5 *Tmain*: a facility for testing main part

Maintainer Masatake YAMATO <yamato@redhat.com>

Tmain is introduced to test the area where *Units* does not cover well.

Units works fine for testing parsers. However, it assumes something input is given to ctags command, and a *tags* file is generated from ctags command.

Other aspects cannot be tested. Such areas are files and directories layout after installation, standard error output, exit status, etc.

You can run test cases with following command line:

```
$ make tmain
```

Tmain is still under development so I will not write the details here.

To write a test case, see files under *Tmain/tmain-example.d*. In the example, *Tmain* does:

1. runs new subshell and change the working directory to *Tmain/tmain-example.d*,
2. runs *run.sh* with *bash*,
3. captures stdout, stderr and exit status, and
4. compares them with *stdout-expected.txt*, *stderr-expected.txt*, and *exit-expected.txt*.
5. compares it with *tags-expected.txt* if run.sh generates *tags* file.

run.sh is run with following 4 arguments:

1. the path for the target ctags
2. the path for *builddir* directory
3. the path for the target readtags

The path for readtags is not reliable; readtags command is not available if –disable-readcmd was given in configure time. A case, testing the behavior of readtags, must verify the command existence with *test -x $4* before going into the main part of the test.

When comparing *tags* file with *tags-expected.txt*, you must specify the path of *tags* explicitly with -o option in ctags command line like:

```
CTAGS=$1
BUILDDIR=$3
${CTAGS} ... -o $BUILDDIR/tags ...
```

This makes it possible to keep the original source directory clean.

See also *tmain_run* and *tmain_compare* functions in *misc/units*.

If run.sh exits with code 77, the test case is skipped. The output to stdout is captured and printed as the reason of skipping.

10.5.1 TODO

- Run under valgrind

10.6 *Tinst* installation test

Maintainer Masatake YAMATO <yamato@redhat.com>

tinst target is for testing the result of `make install`.

```
$ make tinst
```

10.7 *Cspell* spell checking

Maintainer Masatake YAMATO <yamato@redhat.com>

- `make cspell` reports unknown words. After verifying the reported words are correct you should add them to files under *dictfiles* directory.
- `cspell` target assumes the names used in ctags source code are correctly spelled. Such names can be added semi-automatically; use `make dicts` targets. It updates files prefixed with *GENERATED-* under *dictfiles*.
- Either semi-automatically generated or adding by manually, files under *dictfiles* directory should be installed to Universal-ctags git repository.
- `make cspell` makes and users *SPELL_CHECKING.TMP* at the top of source code directory as temporary working space.
- `cspell` target depends on GNU aspell library. If the library is linked to, `ctags --list-features` prints `aspell`.

An example session:

```
$ make cspell
./misc/gen-repoinfo > main/repoinfo.h
  CC       main/ctags-repoinfo.o
  CCLD     ctags
/bin/sh misc/cspell
checking bda36b226cfc0f492908bc5779268c353e615623...unknown words
        orignal
        dictfile
checking 61a71ef37df9fd9ebb0d3e8a941effd643662cda...ok
checking b08c956e155d47a8e689cfede2501ab48890c889...ok
checking c07a6e94140e1d5cfeaf3cb42d2fc28bd0e92e51...ok
...
checking 201e5e774fef84527802113969998bd71b14466d...ok
Makefile:6510: recipe for target 'cspell' failed
make: *** [cspell] Error 1
```

Here cspell reports “orignal” and “dictfile” as unknown words.

“orignal” should be “original”. So you should make a fixup commit for bda36b with “git commit –fixup=bda36b”. Then you may want to do “git rebase -i –autosquash master”.

“dictfile” may be a name used in source code files. Ideally `make dicts` picks up the name and puts to one of *GENERATED-* dictionaries. However, it is not implemented yet. What you can do now is adding it to one of dictionaries under *dictfiles* directory. After do “git add” the directory file and make a fixup commit.

There are cases that you want to add a misspelled word intentionally to source tree: to test cases(Units and Tmain) and to documentations.

About test cases, make a file named *dictfile* under the directory of target test case, and put the words line by line. You can find an example in *Units/simple-ctags-aspell.d/dictfile* of Universal-ctags source tree.

For documentations, there is no good way. Suggestions are welcome. "CSPELL:" prefix line is a temporary solution. A line starting from "CSPELL:" in a commit log is treated specially by `make cspell` when spell-checking the commit; whitespace separated words in the line are added to a temporary dictionary.

An example

```
commit 8efb57fa1c9d7b9b7ba01f49963d7d7779609f21
Author: Masatake YAMATO <yamato@redhat.com>
Date:   Mon Jun 5 23:08:51 2017 +0900

docs(man): fix styles of definition list

CSPELL: xno xyes

Signed-off-by: Masatake YAMATO <yamato@redhat.com>
```

Here "xno" and "xyes" are added to a dictionary temporary used during spell-checking the commit, "8efb57"; "xno" and "xyes" are never reported as unknown words. The temporary dictionary is used only for this commit.

10.8 Input validation for *Units*

Maintainer Masatake YAMATO <yamato@redhat.com>

We have to maintain parsers for languages that we don't know well. We don't have enough time to learn the languages.

Units test cases help us not introduce wrong changes to a parser.

However, there is still an issue; a developer who doesn't know a target language well may write a broken test input file for the language. Here comes "Input validation."

You can validate the test input files of *Units* with *validate-input* make target if a validator for a language is defined.

10.8.1 How to run and an example session

Here is an example validating an input file for JSON.

```
$ make validate-input VALIDATORS=jq
...
Category: ROOT
------------------------------------------------------------
simple-json.d/input.json with jq                          valid

Summary
------------------------------------------------------------
  #valid:                                 1
  #invalid:                               0
  #skipped (known invalidation)           0
  #skipped (validator unavailable)        0
```

This example shows validating *simple-json.d/input.json* as an input file with *jq* validator. With VALIDATORS variable passed via command-line, you can specify validators to run. Multiple validators can be specified using a comma-separated list. If you don't give VALIDATORS, the make target tries to use all available validators.

The meanings of "valid" and "invalid" in "Summary" are apparent. In two cases, the target skips validating input files:

#skipped (known invalidation)

A test case specifies KNOWN-INVALIDATION in its *validator* file.

#skipped (validator unavailable)

A command for a validator is not available.

10.8.2 *validator* file

validator file in a *Units* test directory specifies which validator the make target should use.

```
$ cat Units/simple-json.d/validator
jq
```

If you put *validator* file to a category directory (a directory having *.r* suffix), the make target uses the validator specified in the file as default. The default validator can be overridden with a *validator* file in a subdirectory.

```
$ cat Units/parser-puppetManifest.r/validator
puppet
# cat Units/parser-puppetManifest.r/puppet-append.d/validator
KNOWN-INVALIDATION
```

In the example, the make target uses *puppet* validator for validating the most of all input files under *Units/parser-puppetManifest.r* directory. An exception is an input file under *Units/parser-puppetManifest.r/puppet-append.d* directory. The directory has its specific *validator* file.

If a *Unit* test case doesn't have *expected.tags* file, the make target doesn't run the validator on the file even if a default validator is given in its category directory.

If a *Unit* test case specifies KNOWN-INVALIDATION in its *validator* file, the make target just increments "#skipped (known invalidation)" counter. The target reports the counter at the end of execution.

10.8.3 validator command

A validator specified in a *validator* file is a command file put under *misc/validators* directory. The command must have "validator-" as prefix in its file name. For an example, *misc/validators/validator-jq* is the command for "jq".

The command file must be an executable. *validate-input* make target runs the command in two ways.

is_runnable method

Before running the command as a validator, the target runs the command with "is_runnable" as the first argument. A validator command can let the target know whether the validator command is runnable or not with exit status. 0 means ready to run. Non-zero means not ready to run.

The make target never runs the validator command for validation purpose if the exit status is non-zero.

For an example, *misc/validators/validator-jq* command uses *jq* command as its backend. If *jq* command is not available on a system, *validator-jq* can do nothing. If such case, *is_runnable* method of *validator-jq* command should exit with non-zero value.

validate method

The make target runs the command with "validate* and an input file name for validating the input file. The command exits non-zero if the input file contains invalid syntax. This method

will never run if *is_runnable* method of the command exits with non-zero.

CHAPTER 11

Extending ctags

Exuberant-ctags allows a user to add a new parser to ctags with `--langdef=<LANG>` and `--regex-<LANG>=...` options.

Universal-ctags follows and extends the design of Exuberant-ctags in more powerful ways, as described in the following chapters.

Universal-ctags encourages users to share the new parsers defined by their options. See *optlib* to know how you can share your parser definition with others.

Note that some of the new features are experimental, and will be marked as such in the documentation.

11.1 Extending ctags with Regex parser (*optlib*)

Maintainer Masatake YAMATO <yamato@redhat.com>

11.1.1 Option files

An "option" file is a file in which command line options are written line by line. `ctags` loads it and runs as if the options in the file were passed in command line.

Following file is an example of option file.

```
# Exclude directories that don't contain real code
--exclude=Units
        # indentation is ignored
        --exclude=tinst-root
--exclude=Tmain
```

can be used as a start marker of a line comment. Whitespaces at the start of lines are ignored during loading.

There are two categories of option files, though they both contain command line options: **preload** and **optlib** option files.

Preload option file

Preload option files are option files loaded by `ctags` automatically at start-up time. Which files are loaded at start-up time are very different from Exuberant-ctags.

At start-up time, Universal-ctags loads files having `.ctags` as a file extension under the following statically defined directories:

1. `$HOME/.ctags.d`
2. `$HOMEDRIVE$HOMEPATH/ctags.d` (in `Windows`)
3. `.ctags.d`
4. `ctags.d`

`ctags` visits the directories in the order listed above for preloading files. `ctags` loads files having `.ctags` as file extension in alphabetical order (strcmp(3) is used for comparing, so for example `.ctags.d/ZZZ.ctags` will be loaded *before* `.ctags.d/aaa.ctags`).

Quoted from man page of Exuberant-ctags:

```
FILES
           /ctags.cnf (on MSDOS, MSWindows only)
           /etc/ctags.conf
           /usr/local/etc/ctags.conf
           $HOME/.ctags
           $HOME/ctags.cnf (on MSDOS, MSWindows only)
           .ctags
           ctags.cnf (on MSDOS, MSWindows only)
                       If any of these configuration files exist, each will
                       be expected to contain a set of default options
                       which are read in the order listed when ctags
                       starts, but before the CTAGS environment variable is
                       read or any command line options are read.  This
                       makes it possible to set up site-wide, personal or
                       project-level defaults. It is possible to compile
                       ctags to read an additional configuration file
                       before any of those shown above, which will be
                       indicated if the output produced by the --version
                       option lists the "custom-conf" feature. Options
                       appearing in the CTAGS environment variable or on
                       the command line will override options specified in
                       these files. Only options will be read from these
                       files.  Note that the option files are read in
                       line-oriented mode in which spaces are significant
                       (since shell quoting is not possible). Each line of
                       the file is read as one command line parameter (as
                       if it were quoted with single quotes). Therefore,
                       use new lines to indicate separate command-line
                       arguments.
```

What follows explains the differences and their intentions...

Directory oriented configuration management

Exuberant-ctags provides a way to customize ctags with options like `--langdef=<LANG>` and `--regex-<LANG>`. These options are powerful and make ctags popular for programmers.

Universal-ctags extends this idea; we have added new options for defining a parser, and have extended existing options. Defining a new parser with the options is more than "customizing" in Universal-ctags.

To make it easier to maintain a parser defined using the options, you can put each parser language in a different options file. Universal-ctags doesn't preload a single file. Instead, Universal-ctags loads all files having the `.`

`ctags` extension under the previously specified directories. If you have multiple parser definitions, put them in different files.

Avoiding option incompatibility issues

The Universal-ctags options are different from those of Exuberant-ctags, therefore Universal-ctags doesn't load any of the files Exuberant-ctags loads at start-up. Otherwise there would be incompatibility issues if Exuberant-ctags loaded an option file that used a newly introduced option in Universal-ctags, and vice versa.

No system wide configuration

To make the preload path list short and because it was rarely ever used, Universal-ctags does not load any option files for system wide configuration. (i.e., no `/etc/ctags.d`)

Use `.ctags` for the file extension

Extensions `.cnf` and `.conf` are obsolete. Use the unified extension `.ctags` only.

Optlib option file

From a syntax perspective, there is no difference between optlib option files and preload option files; `ctags` options are written line by line in a file.

Optlib option files are option files not loaded at start-up time automatically. To load an optlib option file, specify a pathname for an optlib option file with `--options=PATHNAME` option explicitly. The pathname can be just the filename if it's in the current directory.

Exuberant-ctags has the `--options` option, but you can only specify a single file to load. Universal-ctags extends the option two aspects: you can specify a directory to load all files in that directory, and you can specify a path search list to look in. See next section for details.

Specifying a directory

If you specify a directory instead of a file as the argument for the `--options=PATHNAME`, Universal-ctags will load all files having a `.ctags` extension under the directory in alphabetical order.

Specifying an optlib path search list

For loading a file (or directory) specified in `--options=PATHNAME`, `ctags` searches "optlib path list" first if the option argument (PATHNAME) doesn't start with '`/`' or '`.`'. If `ctags` finds a file, `ctags` loads it.

If `ctags` doesn't find a file in the path list, `ctags` loads a file (or directory) at the specified pathname.

By default, optlib path list is empty. To set or add a directory path to the list, use `--optlib-dir=PATH`.

For setting (adding one after clearing):

```
--optlib-dir=PATH
```

For adding:

```
--optlib-dir=+PATH
```

Tips for writing an option file

- Use `--quiet --options=NONE` to disable preloading.
- Two options are introduced for debugging the process of loading option files.

  `--_echo=MSG`

  Prints MSG to standard error immediately.

  `--_force-quit=[NUM]`

  Exit immediately with the status of the specified NUM.

- Universal-ctags has an `optlib2c` script that translates an option file into C source code. Your optlib parser can thus easily become a built-in parser, by contributing to Universal-ctags' github. You could be famous! Examples are in the `optlib` directory in Universal-ctags source tree.

11.1.2 Regular expression (regex) engine

Universal-ctags currently uses the same regex engine as Exuberant-ctags does: the POSIX.2 regex engine in GNU glibc-2.10.1. By default it uses the Extended Regular Expressions (ERE) syntax, as used by most engines today; however it does *not* support many of the "modern" extensions such as lazy captures, non-capturing grouping, atomic grouping, possessive quantifiers, look-ahead/behind, etc. It is also notoriously slow when backtracking, and has some known "quirks" with respect to escaping special characters in bracket expressions.

For example, a pattern of `[^\]]+` is invalid in POSIX.2, because the `]` is *not* special inside a bracket expression, and thus should **not** be escaped. Most regex engines ignore this subtle detail in POSIX.2, and instead allow escaping it with `\]` inside the bracket expression and treat it as the literal character `]`. GNU glibc, however, does not generate an error but instead considers it undefined behavior, and in fact it will match very odd things. Instead you **must** use the more unintuitive `[^]]+` syntax. The same is technically true of other special characters inside a bracket expression, such as `[^\)]+`, which should instead be `[^)]+`. The `[^\)]+` will appear to work usually, but only because what it is really doing is matching any character but `\` *or* `)`. The only exceptions for using `\` inside a bracket expression are for `\t` and `\n`, which ctags converts to their single literal character control codes before passing the pattern to glibc.

Another detail to keep in mind is how the regex engine treats newlines. Universal-ctags compiles the regular expressions in the `--regex-<LANG>` and `--mline-regex-<LANG>` options with REG_NEWLINE set. What that means is documented in the POSIX spec. One obvious effect is that the regex special dot any-character `.` does not match newline characters, the `^` anchor *does* match right after a newline, and the `$` anchor matches right before a newline. A more subtle issue is this text from the Regular Expressions chapter: "the use of literal <newline>s or any escape sequence equivalent produces undefined results". What that means is using a regex pattern with `[^\n]+` is invalid, and indeed in glibc produces very odd results. **Never** use `\n` in patterns for `--regex-<LANG>`, and never use them in non-matching bracket expressions for `--mline-regex-<LANG>` patterns. For the experimental `--_mtable-regex-<LANG>` you can safely use `\n` because that regex is not compiled with REG_NEWLINE.

You should always test your regex patterns against test files with strings that do and do not match. Pay particular emphasis to when it should *not* match, and how *much* it matches when it should. A common error is forgetting that a POSIX.2 ERE engine is always greedy; the * and + quantifiers match as much as possible, before backtracking from the end of their match.

For example this pattern:

```
foo.*bar
```

Will match this **entire** string, not just the first part:

```
foobar, bar, and even more bar
```

11.1.3 Regex option argument flags

Many regex-based options described in this document support additonal arguments in the form of long flags. Long flags are specified with surrounding `{` and `}`.

The general format and placement is as follows:

```
--regex-<LANG>=<PATTERN>/<NAME>/[<KIND>/]LONGFLAGS
```

Some examples:

```
--regex-Pod=/^=head1[ \t]+(.+)/\1/c/
--regex-Foo=/set=[^;]+/\1/v/{icase}
--regex-Man=/^\.TH[[:space:]]{1,}"([^"]{1,})".*/\1/t/{exclusive}{icase}{scope=push}
--regex-Gdbinit=/^#//{exclusive}
```

Note that the last example only has two `/` forward slashes following the regex pattern, as a shortened form when no kind-spec exists.

The `--mline-regex-<LANG>` option also follows the above format. The experimental `--_mtable-regex-<LANG>` option follows a slightly modified version as well.

The `--langdef=<LANG>` option also supports long flags, but not using forward-slash separators.

Regex control flags

The regex matching can be controlled by adding flags to the `--regex-<LANG>`, `--mline-regex-<LANG>`, and experimental `--_mtable-regex-<LANG>` options. This is done by either using the single character short flags `b`, `e` and `i` flags as explained in the *ctags.1* man page, or by using long flags described earlier. The long flags require more typing but are much more readable.

The mapping between the older short flag names and long flag names is:

short flag	long flag	description
b	basic	Posix basic regular expression syntax.
e	extend	Posix extended regular expression syntax (default).
i	icase	Case-insensitive matching.

So the following `--regex-<LANG>` expression:

```
--regex-m4=/^m4_define\(\[([^]$\(]+).+$/\1/d,definition/x
```

is the same as:

```
--regex-m4=/^m4_define\(\[([^]$\(]+).+$/\1/d,definition/{extend}
```

The characters `{` and `}` may not be suitable for command line use, but long flags are mostly intended for option files.

Exclusive flag in regex

By default, lines read from the input files will be matched with **all** regular expressions defined with `--regex-<LANG>`. Each matched regular expression will successfully emit a tag.

In some cases another policy, exclusive-matching, is preferable to the all-matching policy. Exclusive-matching means the rest of regular expressions are not tried if one of regular expressions is matched successfully, for that input line.

For specifying exclusive-matching the flags `exclusive` (long) and `x` (short) were introduced. For example, this is used in `optlib/gdbinit.ctags` for ignoring comment lines in `gdb` files, as follows:

```
--regex-Gdbinit=/^#//{exclusive}
```

Comments in gbd files start with # so the above line is the first regex match line in `gdbinit.ctags`, so that subsequent regex matches are not tried for the input line.

If an empty name pattern(//) is used for the `--regex-<LANG>` option, ctags warns it as a wrong usage of the option. However, if the flags `exclusive` or `x` is specified, the warning is suppressed.

NOTE: This flag does not make sense in the multi-line `--mline-regex-<LANG>` option nor the multi-table `--_mtable-regex-<LANG>` option.

Experimental flags

Note: These flags are experimental. They apply to all regex option types: basic `--regex-<LANG>`, multi-line `--mline-regex-<LANG>`, and the experimental multi-table `--_mtable-regex-<LANG>` option.

`_extra`

> This flag indicates the tag should only be generated if the given 'extra' type is enabled, as explained in *Conditional tagging with extras*.

`_field`

> This flag allows a regex match to add additional custom fields to the generated tag entry, as explained in *Adding custom fields to the tag output*.

`_role`

> This flag allows a regex match to generate a reference tag entry and specify the role of the reference, as explained in *Capturing reference tags*.

Ghost kind in regex parser

If a whitespace is used as a kind letter, it is never printed when ctags is called with `--list-kinds` option. This kind is automatically assigned to an empty name pattern.

Normally you don't need to know this.

11.1.4 Scope tracking in a regex parser

With the `scope` long flag, you can record/track scope context. A stack is used for tracking the scope context.

`{scope=push}`

> Push the tag captured with a regex pattern to the top of the stack. If you don't want to record this tag but just push, use *placeholder* long option together.

`{scope=ref}`

> Refer to the thing at the top of the stack as a scope where the tag captured with a regex pattern is. The stack is not modified with this specification. If the stack is empty, this flag is just ignored.

`{scope=pop}`

> Pop the thing at the top of the stack. If the stack is empty, this flag is just ignored.

`{scope=clear}`

> Make the stack empty.

`{scope=set}`

> Clear then push.

`{placeholder}`

Don't print a tag captured with a regex pattern to a tag file. This is useful when you need to push non-named context information to the stack. Well known non-named scope in C language is established with *{*. A non- named scope never appears in tags file as a name or scope name. However, pushing it is important to balance `push` and `pop`.

Example 1:

```
# in /tmp/input.foo
class foo:
def bar(baz):
        print(baz)
class goo:
def gar(gaz):
        print(gaz)
```

```
# in /tmp/foo.ctags:
--langdef=Foo
--map-Foo=+.foo

--regex-Foo=/^class[[:blank:]]+([[:alpha:]]+):/\1/c,class/{scope=set}
--regex-Foo=/^[[:blank:]]+def[[:blank:]]+([[:alpha:]]+).*:/\1/d,definition/
↪{scope=ref}
```

```
$ ctags --options=/tmp/foo.ctags -o - /tmp/input.foo
bar     /tmp/input.foo  /^    def bar(baz):$/;"  d       class:foo
foo     /tmp/input.foo  /^class foo:$/;"         c
gar     /tmp/input.foo  /^    def gar(gaz):$/;"  d       class:goo
goo     /tmp/input.foo  /^class goo:$/;"         c
```

Example 2:

```
// in /tmp/input.pp
class foo {
        int bar;
}
```

```
# in /tmp/pp.ctags:
--langdef=pp
--map-pp=+.pp

--regex-pp=/^[[:blank:]]*\}//{scope=pop}{exclusive}
--regex-pp=/^class[[:blank:]]*([[:alnum:]]+)[[[:blank:]]]*\{/\1/c,class,classes/
↪{scope=push}
--regex-pp=/^[[:blank:]]*int[[:blank:]]*([[:alnum:]]+)/\1/v,variable,variables/
↪{scope=ref}
```

```
$ ctags --options=/tmp/pp.ctags -o - /tmp/input.pp
bar     /tmp/input.pp   /^    include bar$/;"   v       class:foo
foo     /tmp/input.pp   /^class foo {$/;"       c
```

NOTE: This flag doesn't work well with `--mline-regex-<LANG>=`.

11.1.5 Overriding the letter for file kind

One of the built-in tag kinds in Universal-ctags is the `F` file kind. Overriding the letter for file kind is not allowed in Universal-ctags.

Warning: Don't use `F` as a kind letter in your parser. (See issue #317 on github)

11.1.6 Generating fully qualified tags automatically from scope information

If scope fields are filled properly with *{scope=...}* regex flags, you can use the field values for generating fully qualified tags. About the *{scope=..}* flag itself, see "FLAGS FOR –regex-<LANG> OPTION" section of *ctags-optlib(7)* man page or Universal-ctags parser definition language.

Specify *{_autoFQTag}* to the end of `--langdef=<LANG>` option like `--langdef=Foo{_autoFQTag}` to make ctags generate fully qualified tags automatically.

. is the default separator combining names into a fully qualified tag. It is not customizable yet.

input.foo:

```
class X
   var y
end
```

foo.ctags:

```
--langdef=foo{_autoFQTag}
--map-foo=+.foo
--kinddef-foo=c,class,classes
--kinddef-foo=v,var,variables
--regex-foo=/class ([A-Z]*)/\1/c/{scope=push}
--regex-foo=/end///{placeholder}{scope=pop}
--regex-foo=/[ \t]*var ([a-z]*)/\1/v/{scope=ref}
```

Output:

```
$ u-ctags --quiet --options=NONE --options=./foo.ctags -o - input.foo
X       input.foo       /^class X$/;"   c
y       input.foo       /^      var y$/;"       v       class:X

$ u-ctags --quiet --options=NONE --options=./foo.ctags --extras=+q -o - input.foo
X       input.foo       /^class X$/;"   c
X.y     input.foo       /^      var y$/;"       v       class:X
y       input.foo       /^      var y$/;"       v       class:X
```

"X.y" is printed as a fully qualified tag when `--extras=+q` is given.

11.1.7 Multi-line pattern match

We often need to scan multiple lines to generate a tag, whether due to needing contextual information to decide whether to tag or not, or to constrain generating tags to only certain cases, or to grab multiple substrings to generate the tag name.

Universal-ctags has two ways to accomplish this: multi-line regex options, and an experimental multi-table regex options described later.

The newly introduced `--mline-regex-<LANG>` is similar to `--regex-<LANG>` except the pattern is applied to the whole file's contents, not line by line.

This example is based on an issue #219 posted by @andreicristianpetcu:

```
// in input.java:

@Subscribe
public void catchEvent(SomeEvent e)
```

(continues on next page)

(continued from previous page)

```
{
return;
}

@Subscribe
public void
recover(Exception e)
{
return;
}
```

The above java code is similar to the Java Spring framework. The `@Subscribe` annotation is a keyword for the framework, and the developer would like to have a tag generated for each method annotated with `@Subscribe`, using the name of the method followed by a dash followed by the type of the argument. For example the developer wants the tag name `Event-SomeEvent` generated for the first method shown above.

To accomplish this, the developer creates a `spring.ctags` file with the following:

```
# in spring.ctags:
--langdef=javaspring
--map-javaspring:+.java
--mline-regex-javaspring=/@Subscribe([[:space:]])*([a-z ]+)[[:space:]]*([a-zA-
↪Z]*)\(([a-zA-Z]*)/\3-\4/s,subscription/{mgroup=3}
--fields=+ln
```

And now using `spring.ctags` the tag file has this:

```
$ ./ctags -o - --options=./spring.ctags input.java
Event-SomeEvent input.java      /^public void catchEvent(SomeEvent e)$/;"       s␣
↪     line:2  language:javaspring
recover-Exception       input.java      /^    recover(Exception e)$/;"  s      ␣
↪line:10 language:javaspring
```

Multiline pattern flags

Note: These flags also apply to the experimental `--_mtable-regex-<LANG>` option described later.

`{mgroup=N}`

This flag indicates the pattern should be applied to the whole file contents, not line by line. `N` is the number of a capture group in the pattern, which is used to record the line number location of the tag. In the above example `3` is specified. The start position of the regex capture group 3, relative to the whole file is used.

Warning: You **must** add an `{mgroup=N}` flag to the multi-line `--mline-regex-<LANG>` option, even if the `N` is `0` (meaning the start position of the whole regex pattern). You do not need to add it for the multi-table `--_mtable-regex-<LANG>`.

`{_advanceTo=N[start|end]}`

A regex pattern is applied to whole file's contents iteratively. This long flag specifies from where the pattern should be applied in the next iteration for regex matching. When a pattern matches, the next pattern matching starts from the start or end of capture group `N`. By default it advances to the end of of the whole match (i.e., `{_advanceTo=0end}` is the default).

Let's think about following input

```
def def abc
```

Consider two sets of options, foo and bar.

```
# foo.ctags:
--langdef=foo
--langmap=foo:.foo
--kinddef-foo=a,something,something
--mline-regex-foo=/def *([a-z]+)/\1/a/{mgroup=1}
```

```
# bar.ctags:
--langdef=bar
--langmap=bar:.bar
--kinddef-bar=a,something,something
--mline-regex-bar=/def *([a-z]+)/\1/a/{mgroup=1}{_advanceTo=1start}
```

foo.ctags emits following tags output:

```
def   input.foo       /^def def abc$/;"       a
```

bar.ctgs emits following tags output:

```
def   input-0.bar     /^def def abc$/;"       a
abc   input-0.bar     /^def def abc$/;"       a
```

`_advanceTo=1start` is specified in *bar.ctags*. This allows ctags to capture "abc".

At the first iteration, the patterns of both *foo.ctags* and *bar.ctags* match as follows

```
0   1       (start)
v   v
def def abc
          ^
          0,1   (end)
```

"def" at the group 1 is captured as a tag in both languages. At the next iteration, the positions where the pattern matching is applied to are not the same in the languages.

foo.ctags

```
          0end (default)
          v
def def abc
```

bar.ctags

```
        1start (as specified in _advanceTo long flag)
        v
def def abc
```

This difference of positions makes the difference of tags output.

A more relevant use-case is when `{_advanceTo=N[start|end]}` is used in the experimental `--_mtable-regex-<LANG>`, to "advance" back to the beginning of a match, so that one can generate multiple tags for the same input line(s).

Note: This flag doesn't work well with scope related flags and `exclusive` flags.

11.1.8 Advanced pattern matching with multiple regex tables

Note: This is a highly experimental feature. This will not go into the man page of 6.0. But let's be honest, it's the most exciting feature!

In some cases, the `--regex-<LANG>` and `--mline-regex-<LANG>` options are not sufficient to generate the tags for a particular language. Some of the common reasons for this are:

- To ignore commented lines or sections for the language file, so that tags aren't generated for symbols that are within the comments.
- To enter and exit scope, and use it for tagging based on contextual state or with end-scope markers that are difficult to match to their associated scope entry point.
- To support nested scopes.
- To change the pattern searched for, or the resultant tag for the same pattern, based on scoping or contextual location.
- To break up an overly complicated `--mline-regex-<LANG>` pattern into separate regex patterns, for performance or readability reasons.

To help handle such things, Universal-ctags has been enhanced with multi-table regex matching. The feature is inspired by *lex*, the fast lexical analyzer generator, which is a popular tool on Unix environments for writing parsers, and RegexLexer of Pygments. Knowledge about them will help you understand the new options.

The new options are:

`--_tabledef-<LANG>`

Declares a new regex matching table of a given name for the language, as described in *Declaring a new regex table*.

`--_mtable-regex-<LANG>`

Adds a regex pattern and associated tag generation information and flags, to the given table, as described in *Adding a regex to a regex table*.

`--_mtable-extend-<LANG>`

Includes a previously-defined regex table to the named one.

The above will be discussed in more detail shortly.

First, let's explain the feature with an example. Consider a imaginary language "*X*" has a similar syntax as JavaScript: "var" is used as defining variable(s), , and "/* ... */" is used for block comments.

Here is our input, `input.x`:

```
/* BLOCK COMMENT
var dont_capture_me;
*/
var a /* ANOTHER BLOCK COMMENT */, b;
```

We want ctags to capture `a` and `b` - but it is difficult to write a parser that will ignore `dont_capture_me` in the comment with a classical regex parser defined with `--regex-<LANG>` or `--mline-regex-<LANG>`, because of the block comments.

The `--regex-<LANG>` option only works on one line at a time, so cannnot know `dont_capture_me` is within comments. The `--mline-regex-<LANG>` could do it in theory, but due to the greedy nature of the regex engine it is impractical and potentially inefficient to do so, given that there could be multiple block comments in the file, with * inside them, etc.

A parser written with multi-table regex, on the other hand, can capture only `a` and `b` safely. But it is more complicated to understand.

Here is a 1st version of `X.ctags`:

```
--langdef=X
--map-X=.x
--kinddef-X=v,var,variables
```

Not so interesting. It doesn't really *do* anything yet. It just creates a new language named `X`, for files ending with a `.x` suffix, and defines a new tag for variable kinds.

When writing a multi-table parser, you have to think about the necessary states of parsing. For the parser of language `X`, we need the following states:

- *toplevel* (initial state)
- *comment* (inside comment)
- *vars* (var statements)

Declaring a new regex table

Before adding regular expressions, you have to declare tables for each state with the `--_tabledef-<LANG>=<TABLE>` option.

Here is the 2nd version of `X.ctags` doing so:

```
--langdef=X
--map-X=.x
--kinddef-X=v,var,variables

--_tabledef-X=toplevel
--_tabledef-X=comment
--_tabledef-X=vars
```

For table names, only characters in the range `[0-9a-zA-Z_]` are acceptable.

For a given language, for each file's input the ctags multi-table parser begins with the *first* declared table. For `X.ctags`, `toplevel` is the one. The other tables are only ever entered/checked if another table specified to do so, starting with the first table. In other words, if the first declared table does not find a match for the current input, and does not specify to go to another table, the other tables for that language won't be used. The flags to go to another table are `{tenter}`, `{tleave}`, and `{tjump}`, as described later.

Adding a regex to a regex table

The new option to add a regex to a declared table is `--_mtable-regex-<LANG>`, and it follows this form:

```
--_mtable-regex-<LANG>=<TABLE>/<PATTERN>/<NAME>/[<KIND>]/LONGFLAGS
```

The parameters for `--_mtable-regex-<LANG>` look complicated. However, `<PATTERN>`, `<NAME>`, and `<KIND>` are the same as the parameters of the `--regex-<LANG>` and `--mline-regex-<LANG>` options. `<TABLE>` is simply the name of a table previously declared with the `--_tabledef-<LANG>` option.

A regex pattern added to a parser with `--_mtable-regex-<LANG>` is matched against the input at the current byte position, not line. Even if you do not specify the `^` anchor at the start of the pattern, `ctags` adds `^` to the pattern automatically. Unlike the `--regex-<LANG>` and `--mline-regex-<LANG>` options, a `^` anchor does not mean "begging of line" in `--_mtable-regex-<LANG>`; instead it means the beginning of the input string (i.e., the current byte position).

The `LONGFLAGS` include the already discussed flags for `--regex-<LANG>` and `--mline-regex-<LANG>`: `{scope=...}`, `{mgroup=N}`, `{_advanceTo=N}`, `{basic}`, `{extend}`, and `{icase}`. The `{exclusive}` flag does not make sense for multi-table regex.

In addition, several new flags are introduced exclusively for multi-table regex use:

`{tenter}`

Push the current table on the stack, and enter another table.

`{tleave}`

Leave the current table, pop the stack, and go to the table that was just popped from the stack.

`{tjump}`

Jump to another table, without affecting the stack.

`{treset}`

Clear the stack, and go to another table.

`{tquit}`

Clear the stack, and stop processing the current input file for this language.

To explain the above new flags, we'll continue using our example in the next section.

Skipping block comments

Let's continue with our example. Here is the 3rd version of `X.ctags`:

```
--langdef=X
--map-X=.x
--kinddef-X=v,var,variables

--_tabledef-X=toplevel
--_tabledef-X=comment
--_tabledef-X=vars

--_mtable-regex-X=toplevel/\/\*//{tenter=comment}
--_mtable-regex-X=toplevel/.//

--_mtable-regex-X=comment/\*\///{tleave}
--_mtable-regex-X=comment/.//
```

Four `--_mtable-regex-X` lines are added for skipping the block comments. Let's discuss them one by one.

For each new file it scans, `ctags` always chooses the first pattern of the first table of the parser. Even if it's an empty table, `ctags` will only try the first declared table. (in such a case it would immedietaly fail to match anything, and thus stop proessing the input file and effectively do nothing)

The first declared table (`toplevel`) has the following regex added to it first:

```
--_mtable-regex-X=toplevel/\/\*//{tenter=comment}
```

A pattern of `\/\*` is added to the `toplevel` table, to match the beginning of a block comment. A backslash character is used in front of the leading `/` to escape the separation character `/` that separates the fields of `--_mtable-regex-<LANG>`. Another backslash inside the pattern is used before the asterisk `*`, to make it a literal asterisk character in regex.

The last `//` means `ctags` should not tag something matching this pattern. In `--regex-<LANG>` you never use `//` because it would be pointless to match something and not tag it using and single-line `--regex-<LANG>`; in multi-line `--mline-regex-<LANG>` you rarely see it, because it would rarely be useful. But in multi-table regex it's quite common, since you frequently want to transition from one state to another (i.e., `tenter` or `tjump` from one table to another).

The long flag added to our first regex of our first table is `tenter`, which is a long flag for switching the table and pushing on the stack. `{tenter=comment}` means "switch the table from toplevel to comment".

So given the input file `input.x` shown earlier, `ctags` will begin at the `toplevel` table and try to match the first regex. It will succeed, and thus push on the stack and go to the `comment` table.

It will begin at the top of the `comment` table (it always begins at the top of a given table), and try each regex line in sequence until it finds a match. If it fails to find a match, it will pop the stack and go to the table that was just

popped from the stack, and begin trying to match at the top of *that* table. If it continues failing to find a match, and ultimately reaches the end of the stack, it will stop processing for this file. For the next input file, it will begin again from the top of the first declared table.

Getting back to our example, the top of the `comment` table has this regex:

```
--_mtable-regex-X=comment/\*\///{tleave}
```

Similar to the previous `toplevel` table pattern, this one for `\*\/` uses a backslash to escape the separator `/`, as well as one before the `*` to make it a literal asterisk in regex. So what it's looking for, from a simple string perspective, is the sequence `*/`. Note that this means even though you see three backslashes `///` at the end, the first one is escaped and used for the pattern itself, and the `--_mtable-regex-X` only has `//` to separate the regex pattern from the long flags, instead of the usual `///`. Thus it's using the shorthand form of the `--_mtable-regex-X` option. It could instead have been:

```
--_mtable-regex-X=comment/\*\////{tleave}
```

The above would have worked exactly the same.

Getting back to our example, remember we're looking at the `input.x` file, currently using the `comment` table, and trying to match the first regex of that table, shown above, at the following location:

```
   ,ctags is trying to match starting here
  v
/* BLOCK COMMENT
var dont_capture_me;
*/
var a /* ANOTHER BLOCK COMMENT */, b;
```

The pattern doesn't match for the position just after `/*`, because that position is a space character. So `ctags` tries the next pattern in the same table:

```
--_mtable-regex-X=comment/.//
```

This pattern matches any any one character including newline; the current position moves one character forward. Now the character at the current position is `B`. The first pattern of the table `*/` still does not match with the input. So `ctags` uses next pattern again. When the current position moves to the `*/` of the 3rd line of `input.x`, it will finally match this:

```
--_mtable-regex-X=comment/\*\///{tleave}
```

In this pattern, the long flag `{tleave}` is specified. This triggers table switching again. `{tleave}` makes `ctags` switch the table back to the last table used before doing `{tenter}`. In this case, `toplevel` is the table. `ctags` manages a stack where references to tables are put. `{tenter}` pushes the current table to the stack. `{tleave}` pops the table at the top of the stack and chooses it.

So now `ctags` is back to the `toplevel` table, and tries the first regex of that table, which was this:

```
--_mtable-regex-X=toplevel/\/\*//{tenter=comment}
```

It tries to match that against its current position, which is now the newline on line 3, between the `*/` and the word `var`:

```
/* BLOCK COMMENT
var dont_capture_me;
*/ <--- ctags is now at this newline (/n) character
var a /* ANOTHER BLOCK COMMENT */, b;
```

The first regex of the `toplevel` table does not match a newline, so it tries the second regex:

```
--_mtable-regex-X=toplevel/.//
```

This matches a newline successfully, but has no actions to perform. So `ctags` moves one character forward (the newline it just matched), and goes back to the top of the `toplevel` table, and tries the first regex again. Eventually we'll reach the beginning of the second block comment, and do the same things as before.

When `ctags` finally reaches the end of the file (the position after `b;`), it will not be able to match either the first or second regex of the `toplevel` table, and quit processing the input file.

So far, we've successfully skipped over block comments for our new `X` language, but haven't generated any tags. The point of `ctags` is to generate tags, not just keep your computer warm. So now let's move onto actually tagging variables...

Capturing variables in a sequence

Here is the 4th version of `X.ctags`:

```
--langdef=X
--map-X=.x
--kinddef-X=v,var,variables

--_tabledef-X=toplevel
--_tabledef-X=comment
--_tabledef-X=vars

--_mtable-regex-X=toplevel/\/\*//{tenter=comment}
# NEW
--_mtable-regex-X=toplevel/var[ \n\t]//{tenter=vars}
--_mtable-regex-X=toplevel/.//

--_mtable-regex-X=comment/\*\///{tleave}
--_mtable-regex-X=comment/.//

# NEW
--_mtable-regex-X=vars/;//{tleave}
--_mtable-regex-X=vars/\/\*//{tenter=comment}
--_mtable-regex-X=vars/([a-zA-Z][a-zA-Z0-9]*)/\1/v/
--_mtable-regex-X=vars/.//
```

One pattern in `toplevel` was added, and a new table `vars` with four patterns was also added.

The new regex in `toplevel` is this:

```
--_mtable-regex-X=toplevel/var[ \n\t]//{tenter=vars}
```

The purpose of this being in *toplevel* is to switch to the *vars* table when the keyword `var` is found in the input stream. We need to switch states (i.e., tables) because we can't simply capture the variables `a` and `b` with a single regex pattern in the `toplevel` table, because there might be block comments inside the `var` statement (as there are in our `input.x`), and we also need to create *two* tags: one for `a` and one for `b`, even though the word `var` only appears once. In other words, we need to "remember" that we saw the keyword `var`, when we later encounter the names `a` and `b`, so that we know to tag each of them; and saving that "in-variable-statement" state is accomplished by switching tables to the `vars` table.

The first regex in our new `vars` table is:

```
--_mtable-regex-X=vars/;//{tleave}
```

This pattern is used to match a single semi-colon `;`, and if it matches pop back to the `toplevel` table using the `{tleave}` long flag. We didn't have to make this the first regex pattern, because it doesn't overlap with any of the other ones other than the `/.//` last one (which must be last for this example to work).

The second regex in our `vars` table is:

```
--_mtable-regex-X=vars/\/\*//{tenter=comment}
```

We need this because block comments can be in variable definitions:

```
var a /* ANOTHER BLOCK COMMENT */, b;
```

So to skip block comments in such a position, the pattern `\/\*` is used just like it was used in the `toplevel` table: to find the literal `/*` beginning of the block comment and enter the `comment` table. Because we're using `{tenter}` and `{tleave}` to push/pop from a stack of tables, we can use the same `comment` table for both `toplevel` and `vars` to go to, because `ctags` will "remember" the previous table and `{tleave}` will pop back to the right one.

The third regex in our `vars` table is:

```
--_mtable-regex-X=vars/([a-zA-Z][a-zA-Z0-9]*)/\1/v/
```

This is nothing special, but is the one that actually tags something: it captures the variable name and uses it for generating a `variable` (shorthand `v`) tag kind.

The last regex in the `vars` table we've seen before:

```
--_mtable-regex-X=vars/.//
```

This makes `ctags` ignore any other characters, such as whitespace or the comma `,`.

Running our example

```
$ cat input.x
/* BLOCK COMMENT
var dont_capture_me;
*/
var a /* ANOTHER BLOCK COMMENT */, b;

$ u-ctags -o - --fields=+n --options=X.ctags input.x
u-ctags -o - --fields=+n --options=X.ctags input.x
a       input.x /^var a \/* ANOTHER BLOCK COMMENT *\/, b;$/;"    v       line:4
b       input.x /^var a \/* ANOTHER BLOCK COMMENT *\/, b;$/;"    v       line:4
```

It works!

You can find additional examples of multi-table regex in our github repo, under the `optlib` directory. For example `puppetManifest.ctags` is a serious example. It is the primary parser for testing multi-table regex parsers, and used in the actual `ctags` program for parsing puppet manifest files.

11.1.9 Conditional tagging with extras

If a matched pattern should only be tagged when an `extra` is enabled, mark the pattern with `{_extra=XNAME}`. `XNAME` is the name of extra. You must define an `XNAME` with the `--_extradef-<LANG>=XNAME, DESCRIPTION` option before defining a regex option marked `{_extra=XNAME}`.

```
if __name__ == '__main__':
        do_something()
```

To capture above lines in a python program(*input.py*), an extra can be used.

```
--_extradef-Python=main,__main__ entry points
--regex-Python=/^if __name__ == '__main__':/__main__/f/{_extra=main}
```

The above optlib(*python-main.ctags*) introduces `main` extra to Python parser. The pattern matching is done only when the `main` is enabled.

```
$ ./ctags --options=python-main.ctags -o - --extras-Python='+{main}' input.py
__main__        input.py        /^if __name__ == '__main__':$/;"        f
```

11.1.10 Adding custom fields to the tag output

Exuberant-ctags allows one of the specified group in a regex pattern can be used as a part of the name of a tagEntry. Universal-ctags offers using the other groups in the regex pattern.

An optlib parser can have its own fields. The groups can be used as a value of the fields of a tagEntry.

Let's think about *Unknown*, an imaginary language. Here is a source file(`input.unknown`) written in *Unknown*:

> public func foo(n, m); protected func bar(n); private func baz(n,...);

With *–regex-Unknown=...* Exuberant-ctags can capture *foo*, *bar*, and *baz* as names. Universal-ctags can attach extra context information to the names as values for fields. Let's focus on *bar*. *protected* is a keyword to control how widely the identifier *bar* can be accessed. *(n)* is the parameter list of *bar*. *protected* and *(n)* are extra context information of *bar*.

With following optlib file(`unknown.ctags`)), `ctags` can attach *protected* to protection field and *(n)* to signature field.

```
--langdef=unknown
--kinddef-unknown=f,func,functions
--map-unknown=+.unknown

--_fielddef-unknown=protection,access scope
--_fielddef-unknown=signature,signatures

--regex-unknown=/^((public|protected|private) +)?func ([^\(]+)\((.*)\)/\3/f/{_
↪field=protection:\1}{_field=signature:(\4)}

--fields-unknown=+'{protection}{signature}'
```

For the line *protected func bar(n);* you will get following tags output:

```
bar     input.unknown    /^protected func bar(n);$/;"    f       ␣
↪protection:protected    signature:(n)
```

Let's see the detail of `unknown.ctags`.

```
--_fielddef-unknown=protection,access scope
```

`--_fielddef-<LANG>=name,description` defines a new field for a parser specified by *<LANG>*. Before defining a new field for the parser, the parser must be defined with `--langdef=<LANG>`. *protection* is the field name used in tags output. *access scope* is the description used in the output of `--list-fields` and `--list-fields=Unknown`.

```
--_fielddef-unknown=signature,signatures
```

This defines a field named *signature*.

```
--regex-unknown=/^((public|protected|private) +)?func ([^\(]+)\((.*)\)/\3/f/{_
↪field=protection:\1}{_field=signature:(\4)}
```

This option requests making a tag for the name that is specified with the group 3 of the pattern, attaching the group 1 as a value for *protection* field to the tag, and attaching the group 4 as a value for *signature* field to the tag. You can use the long regex flag *_field* for attaching fields to a tag with following notation rule:

```
{_field=FIELDNAME:GROUP}
```

`--fields-<LANG>=[+|-]{FIELDNAME}` can be used to enable or disable specified field.

When defining a new parser own field, it is disabled by default. Enable the field explicitly to use the field. See *Parser own fields* about *–fields-<LANG>* option.

passwd parser is a simple example that uses `--fields-<LANG>` option.

11.1.11 Capturing reference tags

To capture a reference tag with an optlib parser, specify a role with *_role* long regex flag. Let's see an example:

```
--langdef=FOO
--kinddef-FOO=m,module,modules
--_roledef-FOO=m.imported,imported module
--regex-FOO=/import[ \t]+([a-z]+)/\1/m/{_role=imported}
--extras=+r
--fields=+r
```

See the line, *–regex-FOO=...*. In this parser *FOO*, a name of imported module is captured as a reference tag with role *imported*. A role must be defined before specifying it as value for *_role* flag. *–_roledef-<LANG>* option is for defining a role.

The parameter of the option comes from three components: a kind letter, the name of role, and the description of role. The kind letter comes first. Following a period, give the role name. The period represents that the role is defined under the kind specified with the kind letter. In the example, *imported* role is defined under *module* kind specified with *m*.

Of course, the kind specified with the kind letter must be defined before using *–_roledef-<FOO>* option. *–kinddef-<LANG>* option is for defining a kind.

The roles are listed with *–list-roles=<LANG>*. The name and description passed to *–_roledef-<LANG>* option are used in the output like:

```
$ ./ctags --langdef=FOO --kinddef-FOO=m,module,modules \
                        --_roledef-FOO='m.imported,imported module' --list-
↪roles=FOO
#KIND(L/N) NAME     ENABLED DESCRIPTION
m/module   imported on      imported module
```

With specifying *_role* regex flag multiple times with different roles, you can assign multiple roles to a reference tag. See following input of C language

```
i += 1;
```

An ultra fine grained C parser may capture a variable *i* with *lvalue* and *incremented*. You can do it with:

```
--_roledef-C=v.lvalue,locator values
--_roledef-C=v.incremented,incremeted with ++ operator
--regex-C=/([a-zA-Z_][a-zA-Z_0-9])+ *+=/\1/v/{_role=lvalue}{_role=incremeted}
```

11.1.12 Submitting an optlib file to the Universal-ctags project

You are encouraged to submit your `.ctags` file to our github through a pull request.

Universal-ctags provides a facility for "Option library". Read "Option library" about the concept and usage first.

Here I will explain how to merge your .ctags into universal-ctags as part of option library. Here I assume you consider contributing an option library in which a regex based language parser is defined. See How to Add Support for a New Language to Exuberant Ctags (EXTENDING) about the way to how to write a regex based language parser. In this section I explains the next step.

I use Swine as the name of programming language which your parser deals with. Assume source files written in Swine language have a suffix *.swn*. The file name of option library is *swine.ctags*.

Copyright notice, contact mail address and license term

Put these information at the header of *swine.ctags*.

An example taken from *data/optlib/ctags.ctags*

```
#
#
#  Copyright (c) 2014, Red Hat, Inc.
#  Copyright (c) 2014, Masatake YAMATO
#
#  Author: Masatake YAMATO <yamato@redhat.com>
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; either version 2
# of the License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
# USA.
#
#
...
```

"GPL version 2 or later version" is needed here. Option file is not linked to `ctags` command. However, I have a plan to write a translator which generates *.c* file from a given option file. As the result the *.c* file is built into `ctags` command. In such a case "GPL version 2 or later version" may be required.

Units test cases

We, universal-ctags developers don't have enough time to learn all languages supported by `ctags`. In other word, we cannot review the code. Only test cases help us to know whether a contributed option library works well or not. We may reject any contribution without a test case.

Read "Using *Units*" about how to write *Units* test cases. Don't write one big test case. Some smaller cases are helpful to know about the intent of the contributor.

- *Units/sh-alias.d*
- *Units/sh-comments.d*
- *Units/sh-quotes.d*
- *Units/sh-statements.d*

are good example of small test cases. Big test cases are good if smaller test cases exist.

See also *parser-m4.r/m4-simple.d* especially *parser-m4.r/m4-simple.d/args.ctags*. Your test cases need `ctags` having already loaded your option library, swine.ctags. You must specify loading it in the test case own *args.ctags*.

Assume your test name is *swine-simile.d*. Put `--option=swine` in *Units/swine-simile.d/args.ctags*.

Makefile.in

Add your optlib file, *swine.ctags* to `PRELOAD_OPTLIB` variable of *Makefile.in*.

If you don't want your optlib loaded automatically when `ctags` starts up, put your optlib file into `OPTLIB` of *Makefile.in* instead of `PRELOAD_OPTLIB`.

Verification

Let's verify all your work here.

1. Run the tests and check whether your test case is passed or failed:

```
$ make units
```

2. Verify your files are installed as expected:

```
$ mkdir /tmp/tmp
$ ./configure --prefix=/tmp/tmp
$ make
$ make install
$ /tmp/tmp/ctags -o - --option=swine something_input.swn
```

Pull-request

Please, consider submitting your well written optlib parser to Universal-ctags. Your *.ctags* is a treasure and can be shared as a first class software component in Universal-ctags.

Pull-requests are welcome.

11.2 ctags Internal API

11.2.1 Input text stream

Function prototypes for handling input text stream are declared in main/read.h. The file exists in exuberant ctags, too. However, the names functions are changed when overhauling `--line-directive` option. (In addition macros were converted to functions for making data structures for the input text stream opaque.)

Ctags has 3 groups of functions for handling input: input, bypass, and raw. Parser developers should use input group. The rest of two are for ctags main part.

inputFile type and the functions of input group

(The original version of this sub sub sub section was written before *inputFile* type and *File* variable are made private.)

inputFile is the type for representing the input file and stream for a parser. It was declared in main/read.h but now it is defined in main/read.c.

Ctags uses a file static variable *File* having type *inputFile* for maintaining the input file and stream. *File* is also defined in main/read.c as *inputFile* is.

fp and *line* are the essential fields of *File*. *fp* having type well known *MIO* declared in main/mio.h. By calling functions of input group (*getcFromInputFile* and *readLineFromInputFile*), a parser gets input text from *fp*.

The functions of input group updates fields *input* and *source* of *File* These two fields has type *inputFileInfo*. These two fields are for mainly tracking the name of file and the current line number. Usually ctags uses only *input* field. *source* is used only when *#line* directive is found in the current input text stream.

A case when a tool generates the input file from another file, a tool can record the original source file to the generated file with using the *#line* directive. *source* is used for tracking/recording the information appeared on #line directives.

Regex pattern matching are also done behind calling the functions of this group.

The functions of bypass group

The functions of bypass group (*readLineFromBypass* and *readLineFromBypassSlow*) are used for reading text from *fp* field of *File* static variable without updating *input* and *source* fields of *File*.

Parsers may not need the functions of this group. The functions are used in ctags main part. The functions are used to make pattern fields of tags file, for example.

The functions of raw group

The functions of this group(*readLineRaw* and *readLineRawWithNoSeek*) take a parameter having type *MIO*; and don't touch *File* static variable.

Parsers may not need the functions of this group. The functions are used in ctags main part. The functions are used to load option files, for example.

promise API

(Currently the tagging via promise API is disabled by default. Use *–extras=+g* option for enabling it.)

Background and Idea

More than one programming languages can be used in one input text stream. promise API allows a host parser running a guest parser in the specified area of input text stream.

e.g. Code written in c language (C code) is embedded in code written in Yacc language (Yacc code). Let's think about this input stream.

```
/* foo.y */
 %token
         END_OF_FILE 0
         ERROR               255
         BELL                1

 %{
 /* C language */
 int counter;
 %}
 %right      EQUALS
 %left       PLUS MINUS
 ...
 %%
 CfgFile             :       CfgEntryList
                         { InterpretConfigs($1); }
                 ;

 ...
 %%
 int
```

(continues on next page)

(continued from previous page)

```
yyerror(char *s)
{
    (void)fprintf(stderr,"%s: line %d of %s\n",s,lineNum,
                                        (scanFile?scanFile:"(unknown)"));
    if (scanStr)
        (void)fprintf(stderr,"last scanned symbol is: %s\n",scanStr);
    return 1;
}
```

In the input the area started from *%{* to *%}* and the area started from the second *%%* to the end of file are written in C. Yacc can be called host language, and C can be called guest language.

Ctags may choose the Yacc parser for the input. However, the parser doesn't know about C syntax. Implementing C parser in the Yacc parser is one of approach. However, ctags has already C parser. The Yacc parser should utilize the existing C parser. The promise API allows this.

More examples are in *Applying a parser to specified areas of input file*.

Usage

See a commit titled with "Yacc: run C parser in the areas where code is written in C". I applied promise API to the Yacc parser.

The parser for host language must track and record the *start* and the *end* of a guest language. Pairs of *line number* and *byte offset* represents the *start* and *end*. When the *start* and *end* are fixed, call *makePromise* with (1) the guest parser name, (2) start, and (3) end. (This description is a bit simplified the real usage.)

Let's see the actual code from parsers/yacc.c.

```
struct cStart {
        unsigned long input;
        unsigned long source;
};
```

Both fields are for recording *start*. *input* field is for recording the value returned from *getInputLineNumber*. *source* is for *getSourceLineNumber*. See *inputFile* for the difference of the two.

enter_c_prologue shown in the next is a function called when *%{* is found in the current input text stream. Remember, in yacc syntax, *%{* is a marker of C code area.

```
static void enter_c_prologue (const char *line CTAGS_ATTR_UNUSED,
                              const regexMatch *matches CTAGS_ATTR_UNUSED,
                              unsigned int count CTAGS_ATTR_UNUSED,
                              void *data)
{
       struct cStart *cstart = data;


       readLineFromInputFile ();
       cstart->input  = getInputLineNumber ();
       cstart->source = getSourceLineNumber ();
}
```

The function just records the start line. It calls *readLineFromInputFile* because the C code may start the next line of the line where the marker is.

leave_c_prologue shown in the next is a function called when *%}*, the end marker of C code area is found in the current input text stream.

```
static void leave_c_prologue (const char *line CTAGS_ATTR_UNUSED,
                              const regexMatch *matches CTAGS_ATTR_UNUSED,
```

(continues on next page)

(continued from previous page)

```
                                unsigned int count CTAGS_ATTR_UNUSED,
                                void *data)
{
        struct cStart *cstart = data;
        unsigned long c_end;

        c_end = getInputLineNumber ();
        makePromise ("C", cstart->input, 0, c_end, 0, cstart->source);
}
```

After recording the line number of the end of the C code area, *leave_c_prologue* calls *makePromise*.

Of course "C" stands for C language, the name of guest parser. Available parser names can be listed by running ctags with *–list-languages* option. In this example two *0* characters are provided as the 3rd and 5th argument. They are byte offsets of the start and the end of the C language area from the beginning of the line which is 0 in this case. In general, the guest language's section does not have to start at the beginning of the line in which case the two offsets have to be provided. Compilers reading the input character by character can obtain the current offset by calling *getInputLineOffset()*.

Internal design

A host parser cannot run a guest parser directly. What the host parser can do is just asking the ctags main part scheduling of running the guest parser for specified area which defined with the *start* and *end*. These scheduling requests are called promises.

After running the host parser, before closing the input stream, the ctags main part checks the existence of promise(s). If there is, the main part makes a sub input stream and run the guest parser specified in the promise. The sub input stream is made from the original input stream by narrowing as requested in the promise. The main part iterates the above process till there is no promise.

Theoretically a guest parser can be nested; it can make a promise. The level 2 guest is also just scheduled. (However, I have never tested such a nested guest parser).

Why not running the guest parser directly from the context of the host parser? Remember many parsers have their own file static variables. If a parser is called from the parser, the variables may be crashed.

11.2.2 Automatic parser guessing

11.2.3 Managing regular expression parsers

11.2.4 Parser written in C

tokenInfo API

In Exuberant-ctags, a developer can write a parser anyway; only input stream and tagEntryInfo data structure is given.

However, while maintaining Universal-ctags I (Masatake YAMATO) think we should have a framework for writing parser. Of course the framework is optional; you can still write a parser without the framework.

To design a framework, I have studied how @b4n (Colomban Wendling) writes parsers. tokenInfo API is the first fruit of my study.

TBW

11.2.5 Output tag stream

Ctags provides *makeTagEntry* to parsers as an entry point for writing tag information to MIO. *makeTagEntry* calls *writeTagEntry* if the parser does not set *useCork* field. *writeTagEntry* calls *writerWriteTag*. *writerWriteTag* just calls *writeEntry* of writer backends. *writerTable* variable holds the four backends: ctagsWriter, etagsWriter, xrefWriter, and jsonWriter. One of them is chosen depending on the arguments passed to ctags.

If *useCork* is set, the tag information goes to a queue on memory. The queue is flushed when *useCork* in unset. See *cork API* for more details.

cork API

Background and Idea

cork API is introduced for recording scope information easier.

Before introducing cork, a scope information must be recorded as strings. It is flexible but memory management is required. Following code is taken from clojure.c(with modifications).

```
if (vStringLength (parent) > 0)
{
        current.extensionFields.scope[0] = ClojureKinds[K_NAMESPACE].name;
        current.extensionFields.scope[1] = vStringValue (parent);
}

makeTagEntry (&current);
```

parent, values stored to *scope [0]* and *scope [1]* are all kind of strings.

cork API provides more solid way to hold scope information. cork API expects *parent*, which represents scope of a tag(*current*) currently parser dealing, is recorded to a *tags* file before recording the *current* tag via *makeTagEntry* function.

For passing the information about *parent* to *makeTagEntry*, *tagEntryInfo* object was created. It was used just for recording; and freed after recording. In cork API, it is not freed after recording; a parser can reused it as scope information.

How to use

See a commit titled with "clojure: use cork". I applied cork API to the clojure parser.

cork can be enabled and disabled per parser. cork is disabled by default. So there is no impact till you enables it in your parser.

useCork field is introduced in *parserDefinition* type:

```
typedef struct {
...
                boolean useCork;
...
} parserDefinition;
```

Set *TRUE* to *useCork* like:

```
extern parserDefinition *ClojureParser (void)
{
        ...
        parserDefinition *def = parserNew ("Clojure");
        ...
```

(continues on next page)

(continued from previous page)

```
        def->useCork = TRUE;
        return def;
}
```

When ctags running a parser with *useCork* being *TRUE*, all output requested via *makeTagEntry* function calling is stored to an internal queue, not to *tags* file. When parsing an input file is done, the tag information stored automatically to the queue are flushed to *tags* file in batch.

When calling *makeTagEntry* with a *tagEntryInfo* object(*parent*), it returns an integer. The integer can be used as handle for referring the object after calling.

```
static int parent = CORK_NIL;
...
parent = makeTagEntry (&e);
```

The handle can be used by setting to a *scopeIndex* field of *current* tag, which is in the scope of *parent*.

```
current.extensionFields.scopeIndex = parent;
```

When passing *current* to *makeTagEntry*, the *scopeIndex* is refereed for emitting the scope information of *current*.

scopeIndex must be set to *CORK_NIL* if a tag is not in any scope. When using *scopeIndex* of *current*, *NULL* must be assigned to both *current.extensionFields.scope[0]* and *current.extensionFields.scope[1]*. *initTagEntry* function does this initialization internally, so you generally you don't have to write the initialization explicitly.

Automatic full qualified tag generation

If a parser uses the cork for recording and emitting scope information, ctags can reuse it for generating full qualified(FQ) tags. Set *requestAutomaticFQTag* field of *parserDefinition* to *TRUE* then the main part of ctags emits FQ tags on behalf of the parser if *–extras=+q* is given.

An example can be found in DTS parser:

```
extern parserDefinition* DTSParser (void)
{
        static const char *const extensions [] = { "dts", "dtsi", NULL };
        parserDefinition* const def = parserNew ("DTS");
        ...
        def->requestAutomaticFQTag = TRUE;
        return def;
}
```

Setting *requestAutomaticFQTag* to *TRUE* implies setting *useCork* to *TRUE*.

CHAPTER 12

Tips for hacking

12.1 Fussy syntax checking

If -Wall of gcc is not enough, you may be interested in this.

You can change C compiler warning options with 'WARNING_CFLAGS' configure arg-var option.

```
$ ./configure WARNING_CFLAGS='-Wall -Wextra'
```

If configure option '–with-sparse-cgcc' is specified, cgcc is used as CC. cgcc is part of Sparse, Semantic Parser for C. It is used in development of Linux kernel for finding programming error. cgcc acts as a c compiler but more fussy. '-Wsparse-all' is used as default option passed to cgcc but you can change with 'CGCC_CFLAGS' configure arg-var option.

```
$ ./configure --with-sparse-cgcc [CGCC_CFLAGS='-Wsparse-all']
```

12.2 Finding performance bottleneck

See https://wiki.geany.org/howtos/profiling/gperftools and #383

12.3 Checking coverage

Before starting coverage measuring, you need to specify '–enable-coverage-gcov' configure option.

```
$ ./configure --enable-coverage-gcov
```

After doing `make clean`, you can build coverage measuring ready ctags by `make COVERAGE=1`. At this time **.gcno* files are generated by the compiler. **.gcno* files can be removed with `make clean`.

After building ctags, you can run run-gcov target. When running **.gcda* files. The target runs ctags with all input files under *Units/**/input.**; and call gcov. Human readable result is printed. The detail can be shown in **.gcov*. files. **.gcda* files and **.gcov* files can be removed with `make clean-gcov`.

12.4 Reviewing the result of Units test

Try misc/review. [TBW]

12.5 Running cppcheck

cppcheck is a tool for static C/C++ code analysis.

To run it do as following after install cppcheck:

```
$ make cppcheck
```

CHAPTER 13

Relationship between other projects

Table of contents

13.1 Geany

Geany maintains their own tagging engine derived from ctags. We are looking for the way to merge or share the source code each other.

Repo

> https://github.com/geany/geany/tree/master/tagmanager/ctags

Geany has created a library out of ctags

https://github.com/universal-ctags/ctags/issues/63

Their language parsers have many improvements to various parsers. Changes known by devs worth backporting:

- HTML reads <h1><h2><h3> tags
- Make has support for targets
- Various fixes for D parser (c.c), but currently the code diverges from ours to some extent.

They have these additional language parsers:

- Abaqus
- ActionScript
- AsciiDoc
- DocBook
- Ferite (c.c)
- GLSL (c.c)
- Haskell
- Haxe
- NSIS
- txt2tags
- Vala (c.c)

These changes have been merged:

- Fix regex callback match count - https://github.com/universal-ctags/ctags/pull/104
- SQL tags are stored with scopes instead of "tablename.field" - https://github.com/universal-ctags/ctags/pull/100
- Some fixes for D parser
- C++11's enum class/struct support

  orphan

13.2 Tracking other projects

This is working note for tracking activities other projects, especially activity at exuberant-ctags.

I(Masatake YAMATO) consider tracking activities as the first class fruits of this project.

13.2.1 exuberant-ctags

subversion

- status

  Revisions up to <r815> are merged except:

  NOTHING HERE NOW

  (Mon Sep 22 12:41:32 2014 by yamato)
- howto

```
<svn>
=> <git: local universal-ctags repo>
   => <git: local universal-ctags repo>
```

1. prepare your own universal-ctags repo: a local git repo cloned from github. You may know how to do it :)

   ```
   $ git clone https://github.com/universal-ctags/ctags.git
   ```

2. prepare exuberant-ctags SVN repo: a local git repo clone from exuberant-ctags svn tree.

   The original clone is already part of exuberant tree.

   To initialize your git repository with the required subversion information do

   ```
   $ git svn init https://svn.code.sf.net/p/ctags/code/trunk
   $ git update-ref refs/remotes/git-svn refs/remotes/origin/sourceforge
   ```

   and then

   ```
   $ git svn fetch
   $ git svn rebase
   ```

   to get the latest changes and reflect it to the local copy.

3. merge

   TODO

4. cherry-pick

   4.1. Make a branch at local universal-ctags repo and switch to it.

   4.2. Do cherry-pick like:

   ```
   $ git cherry-pick -s -x c81a8ce
   ```

   You can find commit id on the another terminal <git: local universal-ctags repo>:

   ```
   $ git log
   ```

   or

   ```
   $ git log --oneline
   ```

   If conflicts are occurred in cherry-picking, you can abort/reset cherry-picking with:

   ```
   $ git reset --hard
   ```

   <git: local universal-ctags repo> at the branch for picking.

bugs

<367> C++11 override makes a C++ member function declaration ignored

- fixed in:

  ```
  d4fcbdd
  #413
  #405
  ```

<366> –options=.ctags doesn't work under Windows

- fixed in:

```
15cedc6c94e95110cc319b5cdad7807caf3db1f4
```

<365> Selecting Python kinds is broken

- fixed in:

```
4a95e4a55f67230fc4eee91ffb31c18c422df6d3
```

- discussed at #324.

<364> Ruby method on self is missing the trailing ? in the generated tag name

- fixed in:

```
d9ba5df9f4d54ddaa511bd5440a1a3decaa2dc28
```

<363> Invalid C input file causes invalid read / heap overflow

- it is not reproduced.
- the test case is imported as parser-c.r/c-heapoverflow-sh-bug-363.d:

```
$ make units UNITS=c-heapoverflow-sh-bug-363 VG=1
```

<361> Invalid C input file causes invalid read / heap overflow

- it is not reproduced.

<360> Fails to parse annotation's fields with default value

- fixed in:

```
682a7f3b180c27c1196f8a1ae662d6e8ad142939
```

<358> Vim parser: Segmentation fault when reading empty vim file

- directly contributed by the original author of bug report and patch:

```
e0f854f0100e7a3cb8b959a23d6036e43f6b6c85
```

- it is fixed in sf, too:

```
5d774f6022a1af71fa5866994699aafce0253085
```

<356> [python] mistakes module level attribute for class level attribute in module level if

- fixed in:

```
ab91e6e1ae84b80870a1e8712fc7f3133e4b5542
```

<355> Error when parsing empty file (OCaml)

- fixed in:

```
02ec2066b5be6b129eba49685bd0b17fef4acfa
```

<341> Lua: "function f ()" whitespace

- fixed in:

```
8590bbef5fcf70f6747d509808c29bf84342cd0d
```

<341> Introducing ctags.conf.d

- merged the improved version:

```
216880c5287e0421d9c49898d983144db61c83aa
```

<271> regex callback is broken; <320> [PATCH] fix regex callback match count

- merged patch (with updated bug number):

```
a12b3a24b62d6535a968e076675f68bac9ad32ba
```

<177> Lua: "function" results in function tag (includes patch)

- fixed in:

```
5606f3f711afeac74587a249650a5f7b416f19be
```

13.2.2 patches

Tracking the tickets in patch tracker is quite fruitful. Patches are always there. So it is easy to evaluate the value:)

[(<]TICKET#[>)] TITLE

- STATUS
  - MORE STATUS

<TICKET#>

means the ticket is closed from the view of exuberant tree developers. We don't have to take time for this ticket.

(TICKET#)

means the ticket is still opened from the view of exuberant tree developers. We don't have to take time for this ticket.

<85> Add –encoding option to make utf-8 encoded tags file

- contributed by the original author:

```
b3f670c7c4a3c3570b8d2d82756735586aafc0cb
```

<84> C++11 new using semantics

- solved by another implementation:

```
c93e3bfa05b70d7fbc2539454c957eb2169e16b3
502355489b1ba748b1a235641bbd512ba6da315e
```

<83> New full non-regex PHP parser

- contributed by the original author

<82> Support for comments in .ctags files

- contributed by the original author:

```
cab4735e4f99ce23c52b78dc879bc06af66796fd
```

<81> ocaml parser segfaults on invalid files

- the bug is not reproduced

<80> Add support for falcon pl

- contributed by the original author

<74> protobuf parser

- Merged after getting approval from the original author

<67> Objective C language parser

- This is the implementation we have in universal-ctags tree.

<65> absoluteFilename uses strcpy on overlapping strings

- Fixed in universal-ctags tree, however the ticket is still open:

```
d2bdf505abb7569deae2b50305ea1edce6208557
```

<64> Fix strcpy() misuse

- Fixed in universal-ctags tree, however the ticket is still open:

```
d2bdf505abb7569deae2b50305ea1edce6208557
```

<55> TTCN-3 support

- contributed by the original author

<51> Ada support

- Ada support is now available in universal-ctags tree:

```
4b6b4a72f3d2d4ef969d7c650de1829d79f0ea7c
```

<38> Ada support

- Ada support is now available in universal-ctags tree:

```
4b6b4a72f3d2d4ef969d7c650de1829d79f0ea7c
```

<33> Add basic ObjC support

- This one is written in regexp.
- we have better objc parser.

(1) bibtex parser

- Reject because...
  - the owner of the ticket is anonymous.
  - the name of patch author is not written explicitly at the header of patch.
- Alternative

  https://gist.github.com/ptrv/4576213

13.2.3 devel mailing list (ctags-devel@sourceforge)

<[Ctags] Shebang with python3 instead of python> From: Martin Ueding <dev@ma...> - 2013-01-26 18:36:32

> Added python, python2 and python3 as extensions of python parser:
>
> ```
> bb81485205c67617f1b34f61341e60b9e8030502
> ```

<[Ctags-devel] Lack of fnmatch(3) in Windows> From: Frank Fesevur <ffes@us...> - 2013-08-24 20:25:47

> There is no fnmatch() in the Windows C library. Therefore a string comparison is done in fileNameMatched() in strlist.c and patterns are not recognized:

```
698bf2f3db692946d2358892d228a864014abc4b
```

<Re: [Ctags-devel] WindRes parser> From: Frank Fesevur <ffes@unns...> - 2013-08-30 21:23:50

A parser for Windows Resource files. http://en.wikipedia.org/wiki/Resource_%28Windows%29

```
95b4806ba6c006e4b7e72a006700e33c720ab9e7
```

([Ctags-devel] Skip repeat PATH_SEPARATORs in relativeFilename()) From: Seth Dickson <whefxlr@gm...> - 2013-12-24 04:51:01

Looks interesting.

13.2.4 Fedora

Some patches are maintained in ctags package of Fedora. Inventory of patches are http://pkgs.fedoraproject.org/cgit/ctags.git/tree/ctags.spec

<ctags-5.7-destdir.patch>

This patch was merged in universal-ctags git tree:

```
d4b5972427a46cbdcbfb050a944cf62b300676be
```

<ctags-5.7-segment-fault.patch>

This patch was merged in universal-ctags git tree:

```
8cc2b482f6c7257c5151893a6d02b8c79851fedd
```

(ctags-5.8-cssparse.patch)

Not in universal-ctags tree.

The reproducer is attached to the following page: https://bugzilla.redhat.com/show_bug.cgi?id=852101

However, universal-ctags doesn't reproduce with it.

I, Masatake YAMATO, read the patch. However, I don't understand the patch.

<ctags-5.8-css.patch>

This patch was merged in universal-ctags git tree:

```
80c1522a36df3ba52b8b7cd7f5c79d5c30437a63
```

<ctags-5.8-memmove.patch>

This patch was merged in exuberant ctags svn tree. As the result this patch is in universal-ctags tree:

```
d2bdf505abb7569deae2b50305ea1edce6208557
```

<ctags-5.8-ocaml-crash.patch>

This patch was merged in exuberant ctags svn tree. As the result this patch is in universal-ctags tree:

```
ddb29762b37d60a875252dcc401de0b7479527b1
```

<ctags-5.8-format-security.patch>

This patch was merged in exuberant ctags svn tree. As the result this patch is in universal-ctags tree:

```
2f7a78ce21e4156ec3e63c821827cf1d5680ace8
```

13.2.5 Debian

Some patches are maintained in ctags package of Debian. Inventory of patches are http://anonscm.debian.org/cgit/users/cjwatson/exuberant-ctags.git/tree/debian/patches/series

<python-disable-imports.patch>

> universal-ctags tags Y in *import X as Y* and Z in *from X import Y as Z* as definition tags. They are turned on by default. The others are tagged as reference tags. reference tags are recorded only when "r" extra tags are enabled. e.g. *–extras=+r*.

<vim-command-loop.patch>

> This patch was merged as an alternative for 7fb36a2f4690374526e9e7ef4f1e24800b6914ec
>
> Discussed on https://github.com/fishman/ctags/issues/74

```
e59325a576e38bc63b91abb05a5a22d2cef25ab7
```

13.2.6 Other interesting ctags repositories

There are several interesting repo's with ctags around. These are interesting to integrate in the future.

VIM-Japan

VIM-Japan have some interesting things, especially regarding encoding.

Anjuta

Anjuta is a Gnome IDE. They did not fork Exuberant ctags, but they did natively include it in Anjuta. They have made several additions to their version of it including fairly extensive Vala language support.

tagbar

Wiki

> https://github.com/majutsushi/tagbar/wiki

This is a gold mine of optlibs.

13.3 Software using ctags

pygments

> pygments can generate html files. It can utilize tags file as input for making hyperlinks. However, pygments just looks at names and lines in tags file. scopes and kinds are not used.
>
> As far as I(Masatake YAMATO) tried, using pygments from ctags is not so useful. There are critical gap between ctags and pygments. ctags focuses on identifiers. pygments focuses on keywords.

GNU global

> I(Masatake YAMATO) don't inspect this much but GNU global uses ctags internally.
>
> A person at GNU global project proposed an extension for the tags file format:
>
> http://sourceforge.net/p/ctags/mailman/message/30020186/

GNU source highlight

highlight can generate html files. It can utilize tags file as input for making hyperlinks. http://www.gnu.org/software/src-highlite/source-highlight.html#Generating-References

I(Masatake YAMATO) have not tried the feature yet.

OpenGrok

I(Masatake YAMATO) don't inspect this much but OpenGrok uses ctags internally.

Linux kernel

See linux/scripts/tags.sh of Linux kernel source tree. It utilizes c parser to the utmost limit.

CHAPTER 14

Proposal for extended Vi tags file format

Note: The contents of next section is a copy of FORMAT file in exuberant ctags source code in its subversion repository at sourceforge.net.

We have made some modifications:

- Exceptions introduced in Universal-ctags are explained with "EXCEPTION" marker.
- *Exceptions in Universal-ctags* subsction summarizes the exceptions.

Table of contents

Version 0.06 DRAFT

Date 1998 Feb 8

Author Bram Moolenaar <Bram at vim.org> and Darren Hiebert <dhiebert at users.sourceforge.net>

14.1 Introduction

The file format for the "tags" file, as used by Vi and many of its descendants, has limited capabilities.

This additional functionality is desired:

1. Static or local tags. The scope of these tags is the file where they are defined. The same tag can appear in several files, without really being a duplicate.
2. Duplicate tags. Allow the same tag to occur more then once. They can be located in a different file and/or have a different command.
3. Support for C++. A tag is not only specified by its name, but also by the context (the class name).
4. Future extension. When even more additional functionality is desired, it must be possible to add this later, without breaking programs that don't support it.

14.2 From proposal to standard

To make this proposal into a standard for tags files, it needs to be supported by most people working on versions of Vi, ctags, etc.. Currently this standard is supported by:

Darren Hiebert <dhiebert at users.sourceforge.net> Exuberant ctags

Bram Moolenaar <Bram at vim.org> Vim (Vi IMproved)

These have been or will be asked to support this standard:

Nvi Keith Bostic <bostic at bsdi.com>

Vile Tom E. Dickey <dickey at clark.net>

NEdit Mark Edel <edel at ltx.com>

CRiSP Paul Fox <fox at crisp.demon.co.uk>

Lemmy James Iuliano <jai at accessone.com>

Zeus Jussi Jumppanen <jussij at ca.com.au>

Elvis Steve Kirkendall <kirkenda at cs.pdx.edu>

FTE Marko Macek <Marko.Macek at snet.fri.uni-lj.si>

14.3 Backwards compatibility

A tags file that is generated in the new format should still be usable by Vi. This makes it possible to distribute tags files that are usable by all versions and descendants of Vi.

This restricts the format to what Vi can handle. The format is:

1. The tags file is a list of lines, each line in the format:

```
{tagname}<Tab>{tagfile}<Tab>{tagaddress}
```

{tagname} Any identifier, not containing white space..

EXCEPTION: Universal-ctags violates this item of the proposal; tagname may contain spaces. However, tabs are not allowed.

<Tab> Exactly one TAB character (although many versions of Vi can handle any amount of white space).

{tagfile} The name of the file where {tagname} is defined, relative to the current directory (or location of the tags file?).

{tagaddress} Any Ex command. When executed, it behaves like 'magic' was not set.

2. The tags file is sorted on {tagname}. This allows for a binary search in the file.
3. Duplicate tags are allowed, but which one is actually used is unpredictable (because of the binary search).

The best way to add extra text to the line for the new functionality, without breaking it for Vi, is to put a comment in the {tagaddress}. This gives the freedom to use any text, and should work in any traditional Vi implementation.

For example, when the old tags file contains:

```
main    main.c  /^main(argc, argv)$/
DEBUG   defines.c       89
```

The new lines can be:

```
main    main.c  /^main(argc, argv)$/;"any additional text
DEBUG   defines.c       89;"any additional text
```

Note that the ';' is required to put the cursor in the right line, and then the '"' is recognized as the start of a comment.

For Posix compliant Vi versions this will NOT work, since only a line number or a search command is recognized. I hope Posix can be adjusted. Nvi suffers from this.

14.4 Security

Vi allows the use of any Ex command in a tags file. This has the potential of a trojan horse security leak.

The proposal is to allow only Ex commands that position the cursor in a single file. Other commands, like editing another file, quitting the editor, changing a file or writing a file, are not allowed. It is therefore logical to call the command a tagaddress.

Specifically, these two Ex commands are allowed:

- A decimal line number:

```
89
```

- A search command. It is a regular expression pattern, as used by Vi, enclosed in // or ??:

```
/^int c;$/
?main()?
```

There are two combinations possible:

- Concatenation of the above, with ';' in between. The meaning is that the first line number or search command is used, the cursor is positioned in that line, and then the second search command is used (a line number would not be useful). This can be done multiple times. This is useful when the information in a single line is not unique, and the search needs to start in a specified line.

```
/struct xyz {/;/int count;/
389;/struct foo/;/char *s;/
```

- A trailing comment can be added, starting with ';"' (two characters: semi-colon and double-quote). This is used below.

```
89;" foo bar
```

This might be extended in the future. What is currently missing is a way to position the cursor in a certain column.

14.5 Goals

Now the usage of the comment text has to be defined. The following is aimed at:

1. Keep the text short, because:
   - The line length that Vi can handle is limited to 512 characters.
   - Tags files can contain thousands of tags. I have seen tags files of several Mbytes.
   - More text makes searching slower.
2. Keep the text readable, because:
   - It is often necessary to check the output of a new ctags program.
   - Be able to edit the file by hand.
   - Make it easier to write a program to produce or parse the file.
3. Don't use special characters, because:
   - It should be possible to treat a tags file like any normal text file.

14.6 Proposal

Use a comment after the {tagaddress} field. The format would be:

```
{tagname}<Tab>{tagfile}<Tab>{tagaddress}[;"<Tab>{tagfield}..]
```

{tagname} Any identifier, not containing white space..

EXCEPTION: Universal-ctags violates this item of the proposal; name may contain spaces. However, tabs are not allowed. Conversion, for some characters including <Tab> in the "value", explained in the last of this section is applied.

<Tab> Exactly one TAB character (although many versions of Vi can handle any amount of white space).

{tagfile} The name of the file where {tagname} is defined, relative to the current directory (or location of the tags file?).

{tagaddress} Any Ex command. When executed, it behaves like 'magic' was not set. It may be restricted to a line number or a search pattern (Posix).

Optionally:

;" semicolon + doublequote: Ends the tagaddress in way that looks like the start of a comment to Vi.

{tagfield} See below.

A tagfield has a name, a colon, and a value: "name:value".

- The name consist only out of alphabetical characters. Upper and lower case are allowed. Lower case is recommended. Case matters ("kind:" and "Kind: are different tagfields).
- The value may be empty. It cannot contain a <Tab>.
  - When a value contains a "\t", this stands for a <Tab>.
  - When a value contains a "\r", this stands for a <CR>.
  - When a value contains a "\n", this stands for a <NL>.
  - When a value contains a "\", this stands for a single '' character.

  Other use of the backslash character is reserved for future expansion. Warning: When a tagfield value holds an MS-DOS file name, the backslashes must be doubled!

  EXCEPTION: Universal-ctags introduces more conversion rules.

- When a value contains a "\a", this stands for a <BEL> (0x07).
- When a value contains a "\b", this stands for a <BS> (0x08).
- When a value contains a "\v", this stands for a <VT> (0x0b).
- When a value contains a "\f", this stands for a <FF> (0x0c).
- The characters in range 0x01 to 0x1F included, 0x7F, and leading space (0x20) and '!' (0x21) are converted to x prefixed hexadecimal number if the characters are not handled in the above "value" rules.

Proposed tagfield names:

FIELD-NAME	DESCRIPTION
arity	Number of arguments for a function tag.
class	Name of the class for which this tag is a member or method.
enum	Name of the enumeration in which this tag is an enumerator.
file	Static (local) tag, with a scope of the specified file. When the value is empty, {tagfile} is used.
function	Function in which this tag is defined. Useful for local variables (and functions). When functions nest (e.g., in Pascal), the function names are concatenated, separated with '/', so it looks like a path.
kind	Kind of tag. The value depends on the language. For C and C++ these kinds are recommended: **c** class name **d** define (from #define XXX) **e** enumerator **f** function or method name **F** file name **g** enumeration name **m** member (of structure or class data) **p** function prototype **s** structure name **t** typedef **u** union name **v** variable When this field is omitted, the kind of tag is undefined.
struct	Name of the struct in which this tag is a member.
union	Name of the union in which this tag is a member.

Note that these are mostly for C and C++. When tags programs are written for other languages, this list should be extended to include the used field names. This will help users to be independent of the tags program used.

Examples:

```
asdf    sub.cc  /^asdf()$/;"    new_field:some\svalue   file:
foo_t   sub.h   /^typedef foo_t$/;"     kind:t
func3   sub.p   /^func3()$/;"   function:/func1/func2   file:
getflag sub.c   /^getflag(arg)$/;"      kind:f  file:
inc     sub.cc  /^inc()$/;"     file: class:PipeBuf
```

The name of the "kind:" field can be omitted. This is to reduce the size of the tags file by about 15%. A program reading the tags file can recognize the "kind:" field by the missing ':'. Examples:

```
foo_t   sub.h   /^typedef foo_t$/;"     t
getflag sub.c   /^getflag(arg)$/;"      f       file:
```

Additional remarks:

- When a tagfield appears twice in a tag line, only the last one is used.

Note about line separators:

Vi traditionally runs on Unix systems, where the line separator is a single linefeed character <NL>. On MS-DOS and compatible systems <CR><NL> is the standard line separator. To increase portability, this line separator is also supported.

On the Macintosh a single <CR> is used for line separator. Supporting this on Unix systems causes problems, because most fgets() implementation don't see the <CR> as a line separator. Therefore the support for a <CR> as line separator is limited to the Macintosh.

Summary:

line separator	generated on	accepted on
<LF>	Unix	Unix, MS-DOS, Macintosh
<CR>	Macintosh	Macintosh
<CR><LF>	MS-DOS	Unix, MS-DOS, Macintosh

The characters <CR> and <LF> cannot be used inside a tag line. This is not mentioned elsewhere (because it's obvious).

Note about white space:

Vi allowed any white space to separate the tagname from the tagfile, and the filename from the tagaddress. This would need to be allowed for backwards compatibility. However, all known programs that generate tags use a single <Tab> to separate fields.

There is a problem for using file names with embedded white space in the tagfile field. To work around this, the same special characters could be used as in the new fields, for example "\s". But, unfortunately, in MS-DOS the backslash character is used to separate file names. The file name "c:\vim\sap" contains "\s", but this is not a <Space>. The number of backslashes could be doubled, but that will add a lot of characters, and make parsing the tags file slower and clumsy.

To avoid these problems, we will only allow a <Tab> to separate fields, and not support a file name or tagname that contains a <Tab> character. This means that we are not 100% Vi compatible. However, there is no known tags program that uses something else than a <Tab> to separate the fields. Only when a user typed the tags file himself, or made his own program to generate a tags file, we could run into problems. To solve this, the tags file should be filtered, to replace the arbitrary white space with a single <Tab>. This Vi command can be used:

```
:%s/^\([^ ^I]*\)[ ^I]*\([^ ^I]*\)[ ^I]*/\1^I\2^I/
```

(replace ^I with a real <Tab>).

TAG FILE INFORMATION:

Psuedo-tag lines can be used to encode information into the tag file regarding details about its content (e.g. have the tags been sorted?, are the optional tagfields present?), and regarding the program used to generate the tag file. This information can be used both to optimize use of the tag file (e.g. enable/disable binary searching) and provide general information (what version of the generator was used).

The names of the tags used in these lines may be suitably chosen to ensure that when sorted, they will always be located near the first lines of the tag file. The use of "!_TAG_" is recommended. Note that a rare tag like "!" can sort to before these lines. The program reading the tags file should be smart enough to skip over these tags.

The lines described below have been chosen to convey a select set of information.

Tag lines providing information about the content of the tag file:

```
!_TAG_FILE_FORMAT   {version-number}        /optional comment/
!_TAG_FILE_SORTED   {0|1}                   /0=unsorted, 1=sorted/
```

The {version-number} used in the tag file format line reserves the value of "1" for tag files complying with the original UNIX vi/ctags format, and reserves the value "2" for tag files complying with this proposal. This value may be used to determine if the extended features described in this proposal are present.

Tag lines providing information about the program used to generate the tag file, and provided solely for documentation purposes:

```
!_TAG_PROGRAM_AUTHOR        {author-name}   /{email-address}/
!_TAG_PROGRAM_NAME  {program-name}  /optional comment/
!_TAG_PROGRAM_URL   {URL}   /optional comment/
!_TAG_PROGRAM_VERSION       {version-id}    /optional comment/
```

14.7 Exceptions in Universal-ctags

Universal-ctags supports this proposal with some exceptions.

14.7.1 Exceptions

1. {tagname} in tags file generated by Universal-ctags may contain spaces and several escape sequences. Parsers for documents like Tex and reStructuredText, or liberal languages such as JavaScript need these exceptions. See {tagname} of Proposal section for more detail about the conversion.

14.7.2 Compatible output and weakness

Default behavior (`--output-format=u-ctags` option) has the exceptions. In other hand, with `--output-format=e-ctags` option ctags has no exception; Universal-ctags command may use the same file format as Exuberant-ctags. However, `--output-format=e-ctags` throws away a tag entry which name includes a space or a tab character. `TAG_OUTPUT_MODE` pseudo tag tells which format is used when ctags generating tags file.

CHAPTER 15

Who we are

Please, add your name, background and interests here If you are interested in contributing to universal-ctags steadily. So we can dispatch a task and/or an issue to the right person!

(Keep the list in alphabetical order.)

Frank Fesevur <ffes@users.sourceforge.net>

> My current use of ctags is for a Notepad++ plug-in I'm writing. The plug-in is not yet released because of problems with the Windows version of ctags. Those problems are fixed by now. I am a Windows developer, but also an occasional Ubuntu and Raspbian user at home. I wrote the windres parser.

Colomban Wendling <colomban@geany.org>

> I am a developer of Geany, a lightweight IDE/editor that uses CTags parsers to provide various code insights for a large variety of languages. I don't use CTags directly but through a (currently) internal library. Hence, my fields of interest are the quality of the parsers (good and comprehensive results) and their code (speed, proof against any inputs, absence of memory leaks, regression tests), and a CTags library applications could use more readily. I am mostly a C developer, but as the maintainer of the CTags parsers in Geany I work on all parsers.

Masatake YAMATO <yamato@redhat.com>

> I'm using ctags in batch jobs running on my source code base where most of all source code in Fedora are deployed. I'm an Emacs user, so generally I don't use ctags interactively except when hacking universal-ctags. Therefore my primary goal is to improve the robustness of parsers: I introduced Units test facility and badinput command for achieving the goal. The secondary goal is to support more languages and formats: I introduced optlib. I'm working on Fedora. I don't have access to the other platforms.

Qingming He <906459647@qq.com>

> I'm mainly a Fortran developer and I use ctags combined with Emacs to handle my projects. My goal is to improve the Fortran parser to make it support Fortran standards from 77 to 2008 and maybe 2015 to be released in the near future. I'm also interested in improving the lisp parsers (elisp and scheme).

Vitor Antunes <vitor.hda@gmail.com>

> I've been working with Verilog for most of the last 10 years and am an avid Vim user. My goal is to improve the Verilog parser such that Vim can get the most out of it in plugins like Tagbar and to support the Omni completion plugin I am writing.

Cameron Eagans <me@cweagans.net>

> I've been a PHP developer for almost 10 years, and have been using Vim almost as long. My goal is to help guide the direction of the PHP parser, as well as maintain the ctags website and help guide new contributors to tasks that they may be able to help with. With time, I may end up contributing directly to ctags development, but my C skills are not so great at the moment.

Szymon Tomasz Stefanek <s.stefanek@gmail.com>

> I'm a multilanguage developer and I use ctags with my own text editor which has some IDE capabilities. I'm the maintainer of the new C/C++ parser.