

---

# **CPIP Documentation**

***Release 0.9.7***

**Paul Ross**

**Oct 04, 2017**



---

## Contents

---

<b>1</b>	<b>CPIP</b>	<b>3</b>
1.1	Features . . . . .	3
1.2	Installation . . . . .	3
<b>2</b>	<b>Visualising Preprocessing</b>	<b>5</b>
2.1	Visualising the Source Code . . . . .	8
2.2	Visualising the <code>#include</code> Dependencies . . . . .	10
2.3	Visualising Conditional Compilation . . . . .	14
2.4	Understanding Macros . . . . .	15
2.5	Status . . . . .	17
2.6	Licence . . . . .	17
2.7	Credits . . . . .	17
<b>3</b>	<b>Installation</b>	<b>19</b>
3.1	Stable release . . . . .	19
3.2	From sources . . . . .	19
3.3	Developing with CPIP . . . . .	20
3.4	Testing the Demo Code . . . . .	20
<b>4</b>	<b>CPIP Introduction</b>	<b>21</b>
4.1	Pre-processing C and C++ . . . . .	21
4.2	CPIP and Pre-processing . . . . .	22
4.3	CPIP Core Architecture . . . . .	23
<b>5</b>	<b>CPIPMain.py Examples</b>	<b>25</b>
5.1	Screenshots . . . . .	25
5.2	Some Real Examples . . . . .	41
<b>6</b>	<b>Command Line Tools</b>	<b>43</b>
6.1	CPIPMain . . . . .	43
<b>7</b>	<b>CPIP Tutorials</b>	<b>49</b>
7.1	PpLexer Tutorial . . . . .	49
7.2	FileIncludeGraph Tutorial . . . . .	60
<b>8</b>	<b>CPIP Reference</b>	<b>65</b>
8.1	CPIPMain . . . . .	65

8.2	CppCondGraphToHtml . . . . .	67
8.3	FileStatus . . . . .	67
8.4	IncGraphSVG . . . . .	69
8.5	IncGraphSVGBase . . . . .	70
8.6	IncGraphXML . . . . .	70
8.7	IncList . . . . .	70
8.8	ItuToHtml . . . . .	70
8.9	MacroHistoryHtml . . . . .	70
8.10	TokenCss . . . . .	71
8.11	Tu2Html . . . . .	71
8.12	TuIndexer . . . . .	72
8.13	cpip.core . . . . .	72
8.14	cpip.util . . . . .	108
8.15	cpip.plot . . . . .	118
<b>9</b>	<b>Usage</b>	<b>129</b>
<b>10</b>	<b>Contributing</b>	<b>131</b>
10.1	Types of Contributions . . . . .	131
10.2	Get Started! . . . . .	132
10.3	Pull Request Guidelines . . . . .	133
10.4	Tips . . . . .	133
<b>11</b>	<b>Credits</b>	<b>135</b>
11.1	Development Lead . . . . .	135
11.2	Contributors . . . . .	135
<b>12</b>	<b>History</b>	<b>137</b>
12.1	0.9.7 Beta Release (2017-10-04) . . . . .	137
12.2	0.9.5 Beta Release (2017-10-03) . . . . .	137
12.3	0.9.1 (2014-09-03) . . . . .	137
12.4	Alpha Plus Release (2014-09-04) . . . . .	137
12.5	Alpha Release (2012-03-25) . . . . .	137
12.6	Alpha Release (2011-07-14) . . . . .	138
<b>13</b>	<b>Indices and tables</b>	<b>139</b>
	<b>Python Module Index</b>	<b>141</b>

Contents:



CPIP is a C/C++ Preprocessor implemented in Python. It faithfully records all aspects of preprocessing and can produce visualisations that make debugging preprocessing far easier.

## Features

- Conformant C/C++ preprocessor.
- Gives programatic access to every preprocessing token and the state of the preprocessor at any point during preprocessing.
- Top level tools such as `CPIPMain.py` can generate preprocessor visualisations from the command line.
- Requires only Python 2.7 or 3.3+
- Fully documented: <https://cpip.readthedocs.io>.
- Free software: GNU General Public License v2

## Installation

You can either clone the public repository:

```
$ git clone git://github.com/paulross/cpip
```

Or download the tarball:

```
$ curl -OL https://github.com/paulross/cpip/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

To run the tests:

```
$ python setup.py test
```



---

### Visualising Preprocessing

---

The top level script `CPIPMain.py` acts like a preprocessor that generates HTML and SVG output for a source code file or directory. This output makes it easy to understand what the preprocessor is doing to your source.

Here is some of that output when preprocessing a single Linux kernel file `cpu.c` ([complete output](#)). The `index.html` page shows how `CPIPMain.py` was invoked<sup>1</sup>, this has a link to preprocessing pages for that file:

---

<sup>1</sup> This was invoked by:

**CPIP Processing in output location: ../output/linux/cpu\_43\_Python3**

Files Processed as Translation Units:

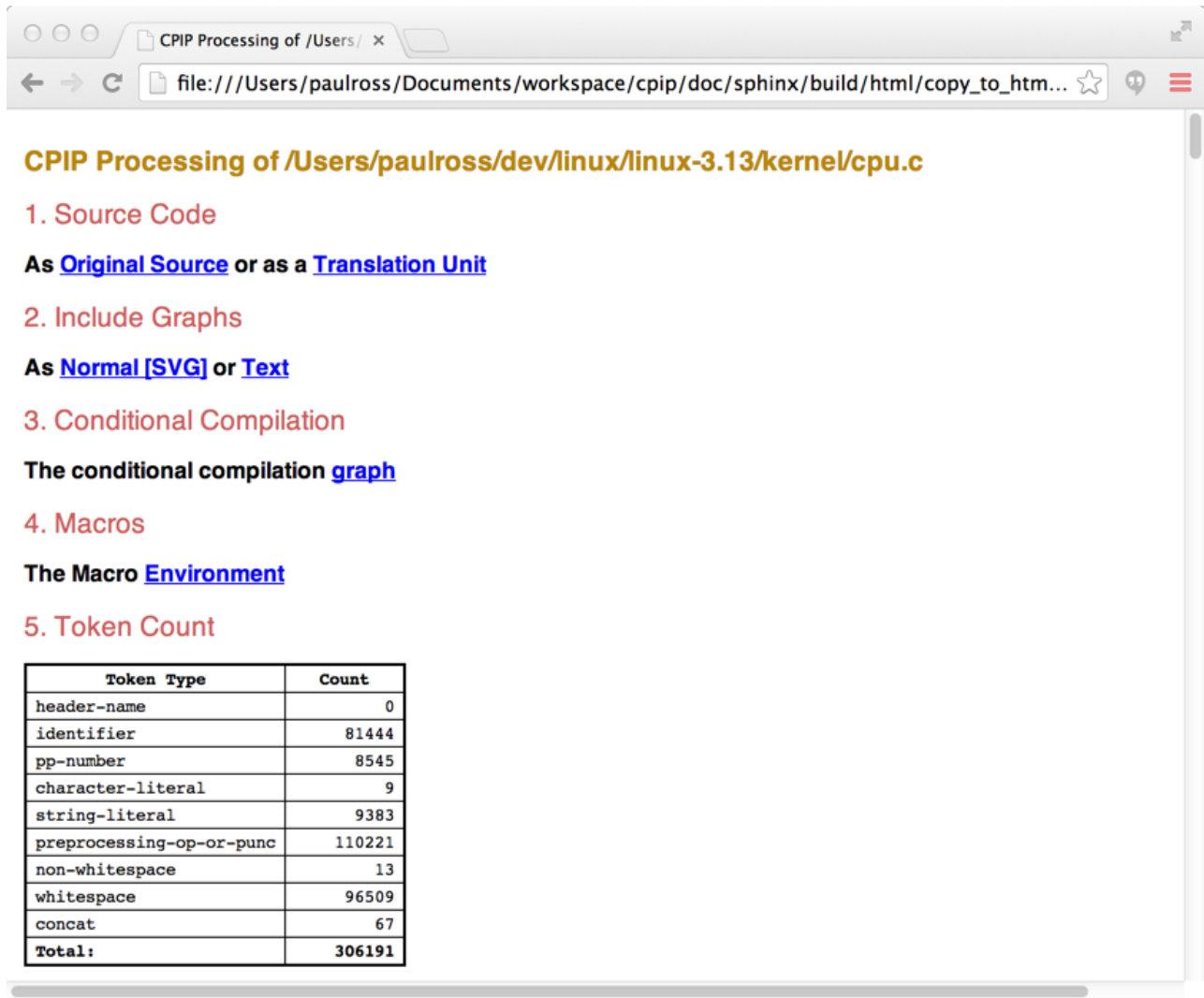
- [/Users/paulross/dev/linux/linux-3.13/kernel/cpu.c](#)

**CPIP Command line:**

```
CPIPMain.py -kp -l20 -o ../../output/linux/cpu_43_Python3 -S __STDC__=1 -D __KERNEL__ -D __EXPORTED_HEADERS__ -D BITS_PER_LONG=64 -D CONFIG_H
```

Option	Value	Description
--heap	False	Profile memory usage.
--version	None	show program's version number and exit
-D/--define	__KERNEL__, __EXPORTED_HEADERS__, BITS_PER_LONG=64, CONFIG_HZ=100, __x86_64__, __GNUC__=4, __has_feature(x)=0, __has_extension=__has_feature, __has_attribute=__has_feature, __has_include=__has_feature	Add macro definitions of the form name<=definition>. These are introduced into the environment before any pre-include.
-I/--usr		Add user include search path.
-J/--sys	/usr/include/, /usr/include/c++/4.2.1/, /usr/include/c++/4.2.1/tr1/, /Users/paulross/dev/linux/linux-3.13/include/, /Users/paulross/dev/linux/linux-3.13/include/uapi/, /Users/paulross/dev/linux/linux-3.13/arch/x86/include/uapi/, /Users/paulross/dev/linux/linux-3.13/arch/x86/include/, /Users/paulross/dev/linux/linux-3.13/arch/x86/include/generated/	Add system include search path.
-P/--pre	/Users/paulross/dev/linux/linux-3.13/include/linux/kconfig.h	Add pre-include file path.

This page has a single link that takes you to the landing page for the file `cpu.c`, at the top this links to other pages that visualise source code, `#include` dependencies, conditional compilation and macros:

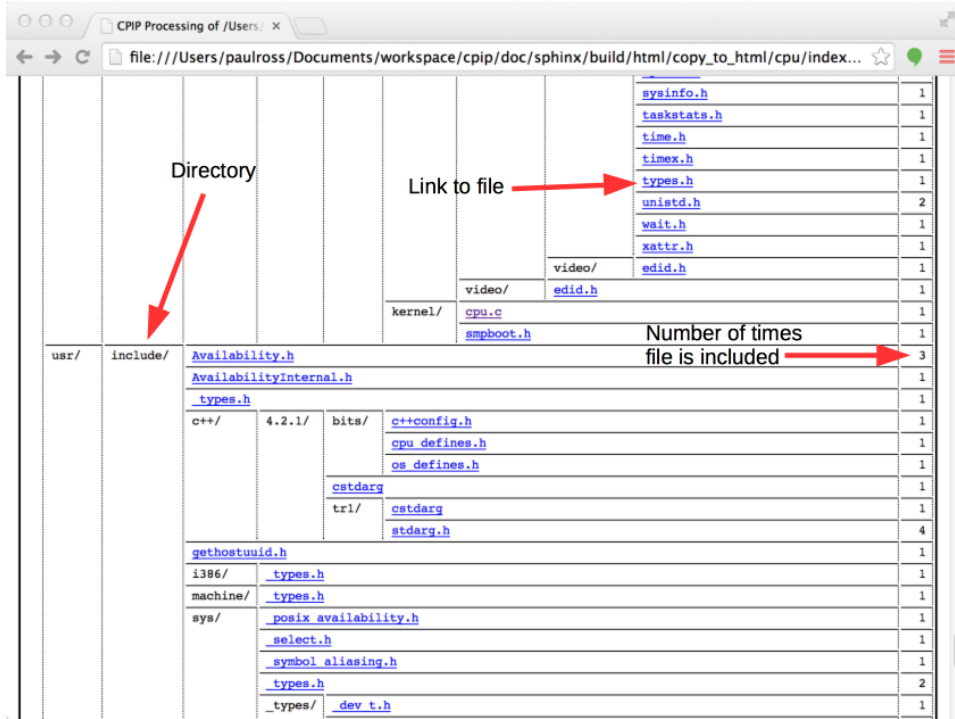


**CPIP Processing of /Users/paulross/dev/linux/linux-3.13/kernel/cpu.c**

1. Source Code  
As [Original Source](#) or as a [Translation Unit](#)
2. Include Graphs  
As [Normal \[SVG\]](#) or [Text](#)
3. Conditional Compilation  
The conditional compilation [graph](#)
4. Macros  
The Macro [Environment](#)
5. Token Count

Token Type	Count
header-name	0
identifier	81444
pp-number	8545
character-literal	9
string-literal	9383
preprocessing-op-or-punc	110221
non-whitespace	13
whitespace	96509
concat	67
<b>Total:</b>	<b>306191</b>

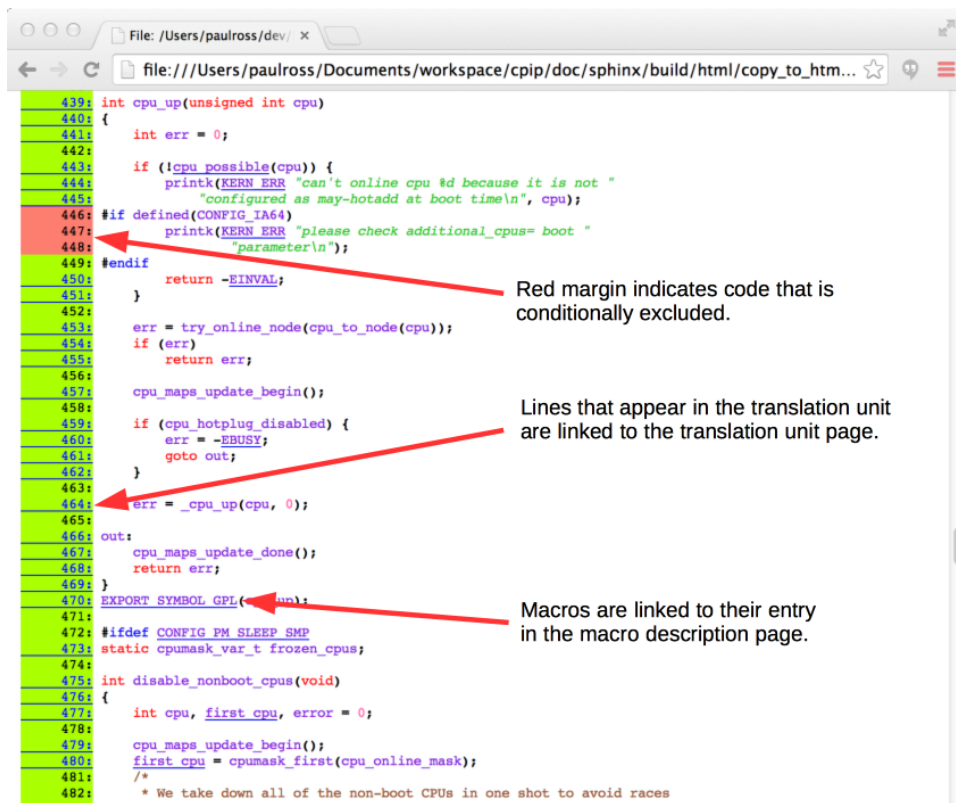
Lower down this page is a table of files that were involved in preprocessing:



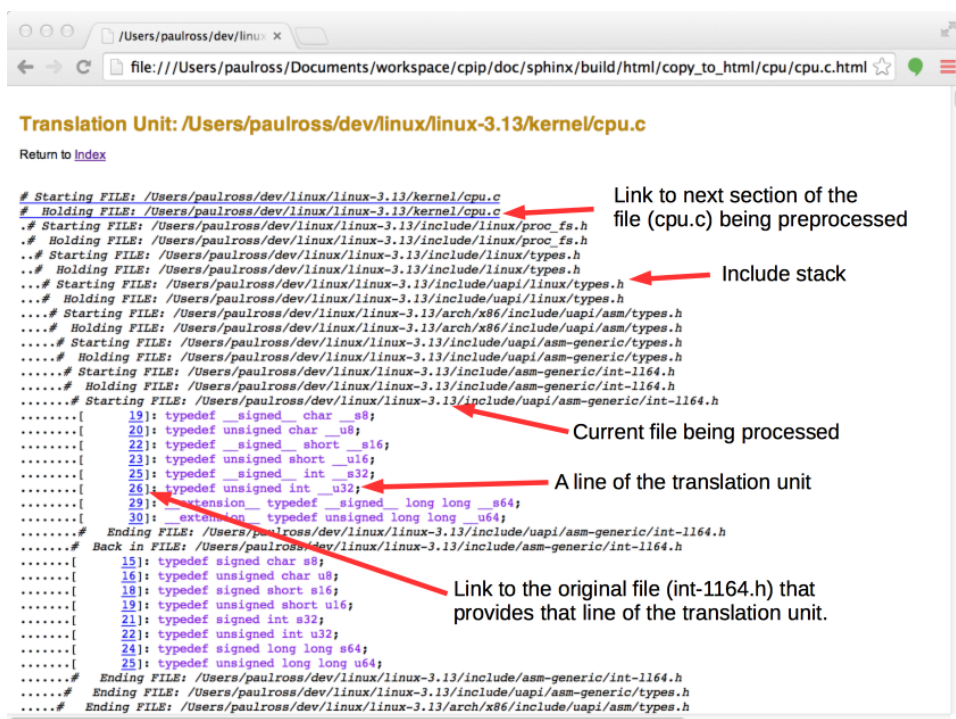
								<a href="#">sysinfo.h</a>	1
								<a href="#">taskstats.h</a>	1
								<a href="#">time.h</a>	1
								<a href="#">timex.h</a>	1
								<a href="#">types.h</a>	1
								<a href="#">unistd.h</a>	2
								<a href="#">wait.h</a>	1
								<a href="#">xattr.h</a>	1
						video/		<a href="#">edid.h</a>	1
						kernel/		<a href="#">cpu.c</a>	1
								<a href="#">smpboot.h</a>	1
usr/	include/							<a href="#">Availability.h</a>	3
								<a href="#">AvailabilityInternal.h</a>	1
								<a href="#">types.h</a>	1
		c++/	4.2.1/	bits/				<a href="#">c++config.h</a>	1
								<a href="#">cpu defines.h</a>	1
								<a href="#">os defines.h</a>	1
								<a href="#">cstdarg</a>	1
				trl/				<a href="#">cstdarg</a>	1
								<a href="#">stdarg.h</a>	4
								<a href="#">gethostuuid.h</a>	1
		i386/						<a href="#">types.h</a>	1
		machine/						<a href="#">types.h</a>	1
		sys/						<a href="#">posix availability.h</a>	1
								<a href="#">select.h</a>	1
								<a href="#">symbol aliasing.h</a>	1
								<a href="#">types.h</a>	2
								<a href="#">types/ _dev t.h</a>	1

## Visualising the Source Code

From the `cpu.c` landing page the link “Original Source” takes you to a syntax highlighted page of the original source of `cpu.c`.

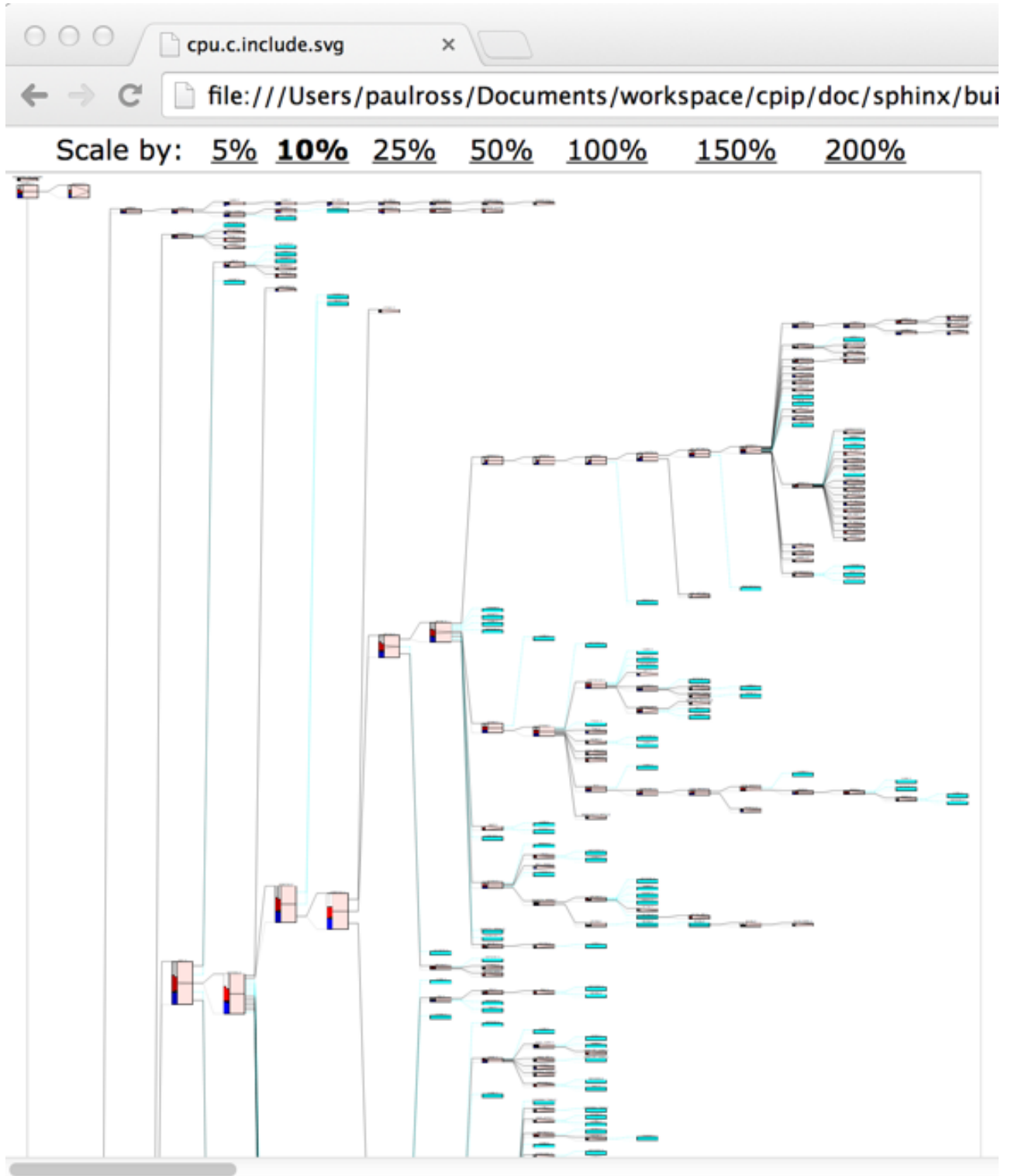


The `cpu.c` landing page link “Translation Unit” takes you to a page that shows the complete translation unit of `cpu.c` (i.e. incorporating all the `#include` files). This page is annotated so that you can understand what part of the translation unit comes from which file.

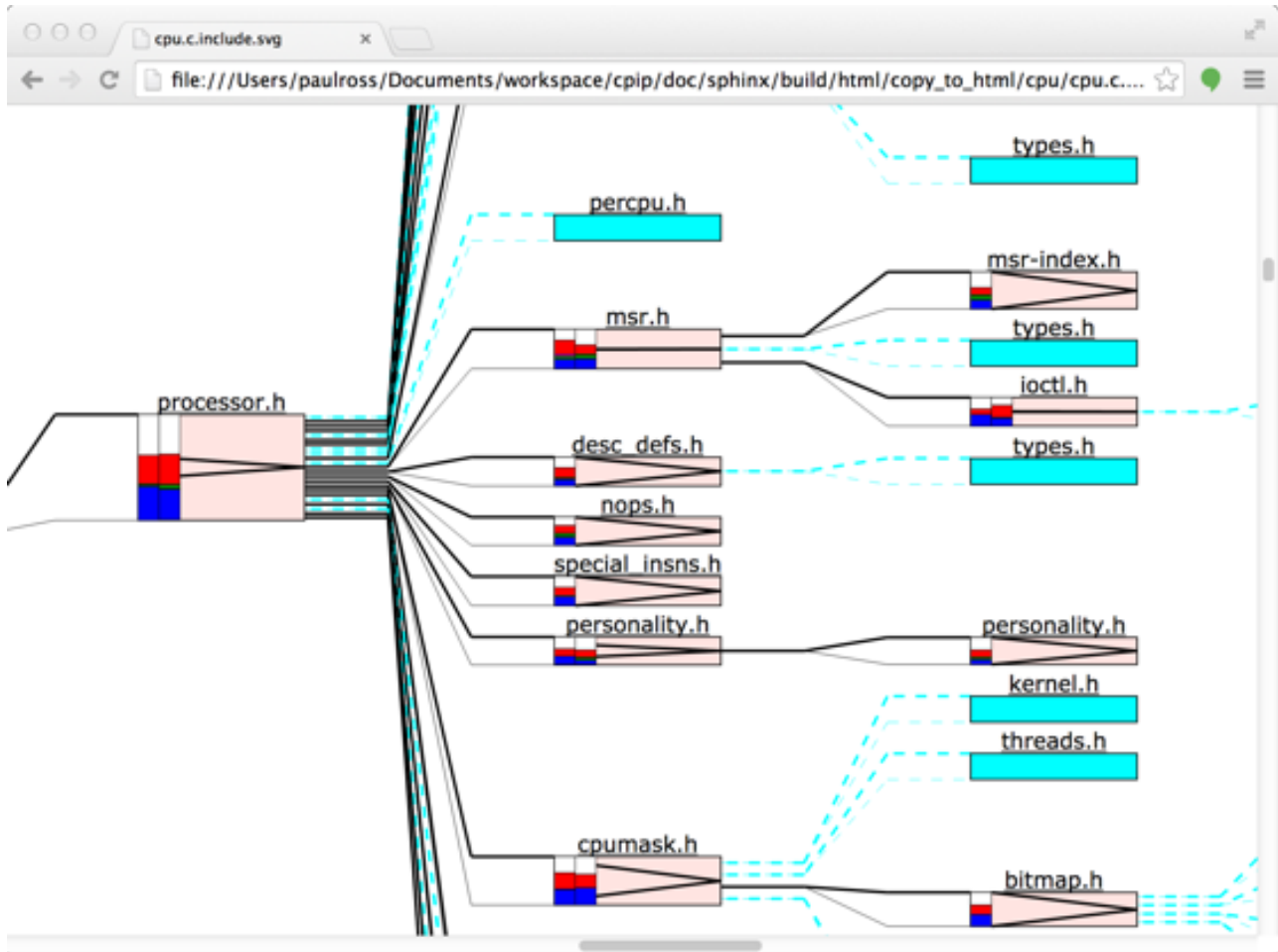


## Visualising the `#include` Dependencies

The `cpu.c` landing page link “Normal [SVG]” takes you to a page that shows the dependencies created by `#include` directives. This is a very rich page that represents a tree with the root at center left. `#include`'s are in order from top to bottom. Each block represents a file, the size is proportional to the number of preprocessing tokens.



Zooming in with the controls at the top gives more detail. If the box is coloured cyan it is because the file does not add any content to the translation unit, usually because of conditional compilation:

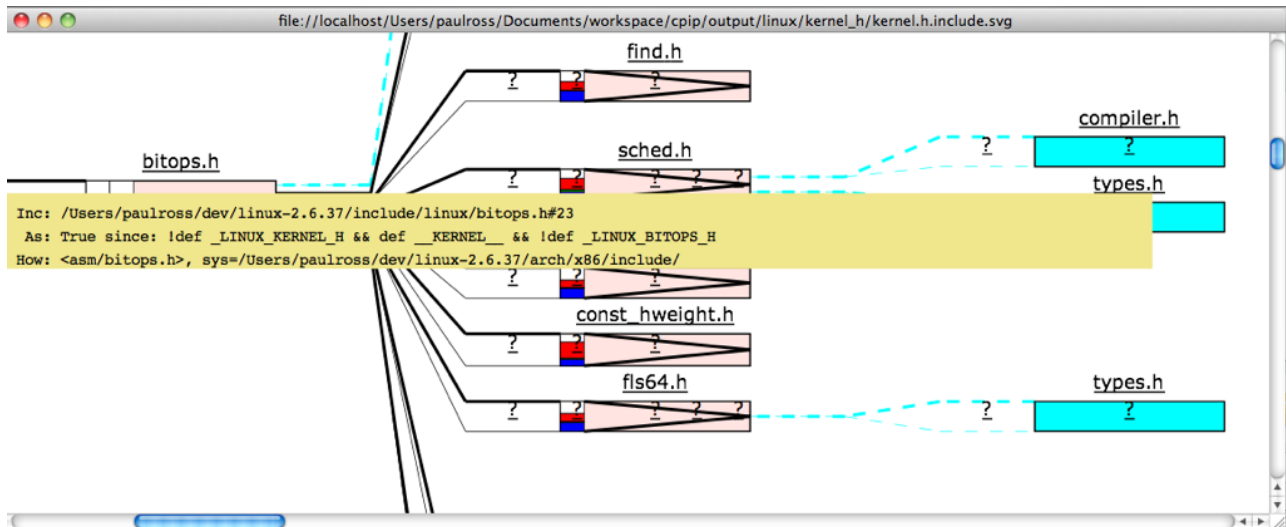


The page is dynamic and hovering over various areas provides more information:

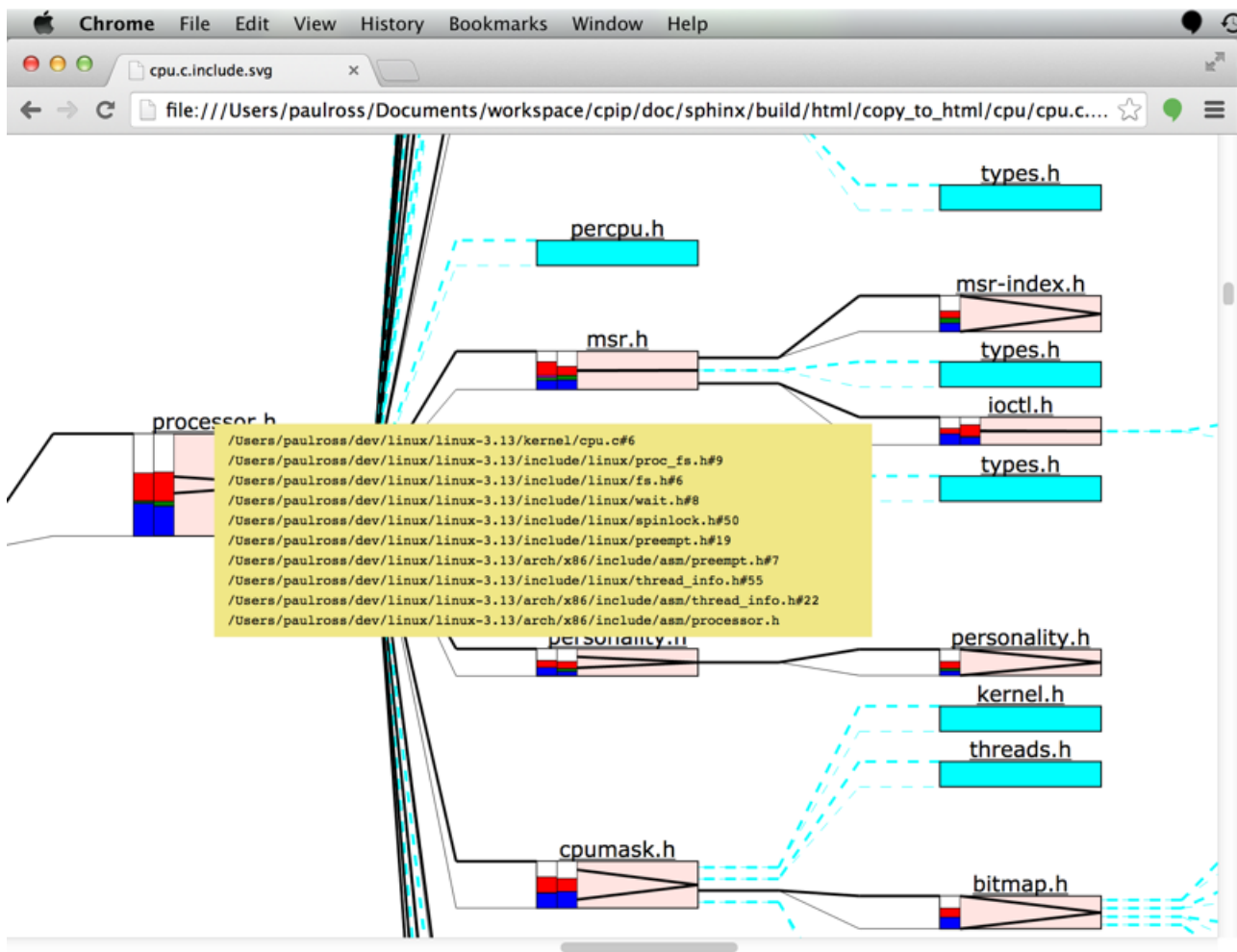
## How and Why the File was Included

Hovering just to the left of the file box produces a popup that explains how the file inclusion process worked for this file, it has the following fields:

- Inc: The filename and line number of the `#include` directive.
- As: The conditional compilation state at the point of the `#include` directive.
- How: The text of the `#include` directive followed by the directory that this file was found in, this directory is prefixed by `sys=` for a system include and `usr=` for a user include.

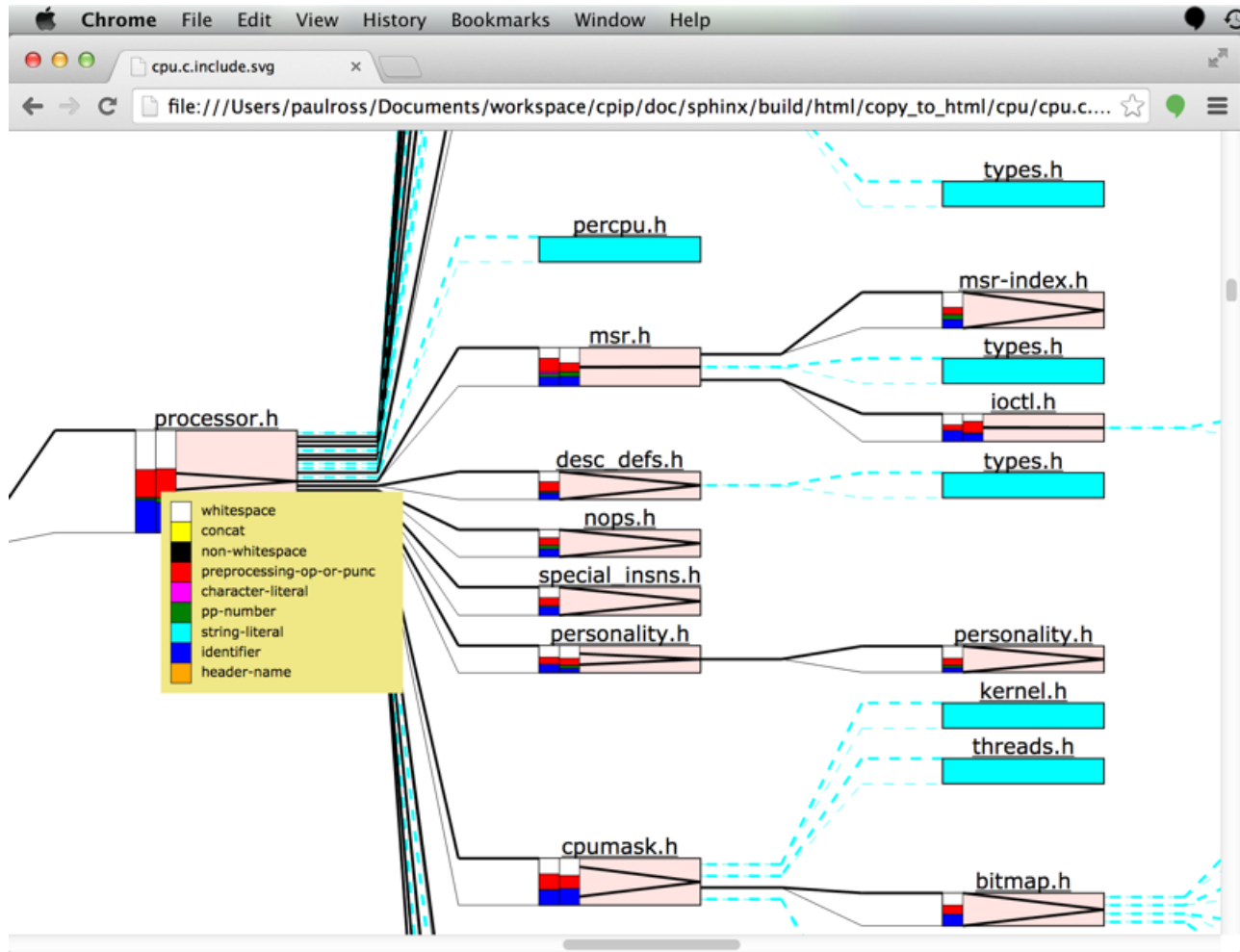


Hovering over the filename above the file box shows the file stack (children are below parents).

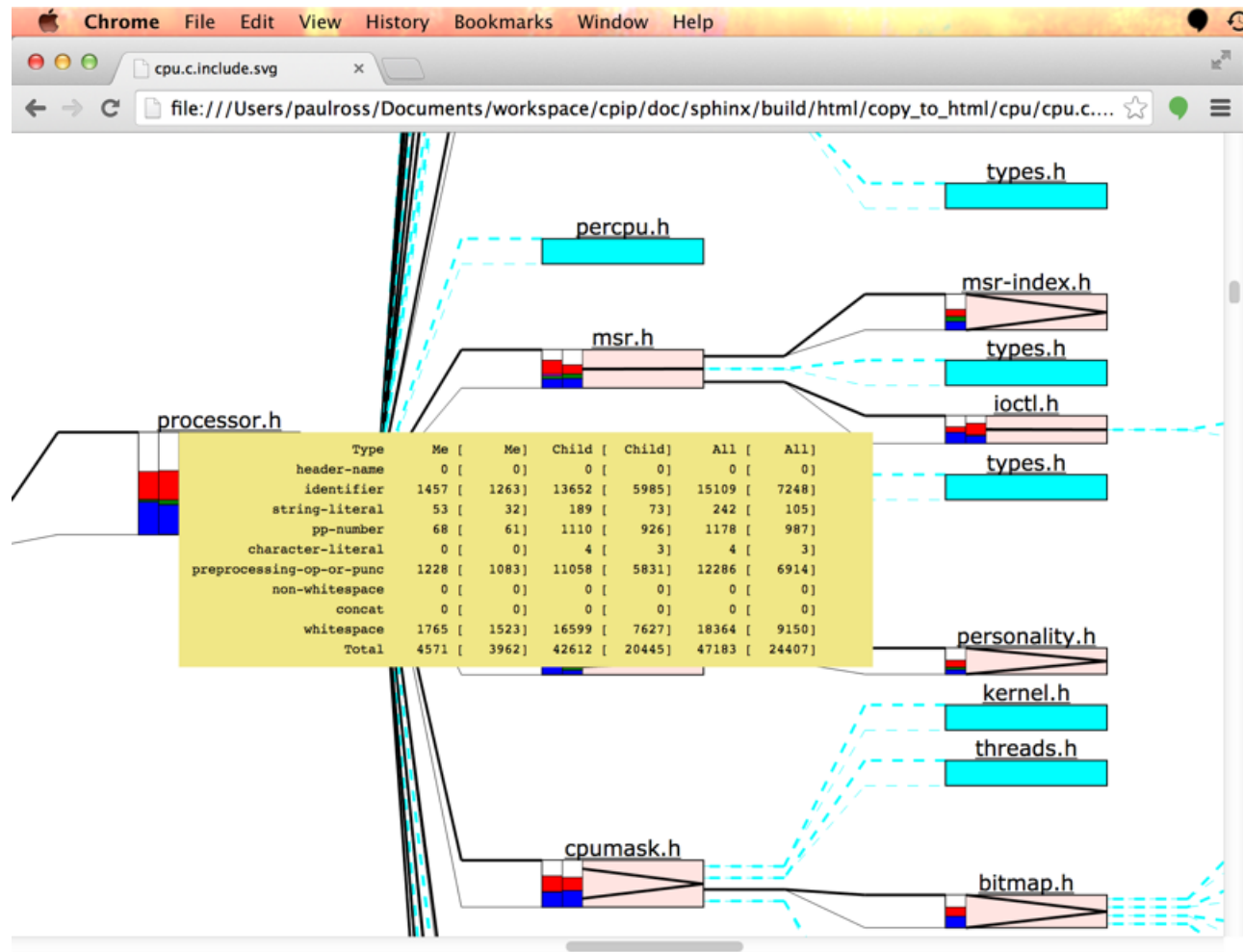


This plot can also tell you what types of preprocessor tokens were processed for each file. The coloured bars on the left of the file box indicate the proportion of preprocessing token types, the left is the file on its own, the right is the file and its child files. To understand the legend hover over those bars:





To see the actual count of preprocessing tokens hover over the file box:



## Visualising Conditional Compilation

The preprocessor is also responsible for handling conditional compilation which becomes very complicated for large projects. `CPIPMain.py` produces a succinct representation showing only the conditional directives. The links in each comment takes you to the syntax highlighted page for that file.

Preprocessing Conditional Compilation Graph: /Users/paulross/dev/linux/linux-3.13/kernel/cpu.c

[Return to Index](#)

```

#ifdef __LINUX_KCONFIG_H /* kconfig.h */
#endif /* kconfig.h */
#ifdef __LINUX_PROC_FS_H /* proc fs.h */
#endif __LINUX_TYPES_H /* types.h */
#ifdef __UAPI_LINUX_TYPES_H /* types.h */
#ifdef __ASM_X86_TYPES_H /* types.h */
#ifdef __ASM_GENERIC_TYPES_H /* types.h */
#ifdef __ASM_GENERIC_INT_LL64_H /* int-ll64.h */
#ifdef __UAPI_ASM_GENERIC_INT_LL64_H /* int-ll64.h */
#ifdef __ASM_X86_BITSPERLONG_H /* bitsperlong.h */
#ifdef __x86_64__ /* bitsperlong.h */
#else /* bitsperlong.h */
#endif /* bitsperlong.h */
#ifdef __ASM_GENERIC_BITS_PER_LONG /* bitsperlong.h */
#ifdef __UAPI_ASM_GENERIC_BITS_PER_LONG /* bitsperlong.h */
#ifdef __BITS_PER_LONG /* bitsperlong.h */
#endif /* bitsperlong.h */
#endif /* bitsperlong.h */
#ifdef CONFIG_64BIT /* bitsperlong.h */
#else /* bitsperlong.h */
#endif /* bitsperlong.h */
#if 0 && BITS_PER_LONG != __BITS_PER_LONG /* bitsperlong.h */
#endif /* bitsperlong.h */
#ifdef BITS_PER_LONG_LONG /* bitsperlong.h */
#endif /* bitsperlong.h */
#endif /* bitsperlong.h */
#endif /* bitsperlong.h */
#ifdef __ASSEMBLY__ /* int-ll64.h */
#ifdef __GNUC__ /* int-ll64.h */
#else /* int-ll64.h */
#endif /* int-ll64.h */
#endif /* int-ll64.h */
#ifdef __ASSEMBLY__ /* int-ll64.h */
#else /* int-ll64.h */
#endif /* int-ll64.h */
#endif /* int-ll64.h */

```

## Understanding Macros

CPIP tracks every macro definition and usage and CPIMain.py produces a page that describes all the macros encountered:

Macro Environment for: /Users/paulross/dev/linux/linux-3.13/kernel/cpu.c

Return to [Index](#)

Referenced Macros:

- Inactive [22]: [ADDR](#) [IS](#) [SUBSYS](#) [ENABLED\(option\)](#) [IS](#) [SUBSYS](#) [ENABLED\(option\)](#) [IS](#) [SUBSYS](#) [ENABLED\(option\)](#) [READ](#) [LOCK](#) [ATOMIC\(n\)](#) [READ](#) [LOCK](#) [SIZE\(insn\)](#) [SUBSYS\(x\)](#) [SUBSYS\(x\)](#) [WRITE](#) [LOCK](#) [ADD\(n\)](#) [WRITE](#) [LOCK](#) [CMP](#) [WRITE](#) [LOCK](#) [SUB\(n\)](#) [SIG](#) [SET](#) [BINOP\(name,op\)](#) [SIG](#) [SET](#) [OP\(name,op\)](#) [copy](#) [to](#) [user](#) [overflow](#) [deprecated](#) [must](#) [check](#) [sig](#) [and\(x,y\)](#) [sig](#) [andn\(x,y\)](#) [sig](#) [not\(x\)](#) [sig](#) [or\(x,y\)](#) [copy](#) [user](#) [diag](#) [u32](#)
- Active [1642]: [ACCESS](#) [ONCE\(x\)](#) [ACPI](#) [PDC](#) [C](#) [C1](#) [FFH](#) [ACPI](#) [PDC](#) [C](#) [C1](#) [HALT](#) [ACPI](#) [PDC](#) [C](#) [C2C3](#) [FFH](#) [ACPI](#) [PDC](#) [C](#) [CAPABILITY](#) [SMP](#) [ACPI](#) [PDC](#) [EST](#) [CAPABILITY](#) [WSMP](#) [ACPI](#) [PDC](#) [P](#) [FFH](#) [ACPI](#) [PDC](#) [SMP](#) [C1PT](#) [ACPI](#) [PDC](#) [SMP](#) [C2C3](#) [ACPI](#) [PDC](#) [SMP](#) [P](#) [HWCOORD](#) [ACPI](#) [PDC](#) [SMP](#) [P](#) [SWCOORD](#) [ACPI](#) [PDC](#) [T](#) [FFH](#) [ALLOC](#) [SPLIT](#) [PTLOCKS](#) [ALTERNATIVE\(oldinstr,newinstr,feature\)](#) [ALTERNATIVE](#) [2\(oldinstr,newinstr,feature1,newinstr2,feature2\)](#) [ALTINSTR](#) [ENTRY\(feature,number\)](#) [ALTINSTR](#) [REPLACEMENT\(newinstr,feature,number\)](#) [APIC](#) [ALL](#) [CPUS](#) [APIC](#) [BASE](#) [APIC](#) [BASE](#) [MSR](#) [APIC](#) [DEBUG](#) [APIC](#) [DFR](#) [APIC](#) [EOI](#) [APIC](#) [EOI](#) [ACK](#) [APIC](#) [ICR](#) [APIC](#) [ID](#) [APIC](#) [LDR](#) [APIC](#) [LVR](#) [APIC](#) [XAPIC\(x\)](#) [ARCH](#) [HAS](#) [IOREMAP](#) [WC](#) [ASM](#) [CLAC](#) [ASM](#) [NOP3](#) [ASM](#) [OUTPUT2\(a,...\)](#) [ASM](#) [STAC](#) [ASM](#) [X86](#) [CMPXCHG](#) [H](#) [ATOMIC](#) [INIT\(i\)](#) [AT](#) [VECTOR](#) [SIZE](#) [AT](#) [VECTOR](#) [SIZE](#) [ARCH](#) [AT](#) [VECTOR](#) [SIZE](#) [BASE](#) [BAD](#) [APICID](#) [BASE](#) [PREFETCH](#) [BITMAP](#) [LAST](#) [WORD](#) [MASK\(nbits\)](#) [BITOP](#) [ADDR\(x\)](#) [BITOP](#) [LE](#) [SWIZZLE](#) [BITS](#) [PER](#) [BYTE](#) [BITS](#) [PER](#) [LONG](#) [BITS](#) [TO](#) [LONGS\(nr\)](#) [BUG\(\)](#) [BUGFLAG](#) [WARNING](#) [BUG](#) [ON\(condition\)](#) [BUILDIO\(bwl,bw,type\)](#) [BUILD](#) [BUG](#) [ON\(condition\)](#) [BUILD](#) [BUG](#) [ON](#) [INVALID\(e\)](#) [CAP](#) [BOP](#) [ALL\(c,a,b,OP\)](#) [CAP](#) [CHOWN](#) [CAP](#) [DAC](#) [OVERRIDE](#) [CAP](#) [DAC](#) [READ](#) [SEARCH](#) [CAP](#) [FOR](#) [EACH](#) [U32\(capi\)](#) [CAP](#) [FOWNER](#) [CAP](#) [FSETID](#) [CAP](#) [FS](#) [MASK](#) [B0](#) [CAP](#) [FS](#) [MASK](#) [B1](#) [CAP](#) [FS](#) [SET](#) [CAP](#) [LINUX](#) [IMMUTABLE](#) [CAP](#) [MAC](#) [OVERRIDE](#) [CAP](#) [MKNOD](#) [CAP](#) [NFSD](#) [SET](#) [CAP](#) [SYS](#) [RESOURCE](#) [CAP](#) [TO](#) [INDEX\(x\)](#) [CAP](#) [TO](#) [MASK\(x\)](#) [CAP](#) [UOP](#) [ALL\(c,a,OP\)](#) [CLEARPAGEFLAG\(uname,lname\)](#) [CONFIG](#) [64BIT](#) [CONFIG](#) [ACPI](#) [CONFIG](#) [ACPI](#) [NUMA](#) [CONFIG](#) [ACPI](#) [SLEEP](#) [CONFIG](#) [AIO](#) [CONFIG](#) [AMD](#) [IOMMU](#) [CONFIG](#) [ARCH](#) [CLOCKSOURCE](#) [DATA](#) [CONFIG](#) [ARCH](#) [DMA](#) [ADDR](#) [T](#) [64BIT](#) [CONFIG](#) [ARCH](#) [ENABLE](#) [SPLIT](#) [PMD](#) [PTLOCK](#) [CONFIG](#) [ARCH](#) [HAS](#) [CACHE](#) [LINE](#) [SIZE](#) [CONFIG](#) [ARCH](#) [HAS](#) [CPU](#) [AUTOPROBE](#) [CONFIG](#) [ARCH](#) [SUPPORTS](#) [INT128](#) [CONFIG](#) [ARCH](#) [SUPPORTS](#) [OPTIMIZED](#) [INLINING](#) [CONFIG](#) [ARCH](#) [SUPPORTS](#) [UPROBES](#) [CONFIG](#) [ARCH](#) [USES](#) [PG](#) [UNCACHED](#) [CONFIG](#) [ARCH](#) [USE](#) [BUILTIN](#) [BSWAP](#) [CONFIG](#) [ARCH](#) [USE](#) [CMPXCHG](#) [LOCKREF](#) [CONFIG](#) [ASSOCIATIVE](#) [ARRAY](#) [CONFIG](#) [AUDIT](#) [CONFIG](#) [AUDITSYSCALL](#) [CONFIG](#) [BASE](#) [SMALL](#) [CONFIG](#) [BINARY](#) [PRINTF](#) [CONFIG](#) [BLK](#) [CGROUP](#) [CONFIG](#) [BLK](#) [DEV](#) [INTEGRITY](#) [CONFIG](#) [BLK](#) [DEV](#) [IO](#) [TRACE](#) [CONFIG](#) [BLOCK](#) [CONFIG](#) [BSD](#) [PROCESS](#) [ACCT](#) [CONFIG](#) [BUG](#) [CONFIG](#) [CC](#) [STACKPROTECTOR](#) [CONFIG](#) [CGROUPS](#) [CONFIG](#) [CGROUP](#) [CPUACCT](#) [CONFIG](#) [CGROUP](#) [DEVICE](#) [CONFIG](#) [CGROUP](#) [FREEZER](#) [CONFIG](#) [CGROUP](#) [PERF](#) [CONFIG](#) [CGROUP](#) [SCHED](#) [CONFIG](#) [CLOCKSOURCE](#) [WATCHDOG](#) [CONFIG](#) [COMPAT](#) [CONFIG](#) [CORE](#) [DUMP](#) [DEFAULT](#) [ELF](#) [HEADERS](#) [CONFIG](#) [CPUSSETS](#) [CONFIG](#) [DEBUG](#) [BUGVERBOSE](#) [CONFIG](#) [DEBUG](#) [RODATA](#) [CONFIG](#) [DETECT](#) [HUNG](#) [TASK](#) [CONFIG](#) [DEVTMPFS](#) [CONFIG](#) [EARLY](#) [PRINTK](#) [CONFIG](#) [EPOLL](#) [CONFIG](#) [FAIR](#) [GROUP](#) [SCHED](#) [CONFIG](#) [FANOTIFY](#) [CONFIG](#) [FILE](#) [LOCKING](#) [CONFIG](#) [FREEZER](#) [CONFIG](#) [FSNOTIFY](#) [CONFIG](#) [FS](#) [POSIX](#) [ACL](#) [CONFIG](#) [FTRACE](#) [MOUNT](#) [RECORD](#) [CONFIG](#) [FUNCTION](#) [GRAPH](#) [TRACER](#) [CONFIG](#) [FUTEX](#) [CONFIG](#) [GENERIC](#) [BUG](#) [CONFIG](#) [GENERIC](#) [BUG](#) [RELATIVE](#) [POINTERS](#) [CONFIG](#) [GENERIC](#) [FIND](#) [FIRST](#) [BIT](#) [CONFIG](#) [GENERIC](#) [PENDING](#) [IRQ](#) [CONFIG](#) [GENERIC](#) [SMP](#) [IDLE](#) [THREAD](#) [CONFIG](#) [HAS](#) [IOPORT](#) [CONFIG](#) [HAVE](#) [ALIGNED](#) [STRUCT](#) [PAGE](#) [CONFIG](#) [HAVE](#) [ARCH](#) [SOFT](#) [DIRTY](#) [CONFIG](#) [HAVE](#) [CMPXCHG](#) [DOUBLE](#) [CONFIG](#) [HAVE](#) [KVM](#) [CONFIG](#) [HAVE](#) [MEMBLOCK](#) [NODE](#) [MAP](#) [CONFIG](#) [HAVE](#) [MEMORY](#) [PRESENT](#) [CONFIG](#) [HAVE](#) [SETUP](#) [PER](#) [CPU](#) [AREA](#) [CONFIG](#) [HAVE](#) [UNSTABLE](#) [SCHED](#) [CLOCK](#) [CONFIG](#) [HIBERNATION](#) [CONFIG](#) [HIGH](#) [RES](#) [TIMERS](#) [CONFIG](#) [HOTPLUG](#) [CPU](#) [CONFIG](#) [HUGETLBFS](#) [CONFIG](#) [HUGETLB](#) [PAGE](#) [CONFIG](#) [HZ](#) [CONFIG](#) [IA32](#) [EMULATION](#)

Each link on the page takes you to a description of the macro containing:

- The macro name, how many times it was referenced and whether it is still defined at the end of preprocessing.
- The verbatim macro definition (rewritten over several lines for long macros).
- File name and line number of definition, linked.
- Places that the macro was used, directly or indirectly. This is a table of file paths with links to the use point.
- **Dependencies, two way:**
  - Macros that this macro invokes.
  - Macros that invoke this macro.

**BITMAP\_LAST\_WORD\_MASK [References: 10] Defined? True**

```
#define BITMAP_LAST_WORD_MASK(nbits) ( ((nbits) % BITS_PER_LONG) ? (1UL<<((nbits) % \
BITS_PER_LONG))-1 : ~0UL )
```

defined @ [/Users/paulross/dev/linux/linux-3.13/include/linux/bitmap.h#150](#)

/	Users/	paulross/	dev/	linux/	linux-3.13/	include/	linux/	bitmap.h: <a href="#">176-20</a> <a href="#">228-20</a> <a href="#">237-31</a> <a href="#">246-29</a> <a href="#">255-34</a> <a href="#">263-20</a> <a href="#">271-23</a> <a href="#">279-30</a> <a href="#">296-24</a>
						kernel/	cpu.c: <a href="#">653-47</a>	

I depend on these macros:

<a href="#">BITMAP_LAST_WORD_MASK</a>	<a href="#">BITS_PER_LONG</a>
---------------------------------------	-------------------------------

These macros depend on me:

<a href="#">BITMAP_LAST_WORD_MASK</a>	<a href="#">CPU_MASK_LAST_WORD</a>	<a href="#">CPU_BITS_ALL</a>
		<a href="#">CPU_MASK_ALL</a>
	<a href="#">NODE_MASK_LAST_WORD</a>	<a href="#">NODE_MASK_ALL</a>

## Status

## Licence

CPIP is a C/C++ Preprocessor implemented in Python. Copyright (C) 2008-2017 Paul Ross

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.

## Credits

This package was created with [Cookiecutter](#) and the [audreyr/cookiecutter-pypackage](#) project template.

Also many thanks to [SourceForge](#) that hosted this project for many years.

```
$ python3 CPIPMain.py -kp -l20 -o ../../output/linux/cpu -S __STDC__=1 -D __KERNEL__ -
→D __EXPORTED_HEADERS__ -D BITS_PER_LONG=64 -D CONFIG_HZ=100 -D __x86_64__ -D __GNUC__
→_4 -D __has_feature(x)=0 -D __has_extension=__has_feature -D __has_attribute=__has_
→feature -D __has_include=__has_feature -P ~/dev/linux/linux-3.13/include/linux/
→kconfig.h -J /usr/include/ -J /usr/include/c++/4.2.1/ -J /usr/include/c++/4.2.1/tr1/
→ -J /Users/paulross/dev/linux/linux-3.13/include/ -J /Users/paulross/dev/linux/
→linux-3.13/include/uapi/ -J ~/dev/linux/linux-3.13/arch/x86/include/uapi/ -J ~/dev/
→linux/linux-3.13/arch/x86/include/ -J ~/dev/linux/linux-3.13/arch/x86/include/
→generated/ ~/dev/linux/linux-3.13/kernel/cpu.c
```



## CHAPTER 3

---

### Installation

---

CPIP has been tested with Python 2.7 and 3.3 to 3.6. CPIP used to run just fine on Windows but I haven't had a recent opportunity (or reason) to test CPIP on a Windows box.

First make a virtual environment in your *<PYTHONVENV>*, say *~/pyvenvs*:

```
$ python3 -m venv <PYTHONVENV>/CPIP
$ . <PYTHONVENV>/CPIP/bin/activate
(CPIP) $
```

### Stable release

To install cpip, run this command in your terminal:

```
(CPIP) $ pip install cpip
```

This is the preferred method to install cpip, as it will always install the most recent stable release.

If you don't have `pip` installed, this [Python installation guide](#) can guide you through the process.

### From sources

The sources for cpip can be downloaded from the [Github repo](#).

You can either clone the public repository:

```
(CPIP) $ git clone git://github.com/paulross/cpip
```

Or download the [tarball](#):

```
(CPIP) $ curl -OL https://github.com/paulross/cpip/tarball/master
```

Once you have a copy of the source, you can install it with:

```
(CPIP) $ python setup.py install
```

Install the test dependencies and run CPIP's tests:

```
(CPIP) $ pip install pytest
(CPIP) $ pip install pytest-runner
(CPIP) $ python setup.py test
```

## Developing with CPIP

If you are developing with CPIP you need test coverage and documentation tools.

### Test Coverage

Install `pytest-cov`:

```
(CPIP) $ pip install pytest-cov
```

The most meaningful invocation that eliminates the top level tools is:

```
(CPIP) $ pytest --cov=cpip.core --cov=cpip.plot --cov=cpip.util --cov-report html_
↪tests/
```

### Documentation

If you want to build the documentation you need to:

```
(CPIP) $ pip install Sphinx
(CPIP) $ cd docs
(CPIP) $ make html
```

The landing page is `docs/_build/html/index.html`.

## Testing the Demo Code

See the *PpLexer Tutorial* for an example of running a CPIP PpLexer on the demonstration code. This gives the core CPIP software a good workout.



---

## CPIP Introduction

---

CPIP is a C/C++ pre-processor implemented in Python. Most pre-processors regard pre-processing as a dirty job that just has to be done as soon as possible. This can make it very hard to track down subtle defects at the pre-processing stage as pre-processors throw away a lot of useful information in favor of getting the result as cheaply as possible.

Few developers really understand pre-processing, to many it is an obscure bit of black magic. CPIP aims to improve that and by recording every detail of preprocessing so CPIP can produce some wonderfully visual information about file dependencies, macro usage and so on.

CPIP is not designed to be a replacement for `cpp` (or any other established pre-processor), instead CPIP regards clarity and understanding as more important than speed of processing.

CPIP takes its standard as C99 or, more formally, ISO/IEC 9899:1999 (E)<sup>1</sup>.

## Pre-processing C and C++

The basic task of any preprocessor is to produce a *Translation Unit* for a compiler to work with. To do this the pre-processor has to do three inter-related tasks<sup>2</sup>:

- File inclusion i.e. responding to `#include` commands.
- Conditional Compilation `#if`, `#ifdef` etc.
- Macro definition and replacement.

## File Inclusion

CPIP supports file inclusion just like any other pre-processor. In fact it goes further as CPIP recognises that whilst the C99 standard (and any other standard) specifies the syntax of the `#include` statement it leaves it as implementation defined *how* the file is located.

---

<sup>1</sup> Other standards are of interest: “C++98” [ISO/IEC 14882:1998(E)] describes more limited pre-processing (no variadic macros for example). “C++11” [ISO/IEC JTC 1/SC 22 N 4411 in draft] and C++14 does not substantially change this. In any case CPIP attempts to emulate common custom and practice (yes, including variadic macros).

<sup>2</sup> Of course the preprocessor has to do many other minor tasks such as replacing trigraphs and removing comments.

CPIP provides a reference implementation that behaves as CPP/RVCT/LLVM behave. CPIP also allows users to construct their own include handlers that obtain files from, for example, a URL or database.

## Conditional Compilation

CPIP supports all conditional compilation statements. What is more CPIP can generate a conditionally compiled view of the source code which makes it much easier to see what part of the code is active.

## Macro Replacement

CPIP supports macro replacement according to C99, CPIP keeps track of where macros were defined (and undefined) and where they were either tested (by an `#if` statement for example) or used in a substitution. All this information is available using public APIs after CPIP has finished processing a Translation Unit.

Macros represent one of the most complicated parts of preprocessing, it seems simple doesn't it? But consider this source code:

```
#define f(a) a*g
#define g(a) f(a)
f(2)(9)
```

What is the result of the last statement?

It is either:

```
2*f(9)
```

Or:

```
2*9*g
```

Which is it? Puzzled? Well the C standards body responded thus:

*“The C89 Committee intentionally left this behavior ambiguous as it saw no useful purpose in specifying all the quirks of preprocessing for such questionably useful constructs.”<sup>3</sup>*

So any pre-processor implementation could produce either result at *any time* and it would still be a compliant implementation.

## CPIP and Pre-processing

CPIP is capable of doing all these aspects of preprocessing and it produces a Translation Unit just like any other pre-processor.

What makes CPIP unique is that it retains all pre-processing information discovered along the way and can present it in many ways. CPIP provides a number of interfaces to that information, not least:

- A command line tool that acts as a C/C++ pre-processor but produces all sorts of wonderful information about pre-processing: [CPIPMain.py Examples](#).
- A Python interface to pre-processing via the PpLexer, see the [PpLexer Tutorial](#). If you want to construct your own pre-processor or understand a specific aspect of preprocessing then this is for you.

---

<sup>3</sup> Rationale for International Standard - Programming Languages - C Revision 5.10 April-2003 Sect. 6.10.3.4

## CPIP Core Architecture

CPIP provides a set of *Command Line Tools* built round a core of Python code. The architecture of this core code is illustrated below, at the heart of it is the `PpLexer`. The user interacts with this in two ways:

- **Constructing a `PpLexer` with the following:**
  - A file-like object that represents the *Initial Translation Unit* i.e. the file to be pre-processed.
  - Any pre-include files.
  - An include handler that manages `#include` statements.
  - *Optionally*: a `CppDiagnostic` to handle error conditions.
  - *Optionally*: a `Pragmahandler` to handle `#pragma` statements.
- Processing the file (and its `#include`'s) token by token.

For the `PpLexer` its construction is fairly straightforward; it just takes a reference to the user supplied objects.

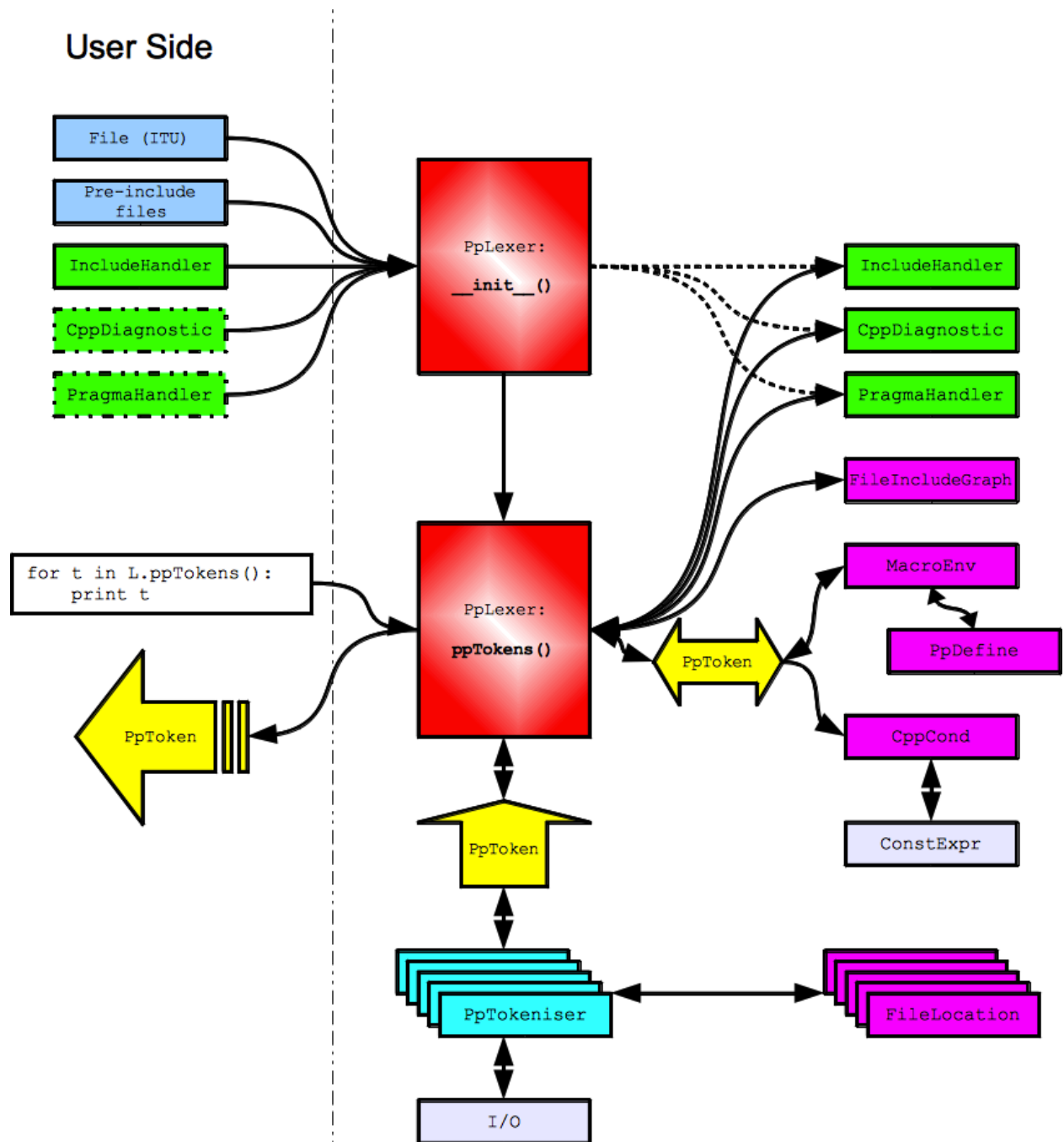
Processing the ITU is a more serious matter. The `PpLexer` uses a `PpTokeniser` to generate pre-processing tokens (shown in yellow below) according to translation phases one to three. The `PpTokeniser` also keeps track of logical to physical file location.

Depending on the parser state the `PpLexer` may/may not pass the token to various internal objects (shown in purple below) that keep track of:

- File inclusion.
- Conditional compilation.
- Macro Environment.

The resulting token (if any) after that processing is yielded to the user.

An extremely useful feature of CPIP is that the `PpLexer` maintains all these data structures and provides an interface to them for the user. Some examples of what can be done with this information is here: *[CPIPMain.py Examples](#)*.



---

## CPIPMain.py Examples

---

### Screenshots

This section shows some screenshots of `CPIPMain.py`'s output. *Some Real Examples* are shown below.

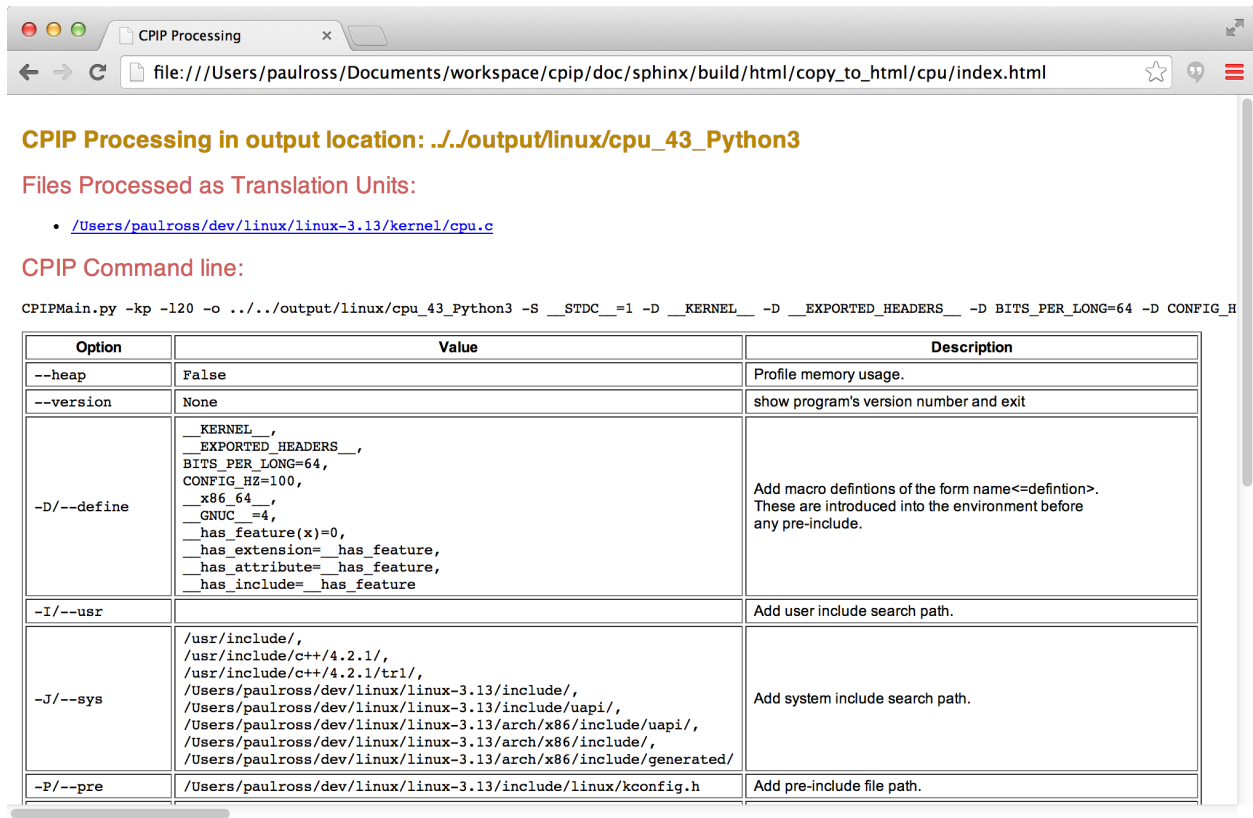
`CPIPMain.py` produces a set of HTML and SVG pages for each source file preprocessed. As well as the *Translation Unit* `CPIPMain.py` generates information about the three important tasks for a preprocessor: file inclusion, conditional compilation and macro replacement.

### Home Page

The `index.html` shows the list of files preprocessed in this pass (linked to file specific pages).

It also shows the command line used and an explanation from the `CPIPMain.py` help system as to what each option means.

For example:



**CPIP Processing in output location: `././output/linux/cpu_43_Python3`**

**Files Processed as Translation Units:**

- `/Users/paulross/dev/linux/linux-3.13/kernel/cpu.c`

**CPIP Command line:**

```
CPIPMain.py -kp -l20 -o .././output/linux/cpu_43_Python3 -S __STDC__=1 -D __KERNEL__ -D __EXPORTED_HEADERS__ -D BITS_PER_LONG=64 -D CONFIG_H
```

Option	Value	Description
<code>--heap</code>	False	Profile memory usage.
<code>--version</code>	None	show program's version number and exit
<code>-D/--define</code>	<code>KERNEL</code> , <code>EXPORTED_HEADERS</code> , <code>BITS_PER_LONG=64</code> , <code>CONFIG_HZ=100</code> , <code>__x86_64__</code> , <code>__GNUC__=4</code> , <code>__has_feature(x)=0</code> , <code>__has_extension=__has_feature</code> , <code>__has_attribute=__has_feature</code> , <code>__has_include=__has_feature</code>	Add macro definitions of the form <code>name&lt;=definition&gt;</code> . These are introduced into the environment before any pre-include.
<code>-I/--usr</code>		Add user include search path.
<code>-J/--sys</code>	<code>/usr/include/</code> , <code>/usr/include/c++/4.2.1/</code> , <code>/usr/include/c++/4.2.1/tr1/</code> , <code>/Users/paulross/dev/linux/linux-3.13/include/</code> , <code>/Users/paulross/dev/linux/linux-3.13/include/uapi/</code> , <code>/Users/paulross/dev/linux/linux-3.13/arch/x86/include/uapi/</code> , <code>/Users/paulross/dev/linux/linux-3.13/arch/x86/include/</code> , <code>/Users/paulross/dev/linux/linux-3.13/arch/x86/include/generated/</code>	Add system include search path.
<code>-P/--pre</code>	<code>/Users/paulross/dev/linux/linux-3.13/include/linux/kconfig.h</code>	Add pre-include file path.

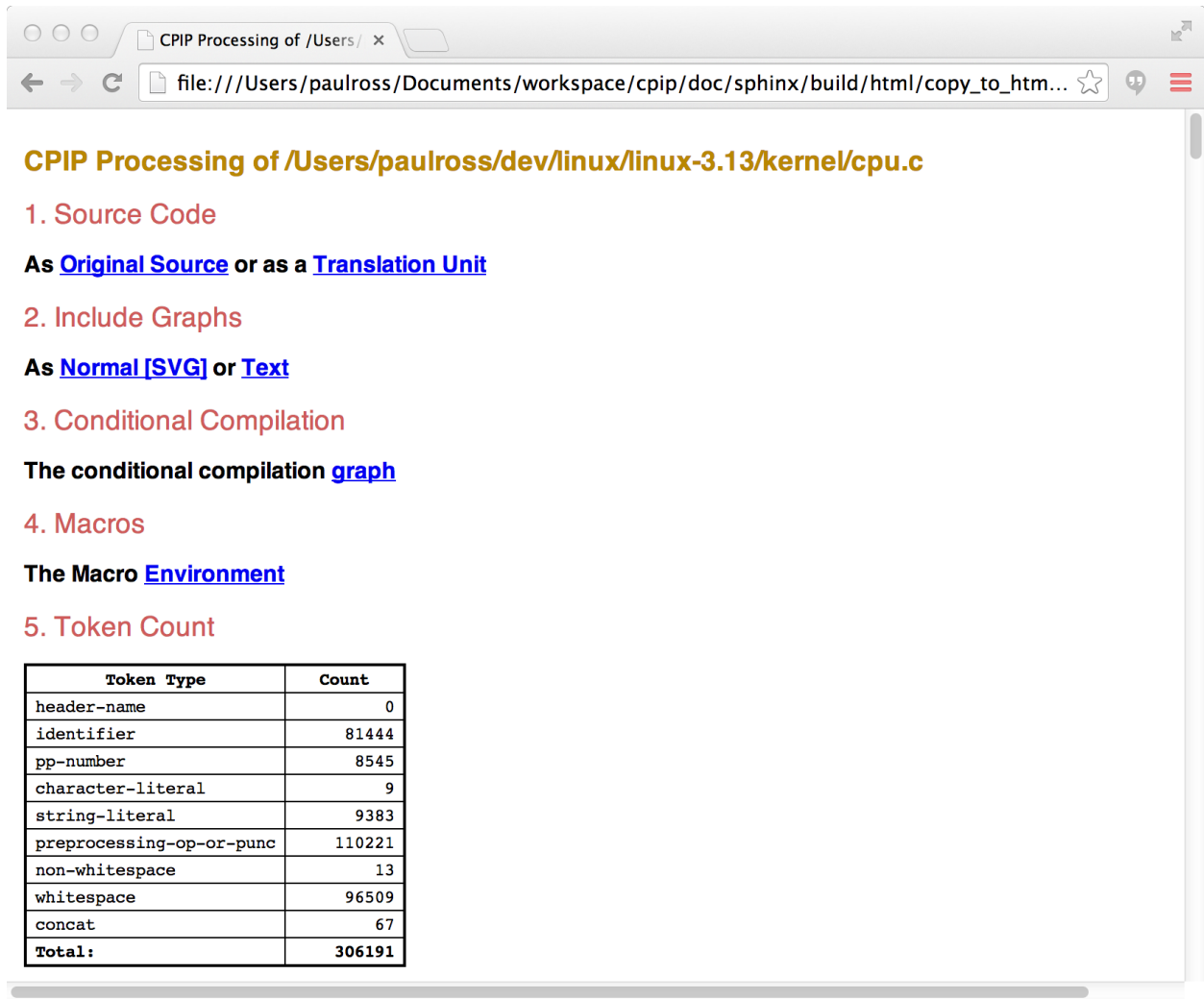
Each in the list of files preprocessed in this pass is linked to file specific page.

## Preprocessed File Specific Page

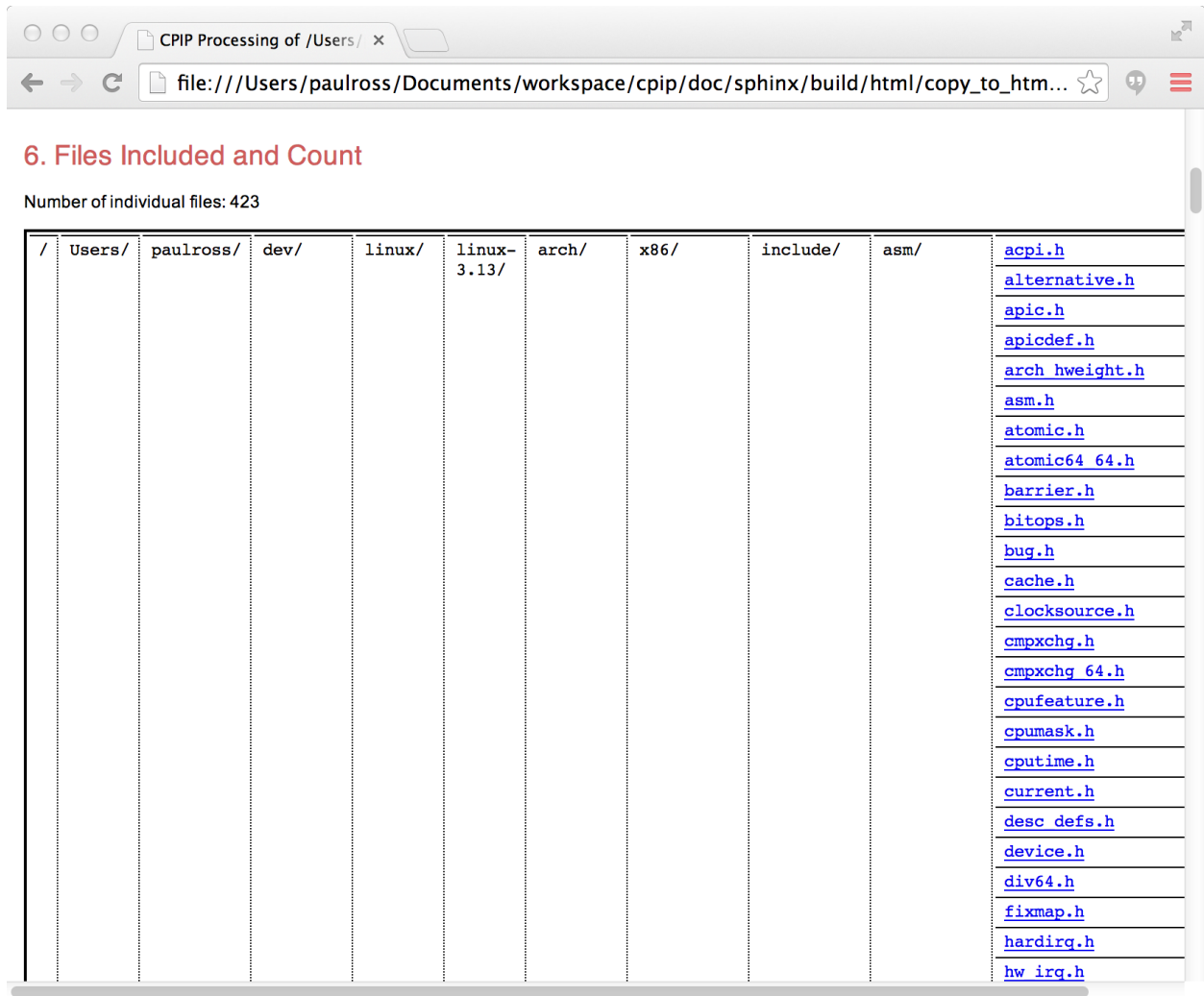
These describe the results of preprocessing a single file, it contains links to:

1. The source and the translation unit as HTML.
2. The results of file inclusion.
3. The results of conditional compilation.
4. Macro processing results.
5. The total token count.
6. What files were included and how many times.

The top of the page includes links to these sections (described in detail below):



Further down the page is a table showing what files were included, from where and how many times:



6. Files Included and Count

Number of individual files: 423

/	Users/	paulross/	dev/	linux/	linux-3.13/	arch/	x86/	include/	asm/
									<a href="#">acpi.h</a>
									<a href="#">alternative.h</a>
									<a href="#">apic.h</a>
									<a href="#">apicdef.h</a>
									<a href="#">arch_hweight.h</a>
									<a href="#">asm.h</a>
									<a href="#">atomic.h</a>
									<a href="#">atomic64_64.h</a>
									<a href="#">barrier.h</a>
									<a href="#">bitops.h</a>
									<a href="#">bug.h</a>
									<a href="#">cache.h</a>
									<a href="#">clocksource.h</a>
									<a href="#">cmpxchg.h</a>
									<a href="#">cmpxchg_64.h</a>
									<a href="#">cpufeature.h</a>
									<a href="#">cpumask.h</a>
									<a href="#">cputime.h</a>
									<a href="#">current.h</a>
									<a href="#">desc_defs.h</a>
									<a href="#">device.h</a>
									<a href="#">div64.h</a>
									<a href="#">fixmap.h</a>
									<a href="#">hardirq.h</a>
									<a href="#">hw_irq.h</a>

Here is an explanation for the table:



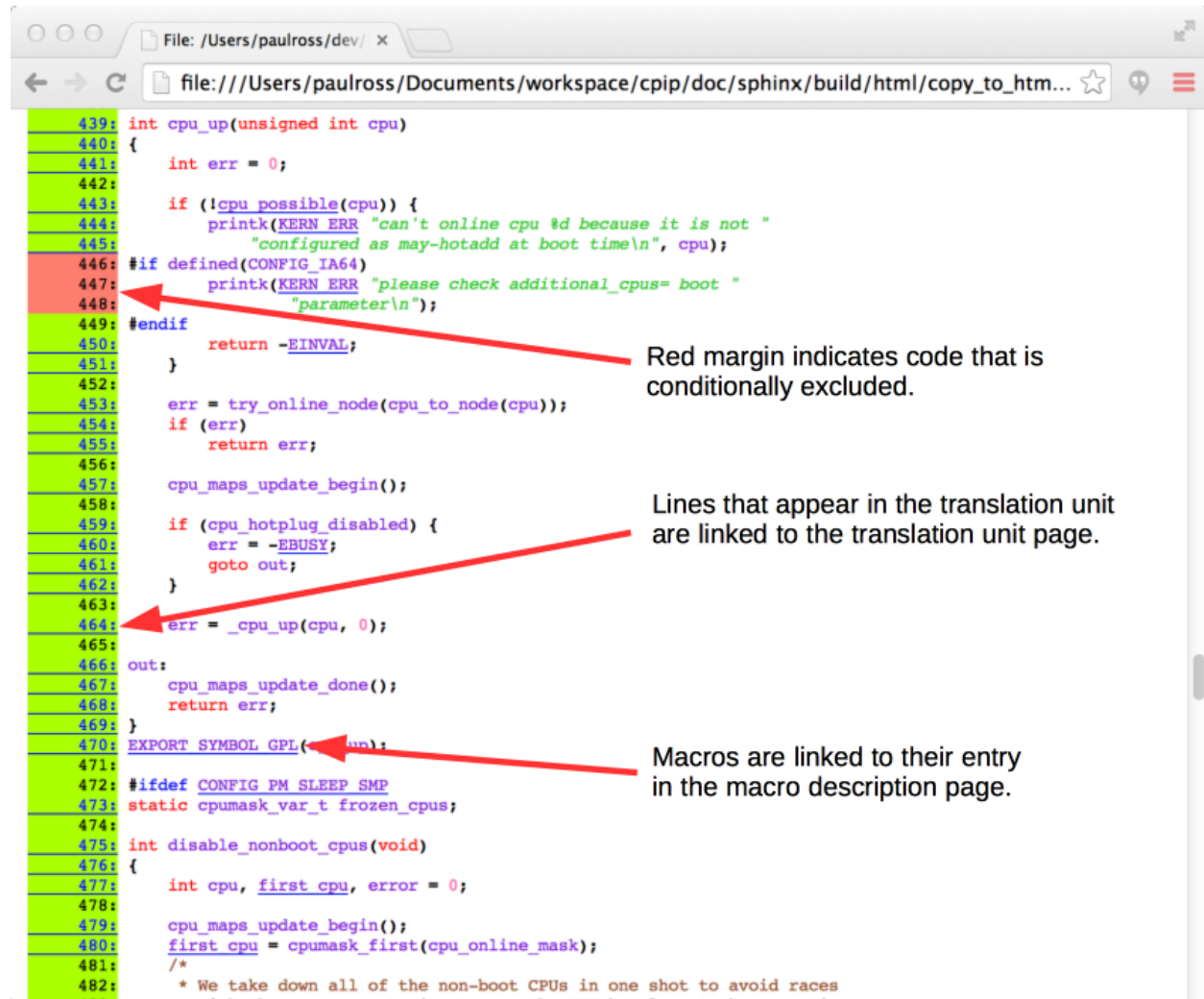
						<a href="#">sysinfo.h</a>	1
						<a href="#">taskstats.h</a>	1
						<a href="#">time.h</a>	1
						<a href="#">timex.h</a>	1
						<a href="#">types.h</a>	1
						<a href="#">unistd.h</a>	2
						<a href="#">wait.h</a>	1
						<a href="#">xattr.h</a>	1
				video/		<a href="#">edid.h</a>	1
			video/			<a href="#">edid.h</a>	1
			kernel/			<a href="#">cpu.c</a>	1
						<a href="#">smpboot.h</a>	1
usr/	include/					<a href="#">Availability.h</a>	3
						<a href="#">AvailabilityInternal.h</a>	1
						<a href="#">types.h</a>	1
	c++/	4.2.1/	bits/			<a href="#">c++config.h</a>	1
						<a href="#">cpu defines.h</a>	1
						<a href="#">os defines.h</a>	1
						<a href="#">cstdarg</a>	1
			trl/			<a href="#">cstdarg</a>	1
						<a href="#">stdarg.h</a>	4
						<a href="#">gethostuuid.h</a>	1
	i386/					<a href="#">types.h</a>	1
	machine/					<a href="#">types.h</a>	1
	sys/					<a href="#">posix_availability.h</a>	1
						<a href="#">select.h</a>	1
						<a href="#">symbol aliasing.h</a>	1
						<a href="#">types.h</a>	2
	_types/					<a href="#">dev t.h</a>	1

## Original File and the Translation Unit

### Original File

All processed source code (original file and included files) is presented as syntax highlighted HTML.

The syntax is the C pre-preprocessor language. Macro names are linked to their definition in the *Macro Definitions* page.



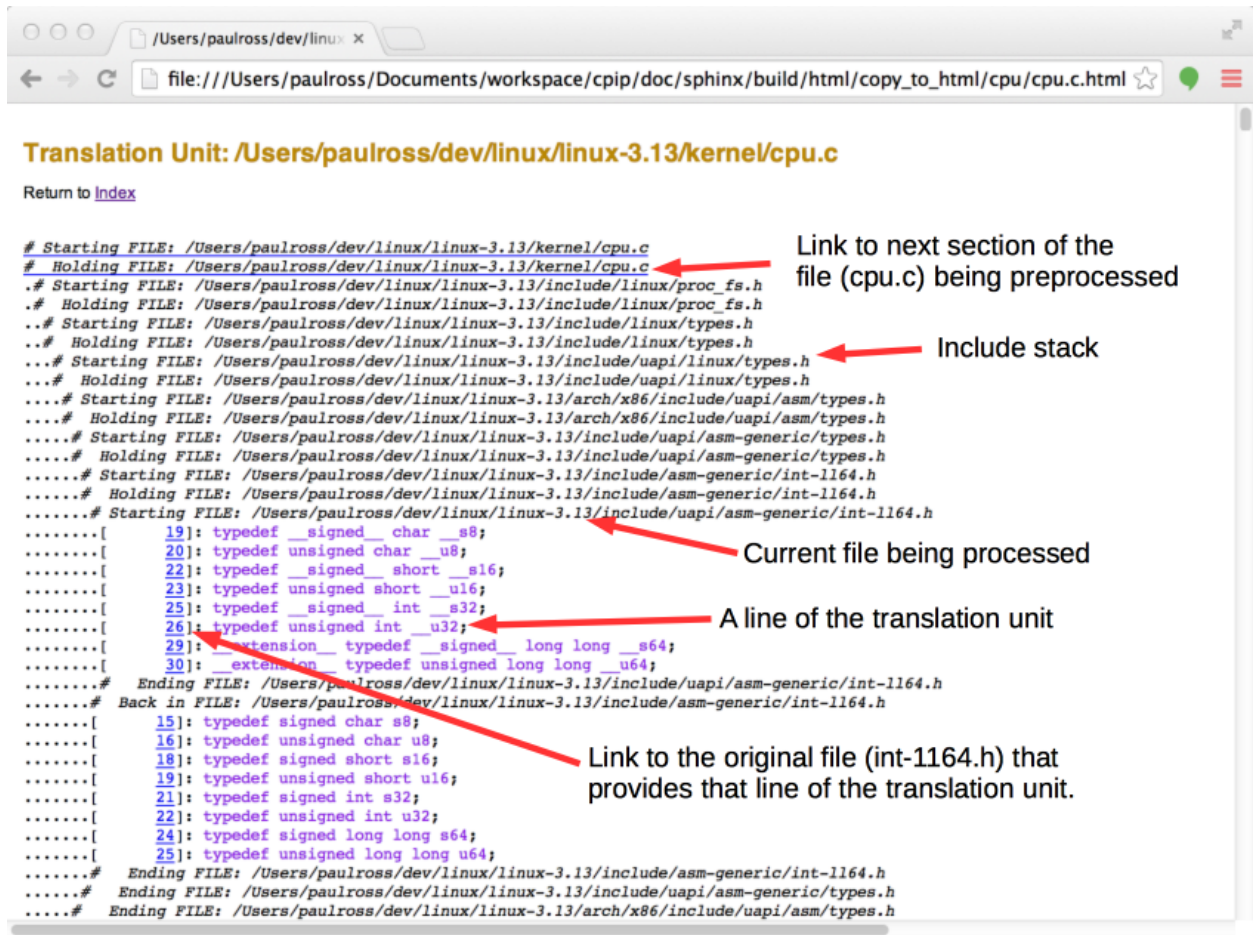
## Translation Unit

The preprocessed file and all its `#include`'s become a *Translation Unit* which `CPIPMain.py` represents as an HTML page.

Each `#include` statement is represented in a nested fashion, any source code in the translation unit is presented syntax highlighted. The syntax is, of course, the C pre-processor language thus both `typedef` and `char` are pre-processor *identifiers* even if later on `typedef` is seen as a C keyword.

The numbered links thus [ 19 ] are to an HTML representation of the original source code file/line.

The other navigational element present is when the file path is the file being pre-processed a forward link is there to the next part of this file, thus skipping over intermediate `#include`'d code.



Further down you can see the actual code from `cpu.c`, notice the macro expansion on line 76.

```

..# Ending FILE: /Users/paulross/dev/linux/linux-3.13/kernel/smpboot.h
..# Back in FILE: /Users/paulross/dev/linux/linux-3.13/kernel/cpu.c
.[ 27]: static struct mutex cpu_add_remove_lock = { .count = { (1) }, .wait_lock = (spinlock_t) { { .rlock = { .raw_lock
.[    ] } }, .wait_list = { &(cpu_add_remove_lock.wait_list), &(cpu_add_remove_lock.wait_list) } };
.[ 33]: void cpu_maps_update_begin(void)
.[ 34]: {
.[ 35]: mutex_lock(&cpu_add_remove_lock);
.[ 36]: }
.[ 38]: void cpu_maps_update_done(void)
.[ 39]: {
.[ 40]: mutex_unlock(&cpu_add_remove_lock);
.[ 41]: }
.[ 43]: static struct raw_notifier_head cpu_chain = { .head = ((void *)0) };
.[ 48]: static int cpu_hotplug_disabled;
.[ 52]: static struct {
.[ 53]: struct task_struct *active_writer;
.[ 54]: struct mutex lock;
.[ 59]: int refcount;
.[ 60]: } cpu_hotplug = {
.[ 61]: .active_writer = ((void *)0),
.[ 62]: .lock = { .count = { (1) }, .wait_lock = (spinlock_t) { { .rlock = { .raw_lock = { { 0 } }, } }, .wait_list =
.[    ] .wait_list), &(cpu_hotplug.lock.wait_list) } },
.[ 63]: .refcount = 0,
.[ 64]: };
.[ 66]: void get_online_cpus(void)
.[ 67]: {
.[ 68]: do { _cond_resched(); } while (0);
.[ 69]: if (cpu_hotplug.active_writer == get_current())
.[ 70]: return;
.[ 71]: mutex_lock(&cpu_hotplug.lock);
.[ 72]: cpu_hotplug.refcount++;
.[ 73]: mutex_unlock(&cpu_hotplug.lock);
.[ 75]: }
.[ 76]: extern typeof(get_online_cpus) get_online_cpus; extern void * __crc_get_online_cpus __attribute__((weak)); static \
.[    ] const unsigned long __kcrctab_get_online_cpus __attribute__((__used__)) __attribute__((section("__kcrctab" \
.[    ] "_gpl" "+" "get_online_cpus"), unused)) = (unsigned long) &__crc_get_online_cpus; static const char __kstrtab_g
.[    ] [] __attribute__((section("__ksymtab_strings"), aligned(1))) = "get_online_cpus"; const struct kernel_symbol \
.[    ] __ksymtab_get_online_cpus __attribute__((__used__)) __attribute__((section("__ksymtab" "_gpl" "+" "get_online_
.[    ] ), unused)) = { (unsigned long)&get_online_cpus, __kstrtab_get_online_cpus };
.[ 78]: void put_online_cpus(void)
.[ 79]: {
.[ 80]: if (cpu_hotplug.active_writer == get_current())
.[ 81]: return;

```

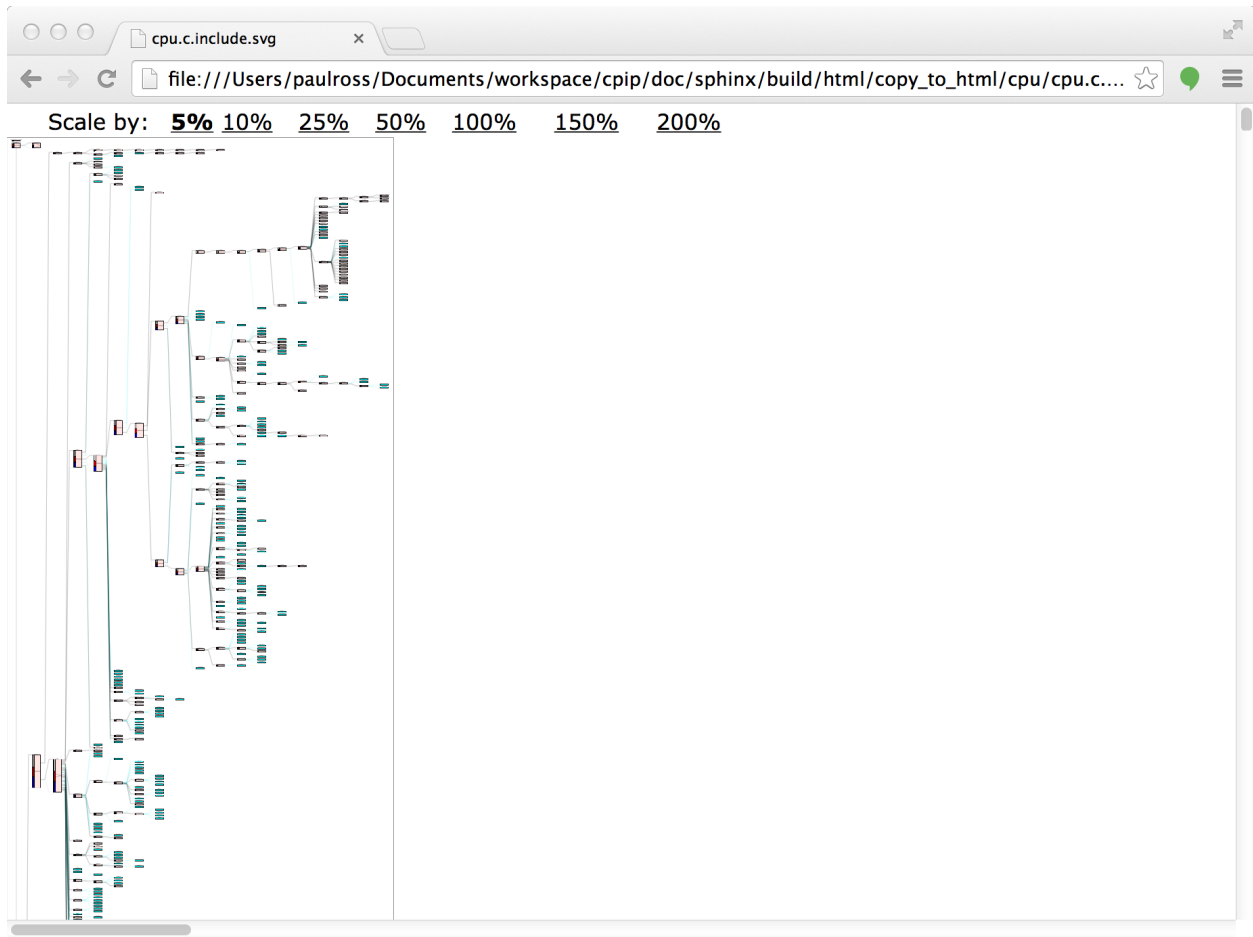
## The SVG Include Graph

The file specific page offers a link to an SVG visualisation of the file include graph.

## The Overall Picture

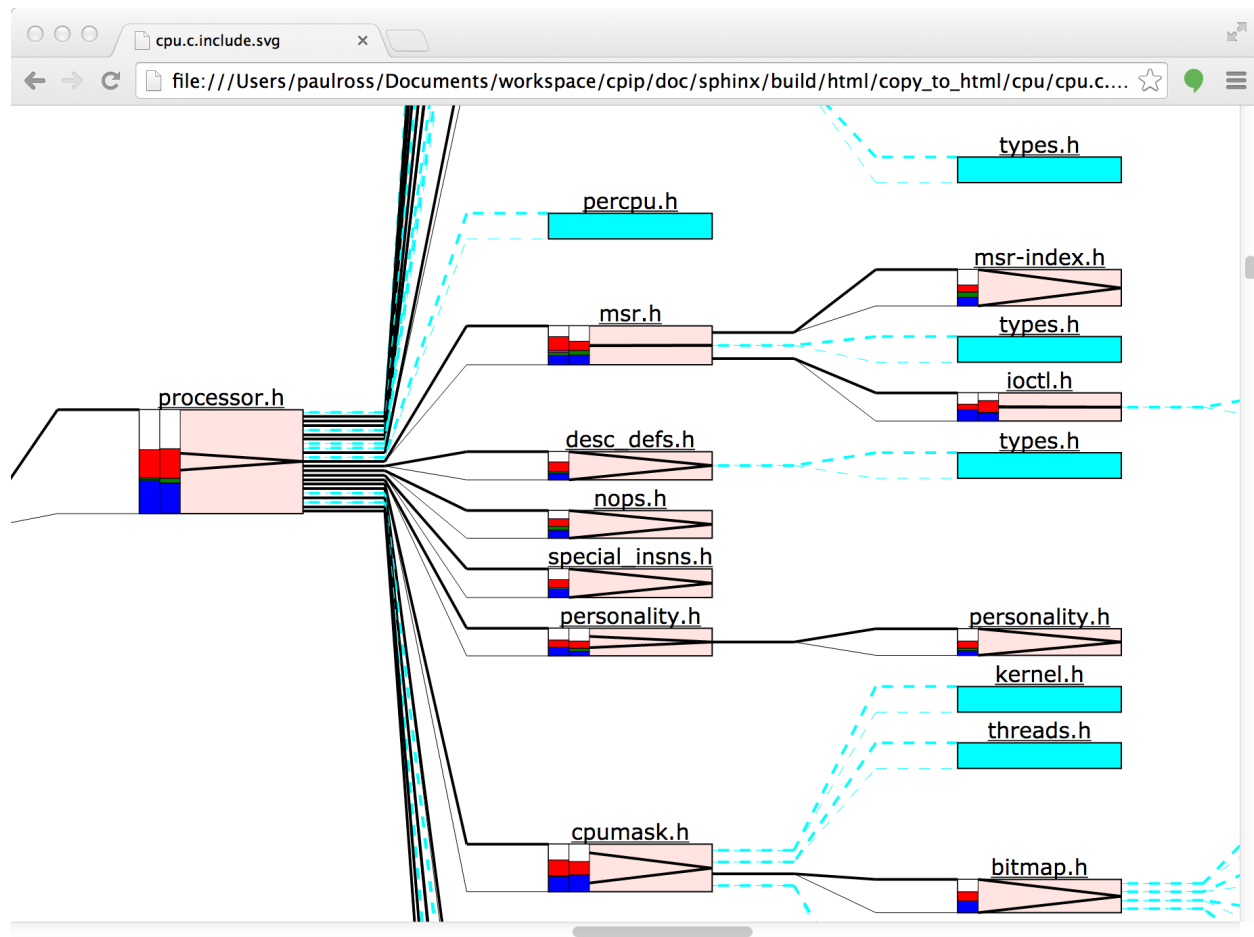
The diagram represents a tree with the root (the file being preprocessed) at center left. Each node represents a file and each edge represents an `#include` directive. Increasing include depth is left-to-right and source code order (i.e. order of the `#include` directives) is top to bottom.

At the top are various zoom factors that you can use to view the graph, initially the page opens at the smallest scale factor to give you an impression of what is going on:



## A Detailed Look

Zooming in to 100% on one part of the graph gives a wealth of information. In this picture the `processor.h` file is represented on the left and the files that it `#include`'s to its right.:



Each file is represented by a fixed width block, the height is proportional to the number of preprocessing tokens produced by a file (and its `#include`'s)<sup>1</sup>. Cyan coloured blocks represent files that are included but contain no effective content, usually because it has already been included and the header guards use conditional compilation to prevent preprocessing more than once (`types.h` for example).

The 'V' symbol in the block represents the relative size of the file and its descendants, if the 'V' touches top and bottom then all the tokens come from this file (`personality.h` for example). Where the 'V' is closed, or almost so, it means the bulk of the tokens are coming from the descendent includes (`msr.h` for example).

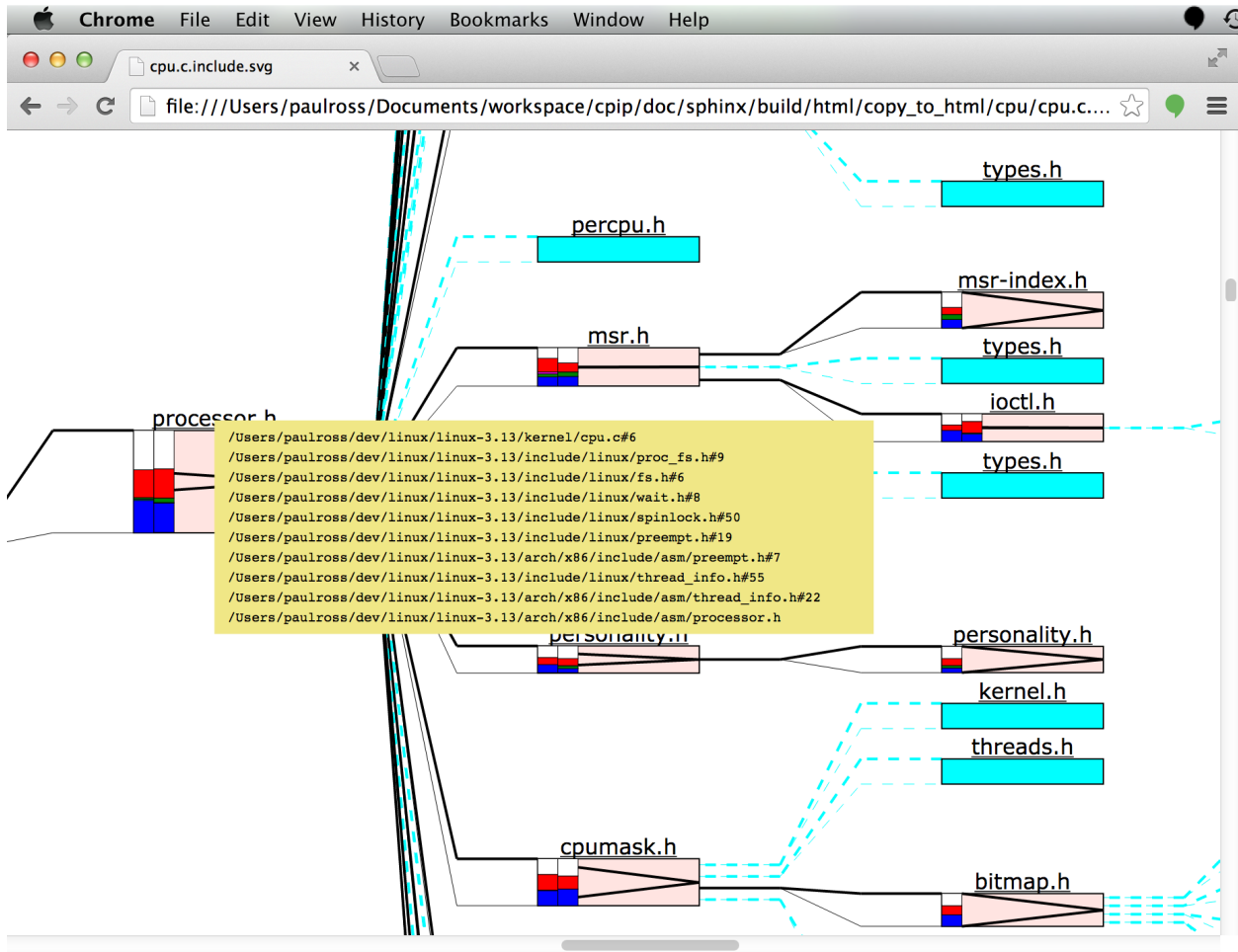
The coloured bars on the left represent the count of different token types, the left bar being the current file, the right bar being the total of the descendants. See below for which *Token Types* correspond to each colour.

Many parts of this diagram can display additional information when moving the mouse over various bits of the file block.

## File Path

For example mousing over the file name above the box shows the the absolute path of the file stack as a pop-up yellow block. At the top of this list is the file we are preprocessing, then the stack of included files downwards to `processor.h`:

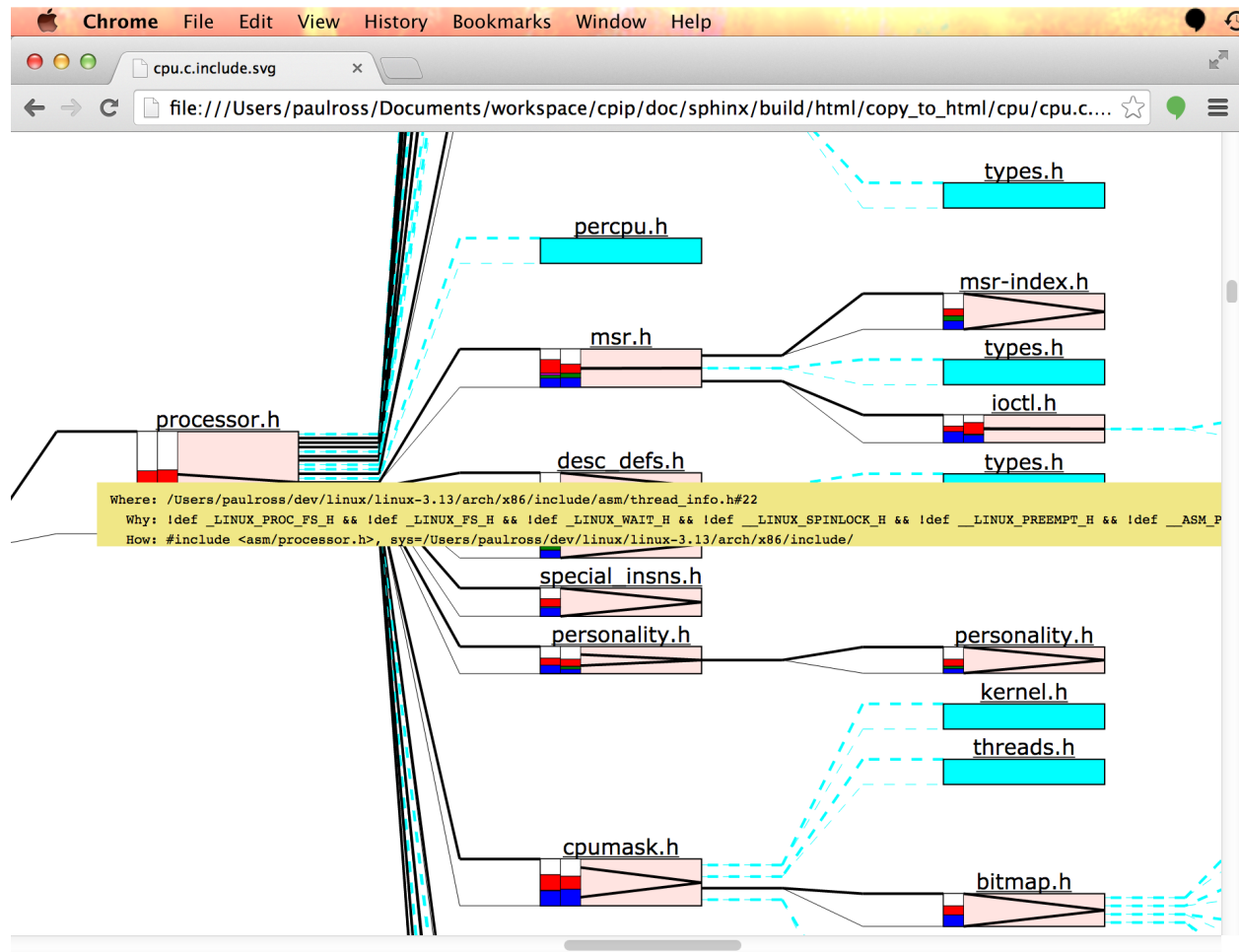
<sup>1</sup> A special case is that there may be a file "Unnamed Pre-Include" at the top left and joined to the preprocessed file with a thick light grey line. This 'virtual' file contains the macro declarations made on the `CPIPMain.py` command line.



### How it was Included?

Moving the mouse over to the left of the block reveals a host of information about the file inclusion process:





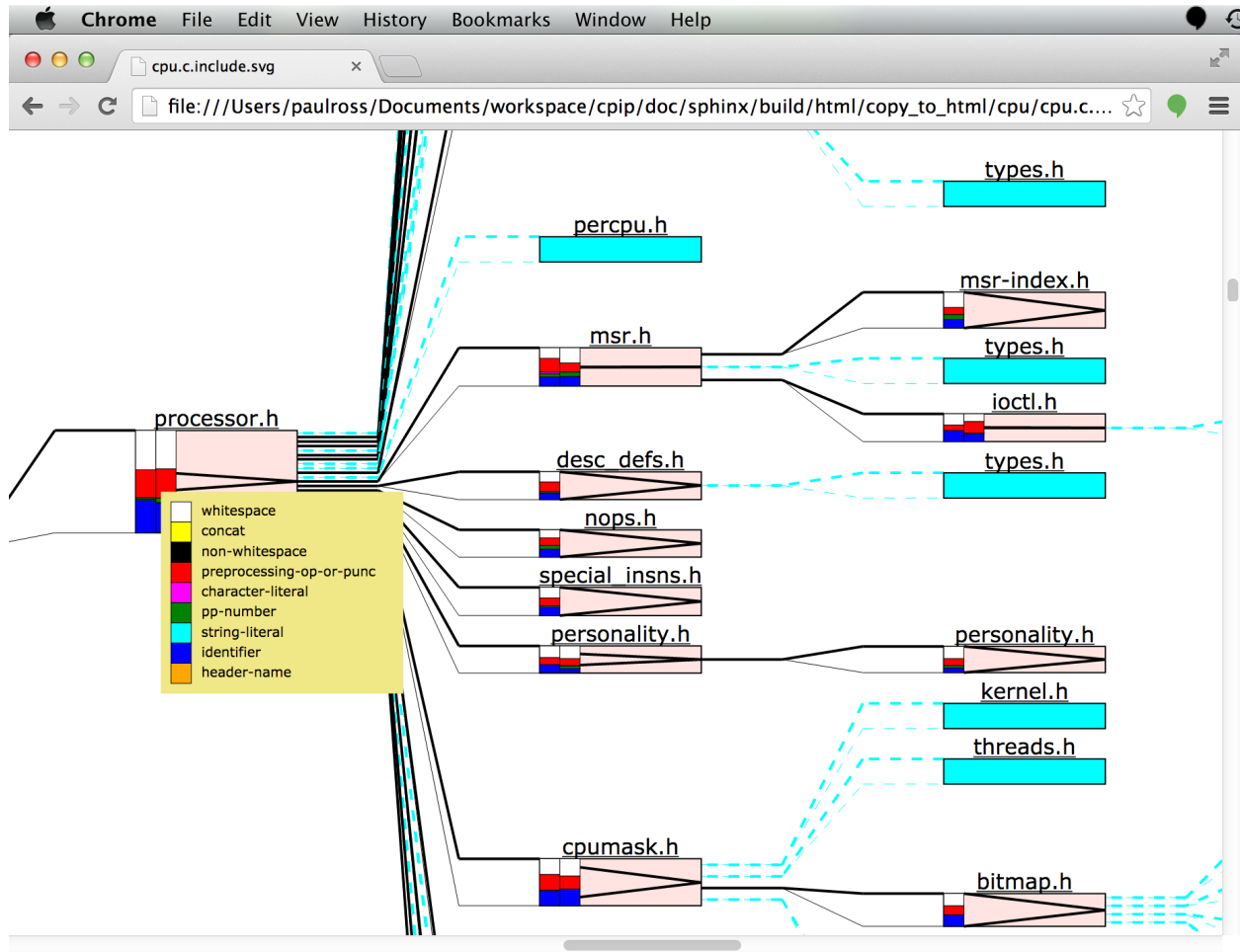
This pop-up yellow block contains the following:

- **Where:** *Where* this was included from. This file is included from line 22 of the `arch/x86/include/asm/thread_info.h` file.
- **Why:** *Why* it was included. This is the current state of the conditional compilation stack.
- **How:** *How* this file was included. This string starts with the text that follows the `#include` statement, in this case `#include <asm/processor.h>`. This is followed by the search results, in this case this file was found by searching the system includes (`sys=`) and was found in `arch/x86/include`. There may be more than one search made as fallback mechanisms are used and a failure will be shown with `None`. For example `usr=None sys=spam/eggs` means that the user include directories were searched first and nothing came up, then the system include directories were searched and the file was found in `spam/eggs`. A special case; `CP`: means ‘the current place’.

## Token Types

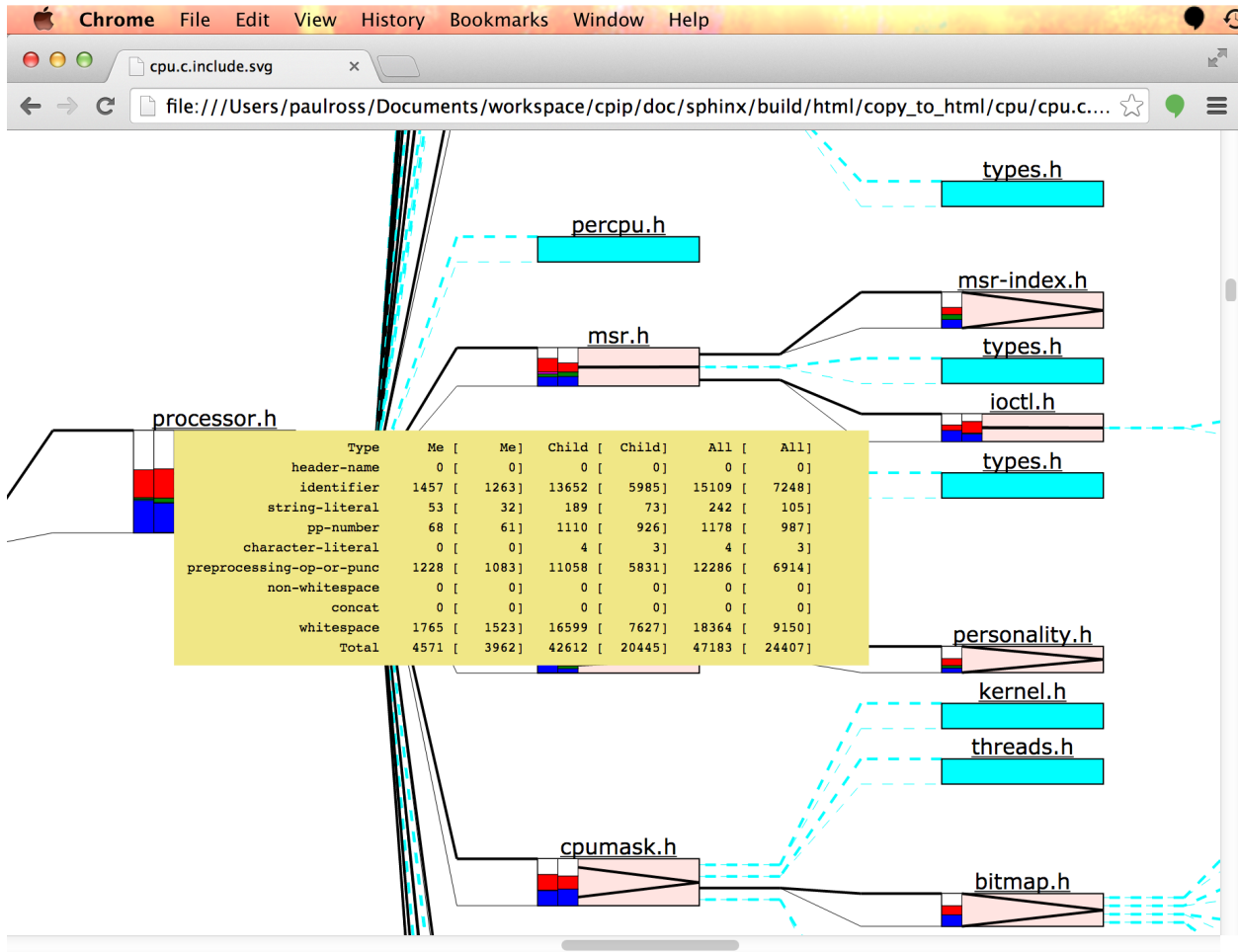
If you are interested in what types of preprocessor tokens were encountered then there is a host of information available to you. On the left hand side of each file block is a colour coded histogram of token types. If the file includes others then there will be two, the left hand one is for the file, the right hand one is for all the files it includes. Hovering over either histogram pops up the legend thus:





The actual count of tokens is seen when moving the mouse over the centre of the box. There are three sets of two columns, the left column of the set is total tokens, the right column is for *significant* tokens, that is those that are not conditionally excluded by `#if` etc. statements.

The first set is for the specific file, the second set is the descendents and the third set is the total.

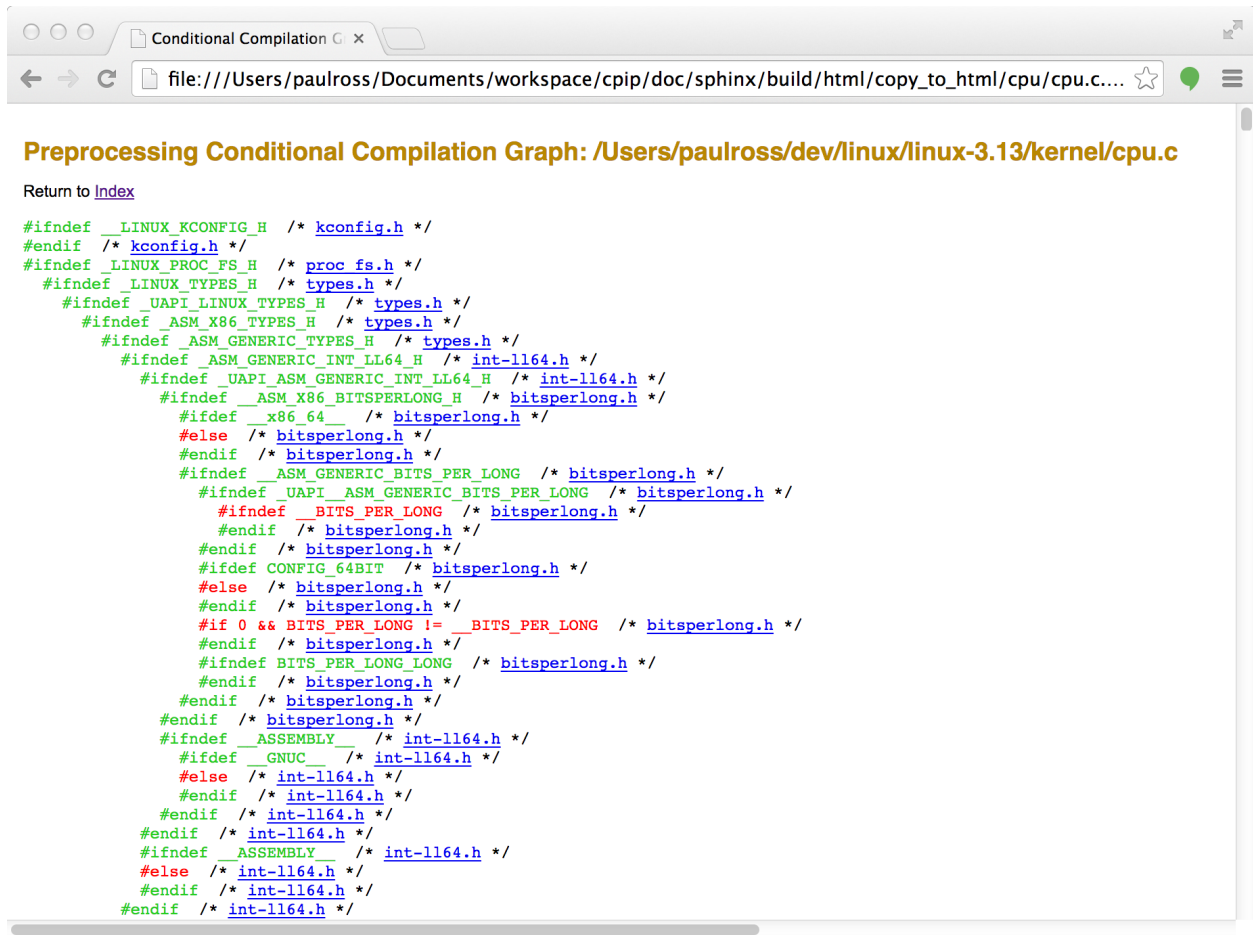


## Conditional Compilation

One tricky area for comprehending source code is understanding what code is conditionally compiled. Looking at a source file it is not immediately obvious which `#if` etc. statements are actually being processed.

As an aid CPIP produces an HTML page that is the translation unit with *only* the conditional compilation statements, what is more they are nested according to their logical execution condition and colour coded according to the resolved state; green means code will be part of the translation unit and red means those statements will be ignored. The links in the (artificial) comment following the statement are to the HTML representation of the file where the statement occurs.

Here is an example:



## Macro Definitions

CPIP retains all information about macros during preprocessing and the file specific page containing macro information starts like this:

Macro Environment for: /U x

file:///Users/paulross/Documents/workspace/cpip/doc/sphinx/build/html/copy\_to\_html/cpu/cpu.c...

## Macro Environment for: /Users/paulross/dev/linux/linux-3.13/kernel/cpu.c

Return to [Index](#)

### Referenced Macros:

- Inactive [22]: [ADDR](#) [IS SUBSYS ENABLED\(option\)](#) [IS SUBSYS ENABLED\(option\)](#) [IS SUBSYS ENABLED\(option\)](#) [READ LOCK ATOMIC\(n\)](#) [READ LOCK SIZE\(insn\)](#) [SUBSYS\(x\)](#) [SUBSYS\(x\)](#) [WRITE LOCK ADD\(n\)](#) [WRITE LOCK CMP](#) [WRITE LOCK SUB\(n\)](#) [SIG SET BINOP\(name,op\)](#) [SIG SET OP\(name,op\)](#) [copy to user overflow](#) [deprecated](#) [must check](#) [sig and\(x,y\)](#) [sig andn\(x,y\)](#) [sig not\(x\)](#) [sig or\(x,y\)](#) [copy user diag](#) [u32](#)
- Active [1642]: [ACCESS ONCE\(x\)](#) [ACPI\\_PDC\\_C\\_C1\\_FFH](#) [ACPI\\_PDC\\_C\\_C1\\_HALT](#) [ACPI\\_PDC\\_C\\_C2C3\\_FFH](#) [ACPI\\_PDC\\_C\\_CAPABILITY\\_SMP](#) [ACPI\\_PDC\\_EST\\_CAPABILITY\\_SWSMP](#) [ACPI\\_PDC\\_P\\_FFH](#) [ACPI\\_PDC\\_SMP\\_C1PT](#) [ACPI\\_PDC\\_SMP\\_C2C3](#) [ACPI\\_PDC\\_SMP\\_P\\_HWCOORD](#) [ACPI\\_PDC\\_SMP\\_P\\_SWCOORD](#) [ACPI\\_PDC\\_T\\_FFH](#) [ALLOC\\_SPLIT\\_PTLOCKS](#) [ALTERNATIVE\(oldinstr,newinstr,feature\)](#) [ALTERNATIVE\\_2\(oldinstr,newinstr1,feature1,newinstr2,feature2\)](#) [ALTINSTR\\_ENTRY\(feature,number\)](#) [ALTINSTR\\_REPLACEMENT\(newinstr,feature,number\)](#) [APIC\\_ALL\\_CPUS](#) [APIC\\_BASE](#) [APIC\\_BASE\\_MSR](#) [APIC\\_DEBUG](#) [APIC\\_DFR](#) [APIC\\_EOI](#) [APIC\\_EOI\\_ACK](#) [APIC\\_ICR](#) [APIC\\_ID](#) [APIC\\_LDR](#) [APIC\\_LVR](#) [APIC\\_XAPIC\(x\)](#) [ARCH\\_HAS\\_IOREMAP\\_WC](#) [ASM\\_CLAC](#) [ASM\\_NOP3](#) [ASM\\_OUTPUT2\(a,...\)](#) [ASM\\_STAC](#) [ASM\\_X86\\_CMPXCHG\\_H](#) [ATOMIC\\_INIT\(i\)](#) [AT\\_VECTOR\\_SIZE](#) [AT\\_VECTOR\\_SIZE\\_ARCH](#) [AT\\_VECTOR\\_SIZE\\_BASE](#) [BAD\\_APICID](#) [BASE\\_PREFETCH](#) [BITMAP\\_LAST\\_WORD\\_MASK\(nbits\)](#) [BITOP\\_ADDR\(x\)](#) [BITOP\\_LE\\_SWIZZLE](#) [BITS\\_PER\\_BYTE](#) [BITS\\_PER\\_LONG](#) [BITS\\_TO\\_LONGS\(nr\)](#) [BUG\(\)](#) [BUGFLAG\\_WARNING](#) [BUG\\_ON\(condition\)](#) [BUILDIO\(bw1,bw,type\)](#) [BUILD\\_BUG\\_ON\(condition\)](#) [BUILD\\_BUG\\_ON\\_INVALID\(e\)](#) [CAP\\_BOP\\_ALL\(c,a,b,OP\)](#) [CAP\\_CHOWN](#) [CAP\\_DAC\\_OVERRIDE](#) [CAP\\_DAC\\_READ\\_SEARCH](#) [CAP\\_FOR\\_EACH\\_U32\(capi\)](#) [CAP\\_FOWNER](#) [CAP\\_FSETID](#) [CAP\\_FS\\_MASK\\_B0](#) [CAP\\_FS\\_MASK\\_B1](#) [CAP\\_FS\\_SET](#) [CAP\\_LINUX\\_IMMUTABLE](#) [CAP\\_MAC\\_OVERRIDE](#) [CAP\\_MKNOD](#) [CAP\\_NFSD\\_SET](#) [CAP\\_SYS\\_RESOURCE](#) [CAP\\_TO\\_INDEX\(x\)](#) [CAP\\_TO\\_MASK\(x\)](#) [CAP\\_UOP\\_ALL\(c,a,OP\)](#) [CLEARPAGEFLAG\(uname,lname\)](#) [CONFIG\\_64BIT](#) [CONFIG\\_ACPI](#) [CONFIG\\_ACPI\\_NUMA](#) [CONFIG\\_ACPI\\_SLEEP](#) [CONFIG\\_AIO](#) [CONFIG\\_AMD\\_IOMMU](#) [CONFIG\\_ARCH\\_CLOCKSOURCE\\_DATA](#) [CONFIG\\_ARCH\\_DMA\\_ADDR\\_T\\_64BIT](#) [CONFIG\\_ARCH\\_ENABLE\\_SPLIT\\_PMD\\_PTLOCK](#) [CONFIG\\_ARCH\\_HAS\\_CACHE\\_LINE\\_SIZE](#) [CONFIG\\_ARCH\\_HAS\\_CPU\\_AUTOPROBE](#) [CONFIG\\_ARCH\\_SUPPORTS\\_INT128](#) [CONFIG\\_ARCH\\_SUPPORTS\\_OPTIMIZED\\_INLINING](#) [CONFIG\\_ARCH\\_SUPPORTS\\_UPROBES](#) [CONFIG\\_ARCH\\_USES\\_PG\\_UNCACHED](#) [CONFIG\\_ARCH\\_USE\\_BUILTIN\\_BSWAP](#) [CONFIG\\_ARCH\\_USE\\_CMPXCHG\\_LOCKREF](#) [CONFIG\\_ASSOCIATIVE\\_ARRAY](#) [CONFIG\\_AUDIT](#) [CONFIG\\_AUDITSYSCALL](#) [CONFIG\\_BASE\\_SMALL](#) [CONFIG\\_BINARY\\_PRINTF](#) [CONFIG\\_BLK\\_CGROUP](#) [CONFIG\\_BLK\\_DEV\\_INTEGRITY](#) [CONFIG\\_BLK\\_DEV\\_IO\\_TRACE](#) [CONFIG\\_BLOCK](#) [CONFIG\\_BSD\\_PROCESS\\_ACCT](#) [CONFIG\\_BUG](#) [CONFIG\\_CC\\_STACKPROTECTOR](#) [CONFIG\\_CGROUPS](#) [CONFIG\\_CGROUP\\_CPUACCT](#) [CONFIG\\_CGROUP\\_DEVICE](#) [CONFIG\\_CGROUP\\_FREEZER](#) [CONFIG\\_CGROUP\\_PERF](#) [CONFIG\\_CGROUP\\_SCHED](#) [CONFIG\\_CLOCKSOURCE\\_WATCHDOG](#) [CONFIG\\_COMPAT](#) [CONFIG\\_CORE\\_DUMP\\_DEFAULT\\_ELF\\_HEADERS](#) [CONFIG\\_CPUSETS](#) [CONFIG\\_DEBUG\\_BUGVERBOSE](#) [CONFIG\\_DEBUG\\_RODATA](#) [CONFIG\\_DETECT\\_HUNG\\_TASK](#) [CONFIG\\_DEVTMPFS](#) [CONFIG\\_EARLY\\_PRINTK](#) [CONFIG\\_EPOLL](#) [CONFIG\\_FAIR\\_GROUP\\_SCHED](#) [CONFIG\\_FANOTIFY](#) [CONFIG\\_FILE\\_LOCKING](#) [CONFIG\\_FREEZER](#) [CONFIG\\_FSNOTIFY](#) [CONFIG\\_FS\\_POSIX\\_ACL](#) [CONFIG\\_FTRACE\\_MCOUNT\\_RECORD](#) [CONFIG\\_FUNCTION\\_GRAPH\\_TRACER](#) [CONFIG\\_FUTEX](#) [CONFIG\\_GENERIC\\_BUG](#) [CONFIG\\_GENERIC\\_BUG\\_RELATIVE\\_POINTERS](#) [CONFIG\\_GENERIC\\_FIND\\_FIRST\\_BIT](#) [CONFIG\\_GENERIC\\_PENDING\\_IRQ](#) [CONFIG\\_GENERIC\\_SMP\\_IDLE\\_THREAD](#) [CONFIG\\_HAS\\_IOPORT](#) [CONFIG\\_HAVE\\_ALIGNED\\_STRUCT\\_PAGE](#) [CONFIG\\_HAVE\\_ARCH\\_SOFT\\_DIRTY](#) [CONFIG\\_HAVE\\_CMPXCHG\\_DOUBLE](#) [CONFIG\\_HAVE\\_KVM](#) [CONFIG\\_HAVE\\_MEMBLOCK\\_NODE\\_MAP](#) [CONFIG\\_HAVE\\_MEMORY\\_PRESENT](#) [CONFIG\\_HAVE\\_SETUP\\_PER\\_CPU\\_AREA](#) [CONFIG\\_HAVE\\_UNSTABLE\\_SCHED\\_CLOCK](#) [CONFIG\\_HIBERNATION](#) [CONFIG\\_HIGH\\_RES\\_TIMERS](#) [CONFIG\\_HOTPLUG\\_CPU](#) [CONFIG\\_HUGETLBFS](#) [CONFIG\\_HUGETLB\\_PAGE](#) [CONFIG\\_HZ](#) [CONFIG\\_IA32\\_EMULATION](#)

The contents starts with a list of links to macro information further down the page; the first set of links is alphabetical to all macros that are declared, even if they are not used. The second set is to any macros that are actually used in pre-processing this file.

These are all linked to the macro details that looks like this, for example `BITMAP_LAST_WORD_MASK`:

Referenced Macros for: /U x

file:///Users/paulross/Documents/workspace/cpip/doc/sphinx/build/html/copy\_to\_html/cpu/cpu.c...

### BITMAP\_LAST\_WORD\_MASK [References: 10] Defined? True

```
#define BITMAP_LAST_WORD_MASK(nbits) ( ((nbits) % BITS_PER_LONG) ? (1UL<<((nbits) % \
BITS_PER_LONG))-1 : -0UL )
```

defined @ [/Users/paulross/dev/linux/linux-3.13/include/linux/bitmap.h#150](#)

/	Users/	paulross/	dev/	linux/	linux-3.13/	include/	linux/	bitmap.h: 176-20 228-20 237-31 246-29 255-34 263-20
						kernel/	cpu.c: 653-47	

I depend on these macros:

<a href="#">BITMAP_LAST_WORD_MASK</a>	<a href="#">BITS_PER_LONG</a>
---------------------------------------	-------------------------------

These macros depend on me:

<a href="#">BITMAP_LAST_WORD_MASK</a>	<a href="#">CPU_MASK_LAST_WORD</a>	<a href="#">CPU_BITS_ALL</a>
		<a href="#">CPU_MASK_ALL</a>
	<a href="#">NODE_MASK_LAST_WORD</a>	<a href="#">NODE_MASK_ALL</a>

Each macro description has the following:

- The macro name followed by the reference count for the macro i.e. how many times the pre-processor was required to invoke the definition. This line ends with whether it is still defined at the end of preprocessing (True in this case).
- The macro definition (this is artificially wrapped for clarity).
- Following `defined @` is where the macro was defined and a link to the source file where the macro is defined.
- Then follows a table of locations that the macro was used. In this case it was referenced by `include/linux/bitmap.h` on line 176, column 20, then line 228, column 20 and so on. Each of these references is a link to the source file representation where the macro is used. NOTE: Where macros are defined in terms of other macros then this location will not necessarily have the literal macro name, it is implicit because of macro dependencies. For example if you look at the last entry `kernel/cpu.c` line 653, column 47 then you do not see `BITMAP_LAST_WORD_MASK`, instead you see `CPU_BITS_ALL` however `CPU_BITS_ALL` is defined in terms of `BITMAP_LAST_WORD_MASK`.
- After “I depend on these macros” is a table (a tree actually) of other macros (with links) that `BITMAP_LAST_WORD_MASK` depend on, in this case only one, `BITS_PER_LONG` as you can see in the definition.
- After “These macros depend on me” is another table (a tree) of other macros (with links) that depend on `BITMAP_LAST_WORD_MASK`.

## A Most Powerful Feature

CPIP’s knowledge about macros and its ability to generate linked documents provides an especially powerful feature for understanding macros.

## Some Real Examples

CPIPMain is a command line tool that you can invoke very much like your favorite pre-processor. CPIPMain produces a number of HTML pages and SVG files that make it easier to understand what is happening during preprocessing. This section shows some examples of the kind of thing that CPIP can do.

### From the Linux Kernel

Here is `CPIPMain.py` pre-processing the `cpu.c` file from the Linux Kernel.

### From the CPython Interpreter

Here is `CPIPMain.py` pre-processing the `dictobject.c` file which implements the Python 3.6.2 dictionary.



## Command Line Tools

CPIP has a number of tools run from the command line that can analyse source code. The main one is `CPIPMain.py`. On installation the command line tool `cpipmain` is created which just calls `main()` in `CPIPMain.py`.

## CPIPMain

`CPIPMain.py` acts very much like a normal pre-processor but, instead of writing out a Translation Unit as test it emits a host of HTML and SVG pages about each file to be pre-processed. Here are *Some Real Examples*.

## Usage

```
usage: CPIPMain.py [-h] [-c] [-d DUMP] [-g GLOB] [--heap] [-j JOBS] [-k]
                  [-l LOGLEVEL] [-o OUTPUT] [-p] [-r] [-t] [-G]
                  [-S PREDEFINES] [-C] [-D DEFINES] [-P PREINC] [-I INCUSR]
                  [-J INCSYS]
                  path
```

`CPIPMain.py` - Preprocess the **file** **or** the files **in** a directory.

Created by Paul Ross on 2011-07-10.

Copyright 2008-2017. All rights reserved.

Licensed under GPL 2.0

USAGE

positional arguments:

path Path to source **file** **or** directory.

optional arguments:

-h, --help show this help message **and** exit  
 -c Add conditionally included files to the plots.  
 [default: **False**]  
 -d DUMP, --dump DUMP Dump output, additive. Can be: C - Conditional  
 compilation graph. F - File names encountered **and**

```
their count. I - Include graph. M - Macro environment.
T - Token count. R - Macro dependencies as an input to
DOT. [default: []]
-g GLOB, --glob GLOB    Pattern match to use when processing directories.
                        [default: *.*]
--heap                  Profile memory usage. [default: False]
-j JOBS, --jobs JOBS    Max simultaneous processes when pre-processing
                        directories. Zero uses number of native CPUs [4]. 1
                        means no multiprocessing. [default: 0]
-k, --keep-going        Keep going. [default: False]
-l LOGLEVEL, --loglevel LOGLEVEL
                        Log Level (debug=10, info=20, warning=30, error=40,
                        critical=50) [default: 30]
-o OUTPUT, --output OUTPUT
                        Output directory. [default: out]
-p                      Ignore pragma statements. [default: False]
-r, --recursive          Recursively process directories. [default: False]
-t, --dot                Write an DOT include dependency table and execute DOT
                        on it to create a SVG file. [default: False]
-G                      Support GCC extensions. Currently only #include_next.
                        [default: False]
-S PREDEFINES, --predefine PREDEFINES
                        Add standard predefined macro definitions of the form
                        name<=definition>. They are introduced into the
                        environment before anything else. They can not be
                        redefined. __DATE__ and __TIME__ will be automatically
                        allocated in here. __FILE__ and __LINE__ are defined
                        dynamically. See ISO/IEC 9899:1999 (E) 6.10.8
                        Predefined macro names. [default: []]
-C, --CPP                Sys call 'cpp -dM' to extract and use platform
                        specific macros. These are inserted after -S option
                        and before the -D option. [default: False]
-D DEFINES, --define DEFINES
                        Add macro definitions of the form name<=definition>.
                        These are introduced into the environment before any
                        pre-include. [default: []]
-P PREINC, --pre PREINC
                        Add pre-include file path, this file precedes the
                        initial translation unit. [default: []]
-I INCUSR, --usr INCUSR
                        Add user include search path. [default: []]
-J INCSYS, --sys INCSYS
                        Add system include search path. [default: []]
```

---

**Note:** Multiprocessing: The pre-processor, and information derived from it, can only be run as a single process but writing individual source files can take advantage of multiple processes. As the latter constitutes the bulk of the time CPIPMain.py takes then using the -j option on multi-processor machines can save a lot of time.

---



## Options

Option	Description
<code>--version</code>	Show program's version number and exit
<code>-h, --help</code>	Show this help message and exit.
<code>-c</code>	Even if a file is conditionally included then add it to the plot. This is experimental so use it at your own risk! [default: False]
<code>-d DUMP,</code> <code>--dump=DUMP</code>	Dump various outputs to stdout (see below). This option can be repeated [default: []]
<code>-g GLOB,</code> <code>--glob=GLOB</code>	Pattern to use when searching directories (ignored for <code>#includes</code> ). [default: <code>*.*</code> ]
<code>--heap</code>	Profile memory usage (requires <code>guppy</code> to be installed). [default: False]
<code>-j JOBS,</code> <code>--jobs=JOBS</code>	Max processes when multiprocessing. Zero uses number of native CPUs [4]. Value of 1 disables multiprocessing. [default: 0]
<code>-k</code>	Keep going as far as sensible, for some definition of "sensible". [default: False]
<code>-l LOGLEVEL,</code> <code>--loglevel=LOGLEVEL</code>	Log Level (debug=10, info=20, warning=30, error=40, critical=50) [default: 30]
<code>-o OUTPUT,</code> <code>--output=OUTPUT</code>	Output directory [default: "out"]
<code>-p</code>	Ignore pragma statements. [default: False]
<code>-r</code>	Recursively processes directories. [default: False]
<code>-t, --dot</code>	Write an DOT include dependency file and execute DOT on it to create a SVG file. Requires GraphViz. [default: False]
<code>-C, --CPP</code>	Sys call <code>cpp -dM</code> to extract and use platform specific macros. These are inserted after <code>-S</code> option and before the <code>-D</code> option. [default: False]
<code>-G</code>	Support GCC extensions. Currently only <code>#include_next</code> . [default: False]
<code>-I INCUSR,</code> <code>--usr=INCUSR</code>	Add user include search path (additive). This option can be repeated [default: []]
<code>-J INCSYS,</code> <code>--sys=INCSYS</code>	Add system include search path (additive). This option can be repeated [default: []]
<code>-S PREDEFINES,</code> <code>--predefine=PREDEFINES</code>	Add standard predefined macro definitions of the form <code>name&lt;=defintion&gt;</code> . These are introduced into the environment before anything else. These macros can not be redefined. <code>__DATE__</code> and <code>__TIME__</code> will be automatically defined. This option can be repeated [default: []]
<code>-D DEFINES,</code> <code>--define=DEFINES</code>	Add macro definitions of the form <code>name&lt;=definition&gt;</code> . These are introduced into the environment before any pre-include. This option can be repeated [default: []]
<code>-P PREINC,</code> <code>--pre=PREINC</code>	Add a pre-include file, this will be included before any header. This option can be repeated [default: []]

The `-d` option can be repeated to generate multiple text outputs on stdout:

Output	Description
<code>-d C</code>	Conditional compilation graph.
<code>-d F</code>	File names encountered and their count.
<code>-d I</code>	Include graph.
<code>-d M</code>	Macro environment.
<code>-d T</code>	Token count.
<code>-d R</code>	Macro dependencies as an input to DOT.

Examples of these are shown below [Using -d Option](#).

## Arguments

One or more paths of file(s) to be preprocessed.

## Examples

Here is a simple example of processing the demo code that is in the PpLexer tutorial here: [Files to Pre-Process](#).

Here we set:

- `-l 20` sets logging to INFO
- `-o` sets the output to `../..demo/output_00/`
- `-C` is used to get the platform specific macros.
- `-J` is used to set a single system include as `../..demo/sys/`
- `-I` is used to set a single user include as `../..demo/usr/`

We are processing `../..demo/src/main.cpp` and stdout is something like this:

```
$ python3 CPIPMain.py -l 20 -C -o ../..demo/output_00/ -J ../..demo/sys/ -I ../..demo/
↪demo/usr/ ../..demo/src/main.cpp
2012-03-20 07:41:38,655 INFO      TU in HTML:
2012-03-20 07:41:38,655 INFO      ../..demo/output_00/main.cpp.html
2012-03-20 07:41:38,664 INFO      Processing TU done.
2012-03-20 07:41:38,665 INFO      Macro history to:
2012-03-20 07:41:38,665 INFO      ../..demo/output_00/main.cpp_macros.html
2012-03-20 07:41:38,668 INFO      Include graph (SVG) to:
2012-03-20 07:41:38,668 INFO      ../..demo/output_00/main.cpp.include.svg
2012-03-20 07:41:38,679 INFO      Writing include graph (TEXT) to:
2012-03-20 07:41:38,679 INFO      ../..demo/output_00/main.cpp.include.svg
2012-03-20 07:41:38,679 INFO      Writing include graph (DOT) to:
2012-03-20 07:41:38,679 INFO      ../..demo/output_00/main.cpp.include.svg
2012-03-20 07:41:38,679 INFO      Creating include Graph for DOT...
2012-03-20 07:41:38,692 INFO      dot returned 0
2012-03-20 07:41:38,693 INFO      Creating include Graph for DOT done.
2012-03-20 07:41:38,693 INFO      Conditional compilation graph in HTML:
2012-03-20 07:41:38,693 INFO      ../..demo/output_00/main.cpp.ccg.html
2012-03-20 07:41:38,698 INFO      Done: ../..demo/src/main.cpp
2012-03-20 07:41:38,698 INFO      ITU in HTML: ...main.cpp
2012-03-20 07:41:38,708 INFO      ITU in HTML: ...system.h
2012-03-20 07:41:38,711 INFO      ITU in HTML: ...user.h
2012-03-20 07:41:38,716 INFO      All done.
CPU time =      0.051 (S)
Bye, bye!
```

In the output directory will be the HTML and SVG results.

## Using `-d` Option

All these are using the following command where ? is replace with a letter:

```
$ python3 CPIPMain.py -d? -o ../..demo/output_00/ -J ../..demo/sys/ -I ../..demo/
↪usr/ ../..demo/src/main.cpp
```

Multiple outputs are obtained with, for example, `-dC -dF`

**-d C**

Conditional compilation graph:

```
----- Conditional Compilation Graph -----
#ifndef __USER_H__ /* True "../../demo/usr/user.h" 1 0 */
  #ifndef __SYSTEM_H__ /* True "../../demo/sys/system.h" 1 4 */
    #endif /* True "../../demo/sys/system.h" 6 13 */
  #endif /* True "../../demo/usr/user.h" 7 20 */
  #if defined(LANG_SUPPORT) && defined(FRENCH) /* True "../../demo/src/main.cpp" 5 69 */
  #elif defined(LANG_SUPPORT) && defined(AUSTRALIAN) /* False "../../demo/src/main.cpp"
↪ 7 110 */
  #else /* False "../../demo/src/main.cpp" 9 117 */
  #endif /* False "../../demo/src/main.cpp" 11 124 */
----- END Conditional Compilation Graph -----
```

**-d F**

Files encountered and how many times processed:

```
----- Count of files encountered -----
1  ../../demo/src/main.cpp
1  ../../demo/sys/system.h
1  ../../demo/usr/user.h
----- END Count of files encountered -----
```

**-d I**

The include graph:

```
----- Include Graph -----
../../demo/src/main.cpp [43, 21]: True "" ""
000002: #include ../../demo/usr/user.h
        ../../demo/usr/user.h [10, 6]: True "" ["user.h", 'CP=None', 'usr=../../
↪demo/usr/']"
        000004: #include ../../demo/sys/system.h
        ../../demo/sys/system.h [10, 6]: True "!def __USER_H__" ['<system.h>
↪', 'sys=../../demo/sys/']"
----- END Include Graph -----
```

**-d M**

The macro environment and history:

```
----- Macro Environment and History -----
Macro Environment:
#define FRENCH /* ../../demo/usr/user.h#5 Ref: 1 True */
#define LANG_SUPPORT /* ../../demo/sys/system.h#4 Ref: 2 True */
#define __SYSTEM_H__ /* ../../demo/sys/system.h#2 Ref: 0 True */
#define __USER_H__ /* ../../demo/usr/user.h#2 Ref: 0 True */

Macro History (referenced macros only):
In scope:
```

```
#define FRENCH /* ../../demo/usr/user.h#5 Ref: 1 True */
    ../../demo/src/main.cpp 5 38
#define LANG_SUPPORT /* ../../demo/sys/system.h#4 Ref: 2 True */
    ../../demo/src/main.cpp 5 13
    ../../demo/src/main.cpp 7 15
----- END Macro Environment and History -----
```

## -d T

The token count:

```
----- Token count -----
    0 header-name
    8 identifier
    1 pp-number
    0 character-literal
    1 string-literal
   11 preprocessing-op-or-punc
    0 non-whitespace
   11 whitespace
    0 concat
   32 TOTAL
----- END Token count -----
```

## Performance

As CPIPMain.py/cpipmain is written in Python it is pretty slow, far slower than gcc or clang. Internally in cpip there are some fairly aggressive integrity checks such as `_assertDefineMapIntegrity()` in `cpip.core.MacroEnv.MacroEnv`. These integrity checks are invoked as asserts, for example:

```
assert(self._assertDefineMapIntegrity())
```

So that they can be turned off by using optimisation level 1.

For CPIPMain.py:

```
$ python3 -O CPIPMain.py ...
```

And cpipmain:

```
$ PYTHONOPTIMIZE=1 cpipmain ...
```

This optimisation can reduce the execution time by around 30%.

Various Tutorials on how to use CPIP.

Contents:

### PpLexer Tutorial

The PpLexer module represents the user side view of pre-processing. This tutorial shows you how to get going.

#### Setting Up

##### Files to Pre-Process

First let's get some demonstration code to pre-process. You can find this at *cpip/demo/* and the directory structure looks like this:

```
\---demo/  
|   cpip.py  
|  
\---proj/  
    +---src/  
    |       main.cpp  
    |  
    +---sys/  
    |       system.h  
    |  
    \---usr/  
        user.h
```

In `proj/` is some source code that includes files from `usr/` and `sys/`. This tutorial will take you through writing `cpip.py` to use PpLexer to pre-process them.

First lets have a look at the source code that we are preprocessing. It is a pretty trivial variation of a common them, but beware, pre-processing directives abound!

The file `demo/proj/src/main.cpp` looks like this:

```
#include "user.h"

int main(char **argv, int argc)
{
    #if defined(LANG_SUPPORT) && defined(FRENCH)
        printf("Bonjour tout le monde\n");
    #elif defined(LANG_SUPPORT) && defined(AUSTRALIAN)
        printf("Wotcha\n");
    #else
        printf("Hello world\n");
    #endif
    return 1;
}
```

That includes a file `user.h` that can be found at `demo/proj/usr/user.h`:

```
#ifndef __USER_H__
#define __USER_H__

#include <system.h>
#define FRENCH

#endif // __USER_H__
```

In turn that includes a file `system.h` that can be found at `demo/proj/sys/system.h`:

```
#ifndef __SYSTEM_H__
#define __SYSTEM_H__

#define LANG_SUPPORT

#endif // __SYSTEM_H__
```

Clearly since the system is mandating language support and the user is specifying French as their language of choice then you would not expect this to write out “Hello World”, or would you?

Well you are in the hands of the pre-processor and that is what CPIP knows all about. First we need to create a PpLexer.

## Creating a PpLexer

This is the template that we will use for the tutorial, it just takes a single argument from the command line `sys.argv[1]`:

```
1 import sys
2
3 def main():
4     print('Processing:', sys.argv[1])
5     # Your code here
6
7 if __name__ == "__main__":
8     main()
```

Of course this doesn't do much yet, invoking it just gives:

```
python cpip.py proj/src/main.cpp
Processing: proj/src/main.cpp
```

We now need to import and create a `PpLexer.PpLexer` object, and this takes at least two arguments; firstly the file to pre-process, the secondly an *include handler*. The latter is needed because the C/C++ standards do not specify how an `#include` directive is to be processed as that is an implementation issue. So we need to provide a defined implementation of something that can find `#include'd` files.

CPIP provides several such implementations in the module `IncludeHandler` and the one that does what, I guess, most developers expect from a pre-processor is `IncludeHandler.CppIncludeStdOs`. This class takes at least two arguments; a list of search paths to the user include directories and a list of search paths to the system include directories. With this we can construct a `PpLexer` object so our code now looks like this:

```
import sys
from cpip.core import PpLexer, IncludeHandler

def main():
    print('Processing:', sys.argv[1])
    myH = IncludeHandler.CppIncludeStdOs(
        theUsrDirs=['proj/usr'],
        theSysDirs=['proj/sys'],
    )
    myLex = PpLexer.PpLexer(sys.argv[1], myH)

if __name__ == "__main__":
    main()
```

This still doesn't do much yet, invoking it just gives:

```
python cpip.py proj/src/main.cpp
Processing: proj/src/main.cpp
```

But, in the absence of error, shows that we can construct a `PpLexer`.

## Put the PpLexer to Work

To get `PpLexer` to do something, we need to make the call to `PpLexer.PpTokens()`. This function is a generator of preprocessing *tokens*.

Lets just print them out with this code:

```
import sys
from cpip.core import PpLexer, IncludeHandler

def main():
    print('Processing:', sys.argv[1])
    myH = IncludeHandler.CppIncludeStdOs(
        theUsrDirs=['proj/usr'],
        theSysDirs=['proj/sys'],
    )
    myLex = PpLexer.PpLexer(sys.argv[1], myH)
    for tok in myLex.ppTokens():
        print(tok)
```

```
if __name__ == "__main__":
    main()
```

Invoking it now gives:

```
$ python cpip.py proj/src/main.cpp
Processing: proj/src/main.cpp
PpToken(t="\n", tt=whitespace, line=False, prev=False, ?=False)
...
PpToken(t="int", tt=identifier, line=True, prev=False, ?=False)
PpToken(t=" ", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="main", tt=identifier, line=True, prev=False, ?=False)
PpToken(t="(", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t="char", tt=identifier, line=True, prev=False, ?=False)
PpToken(t=" ", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="*", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t="*", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t="argv", tt=identifier, line=True, prev=False, ?=False)
PpToken(t=",", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t=" ", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="int", tt=identifier, line=True, prev=False, ?=False)
PpToken(t=" ", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="argc", tt=identifier, line=True, prev=False, ?=False)
PpToken(t=")", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t="\n", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="{", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t="\n", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="\n", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="printf", tt=identifier, line=True, prev=False, ?=False)
PpToken(t="(", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t="\"Bonjour tout le monde\n\"", tt=string-literal, line=False, prev=False, ?
↪=False)
PpToken(t=")", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t=";", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t="\n", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="\n", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="return", tt=identifier, line=True, prev=False, ?=False)
PpToken(t=" ", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="1", tt=pp-number, line=False, prev=False, ?=False)
PpToken(t=";", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t="\n", tt=whitespace, line=False, prev=False, ?=False)
PpToken(t="}", tt=preprocessing-op-or-punc, line=False, prev=False, ?=False)
PpToken(t="\n", tt=whitespace, line=False, prev=False, ?=False)
```

The PpLexer is yielding PpToken objects that are interesting in themselves because they not only have content but the type of content (whitespace, punctuation, literals etc.). A simplification is to change the code to print out the token *value* by changing a line in the code from:

```
print tok
```

To:

```
print tok.t
```

To give:

```
Processing: proj/src/main.cpp
```



```

int  main ( char  * * argv ,  int  argc )
{

printf ( "Bonjour tout le monde\n" ) ;

return  1 ;
}

```

It is definitely pre-processed and although the output is correct it is rather verbose because of all the whitespace generated by the pre-processing (newlines are always the consequence of pre-processing directives).

We can clean this whitespace up very simply by invoking `PpTokens.ppTokens()` with a suitable argument to reduce spurious whitespace thus: `myLex.ppTokens(minWs=True)`. This minimises the whitespace runs to a single space or newline. Our code now looks like this:

```

import sys
from cpip.core import PpLexer, IncludeHandler

def main():
    print('Processing:', sys.argv[1])
    myH = IncludeHandler.CppIncludeStdOs(
        theUsrDirs=['proj/usr',],
        theSysDirs=['proj/sys',],
    )
    myLex = PpLexer.PpLexer(sys.argv[1], myH)
    for tok in myLex.ppTokens(minWs=True):
        print(tok.t, end=' ')

if __name__ == "__main__":
    main()

```

Invoking it now gives:

```

Processing: proj/src/main.cpp

int  main ( char  * * argv ,  int  argc )
{
printf ( "Bonjour tout le monde\n" ) ;
return  1 ;
}

```

This is exactly the result that one would expect from pre-processing the original source code.

## And now for something Completely Different

So far, so boring because any pre-processor can do the same, PpLexer can do far more than this. PpLexer keeps track of a large amount of significant pre-processing information and that is available to you through the PpLexer APIs.

For a moment lets remove the `minWs=True` from `myLex.ppTokens()` so that we can inspect the state of the `PpLexer` at every token (rather than skipping whitespace tokens that might represent pre-processing directives).

## File Include Stack

Changing the code to this shows the `include` file hierarchy every step of the way:

```
for tok in myLex.ppTokens():
    print myLex.fileStack
```

Gives the following output:

```
$ python cpip.py proj/src/main.cpp
Processing: proj/src/main.cpp
['proj/src/main.cpp', 'proj/usr/user.h']
['proj/src/main.cpp', 'proj/usr/user.h']
['proj/src/main.cpp', 'proj/usr/user.h', 'proj/sys/system.h']
['proj/src/main.cpp', 'proj/usr/user.h', 'proj/sys/system.h']
['proj/src/main.cpp', 'proj/usr/user.h', 'proj/sys/system.h']
['proj/src/main.cpp', 'proj/usr/user.h', 'proj/sys/system.h']
['proj/src/main.cpp', 'proj/usr/user.h']
['proj/src/main.cpp', 'proj/usr/user.h']
['proj/src/main.cpp', 'proj/usr/user.h']
['proj/src/main.cpp']
...
```

## Conditional State

Changing the code to this:

```
for tok in myLex.ppTokens(condLevel=1):
    print myLex.condState
```

Produces this output:

```
Processing: proj/src/main.cpp
(True, '')
...
(True, '')
(True, 'defined(LANG_SUPPORT) && defined(FRENCH)')
(True, 'defined(LANG_SUPPORT) && defined(FRENCH)')
(True, 'defined(LANG_SUPPORT) && defined(FRENCH)')
(True, 'defined(LANG_SUPPORT) && defined(FRENCH)')
(True, 'defined(LANG_SUPPORT) && defined(FRENCH)')
(True, 'defined(LANG_SUPPORT) && defined(FRENCH)')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && defined(LANG_SUPPORT) &&
↳ defined(AUSTRALIAN))')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && defined(LANG_SUPPORT) &&
↳ defined(AUSTRALIAN))')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && defined(LANG_SUPPORT) &&
↳ defined(AUSTRALIAN))')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && defined(LANG_SUPPORT) &&
↳ defined(AUSTRALIAN))')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && defined(LANG_SUPPORT) &&
↳ defined(AUSTRALIAN))')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && defined(LANG_SUPPORT) &&
↳ defined(AUSTRALIAN))')
```

```
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && !(defined(LANG_SUPPORT) &&
↪defined(AUSTRALIAN)))')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && !(defined(LANG_SUPPORT) &&
↪defined(AUSTRALIAN)))')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && !(defined(LANG_SUPPORT) &&
↪defined(AUSTRALIAN)))')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && !(defined(LANG_SUPPORT) &&
↪defined(AUSTRALIAN)))')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && !(defined(LANG_SUPPORT) &&
↪defined(AUSTRALIAN)))')
(False, '(! (defined(LANG_SUPPORT) && defined(FRENCH)) && !(defined(LANG_SUPPORT) &&
↪defined(AUSTRALIAN)))')
(True, '')
...
(True, '')
```

## State of the PpLexer After Pre-processing

A more common use case is to query the PpLexer after processing the file. The following code example will:

- Capture all tokens as a Translation Unit and write it out with minimal whitespace [lines 11-16].
- Print out a text representation of the file include graph [lines 18-21].
- Print out a text representation of the conditional compilation graph [lines 23-26].
- Print out a text representation of the macro environment as it exists at the end of processing the Translation Unit [lines 28-31].
- Print out a text representation of the macro history for all macros, whether referenced or not, as it exists at the end of processing the Translation Unit [lines 33-36].

Here is the code, named `cpip_07.py`:

```
1 import sys
2 from cpip.core import PpLexer, IncludeHandler
3
4 def main():
5     print('Processing:', sys.argv[1])
6     myH = IncludeHandler.CppIncludeStdOs(
7         theUsrDirs=['proj/usr',],
8         theSysDirs=['proj/sys',],
9     )
10    myLex = PpLexer.PpLexer(sys.argv[1], myH)
11    tu = ''.join(tok.t for tok in myLex.ppTokens(minWs=True))
12
13    print()
14    print(' Translation Unit '.center(75, '='))
15    print(tu)
16    print(' Translation Unit END '.center(75, '='))
17
18    print()
19    print(' File Include Graph '.center(75, '='))
20    print(myLex.fileIncludeGraphRoot)
21    print(' File Include Graph END '.center(75, '='))
22
23    print()
24    print(' Conditional Compilation Graph '.center(75, '='))
```

```

25     print(myLex.condCompGraph)
26     print(' Conditional Compilation Graph END '.center(75, '='))
27
28     print()
29     print(' Macro Environment '.center(75, '='))
30     print(myLex.macroEnvironment)
31     print(' Macro Environment END '.center(75, '='))
32
33     print()
34     print(' Macro History '.center(75, '='))
35     print(myLex.macroEnvironment.macroHistory(incEnv=False, onlyRef=False))
36     print(' Macro History END '.center(75, '='))
37
38 if __name__ == "__main__":
39     main()

```

Invoking this code thus:

```
$ python3 cpip_07.py ../src/main.cpp
```

Gives this output:

```

Processing: ../src/main.cpp
===== Translation Unit =====

int main(char **argv, int argc)
{
printf("Bonjour tout le monde\n");
return 1;
}

===== Translation Unit END =====

===== File Include Graph =====
../src/main.cpp [43, 21]: True "" ""
000002: #include ../usr/user.h
      ../usr/user.h [10, 6]: True "" ["user.h", 'CP=None', 'usr=../usr']
000004: #include ../sys/system.h
      ../sys/system.h [10, 6]: True "def __USER_H__" ["<system.h>",
↪ 'sys=../sys']
===== File Include Graph END =====

===== Conditional Compilation Graph =====
#ifndef __USER_H__ /* True "../usr/user.h" 1 0 */
  #ifndef __SYSTEM_H__ /* True "../sys/system.h" 1 4 */
    #endif /* True "../sys/system.h" 6 13 */
  #endif /* True "../usr/user.h" 7 20 */
  #if defined(LANG_SUPPORT) && defined(FRENCH) /* True "../src/main.cpp" 5 69 */
  #elif defined(LANG_SUPPORT) && defined(AUSTRALIAN) /* False "../src/main.cpp" 7 110 */
  #else /* False "../src/main.cpp" 9 117 */
  #endif /* False "../src/main.cpp" 11 124 */
===== Conditional Compilation Graph END =====

===== Macro Environment =====
#define FRENCH /* ../usr/user.h#5 Ref: 1 True */
#define LANG_SUPPORT /* ../sys/system.h#4 Ref: 2 True */
#define __SYSTEM_H__ /* ../sys/system.h#2 Ref: 0 True */
#define __USER_H__ /* ../usr/user.h#2 Ref: 0 True */

```

```

===== Macro Environment END =====

===== Macro History =====
Macro History (all macros):
In scope:
#define FRENCH /* ../usr/user.h#5 Ref: 1 True */
    ../src/main.cpp 5 38
#define LANG_SUPPORT /* ../sys/system.h#4 Ref: 2 True */
    ../src/main.cpp 5 13
    ../src/main.cpp 7 15
#define __SYSTEM_H__ /* ../sys/system.h#2 Ref: 0 True */
#define __USER_H__ /* ../usr/user.h#2 Ref: 0 True */
===== Macro History END =====

```

This is simple to the point of crude as the PpLexer supplies a far richer data seam than just text.

File Include Graph interface is described here: [FileIncludeGraph Tutorial](#)

## Summary

There are several ways that you can inspect pre-processing with PpLexer:

- Supplying arguments to `PpLexer.ppTokens()` with arguments such as `minWs` or `incCond`.
- Accessing the state of each token as it is generated such as `tok.tt` or `tok.isCond`.
- Accessing the state of PpLexer as each token as it is generated or once all tokens have been generated such as `PpLexer.condState`.
- Creating PpLexer with a user specified behaviour. This is the subject of the next section.

## Advanced PpLexer Construction

The PpLexer constructor allows you to change the behaviour of pre-processing in a number of ways, effectively these are hooks into pre-processing that can:

- Varying how `#include'd` files are inserted into the Translation Unit.
- Pre-including header files.
- Changing the behaviour of PpLexer in unusual circumstances (errors etc.).
- Handling `#pragma` statements, in this way various compilers can be imitated.

### Include Handler

When an `#include` directive is encountered a compliant implementation is required to search for and insert into the Translation Unit the content referenced by the payload of the `#include` directive.

The standard does not specify *how* this should be accomplished. In CPIP the *how* is achieved by an implementation of an `cpip.core.IncludeHandler`.

### An Aside

It is entirely acceptable within the standard to have an `#include` system that does not rely on a file system at all. Perhaps it might rely on a database like this:

```
#include "SQL:spam.eggs#1284"
```

An include handler could take that payload and recover the content from some database rather than the local file system.

Or, more prosaically, an include mechanism such as this:

```
#include "http://some.url.org/spam/eggs#1284"
```

That leads to a fairly obvious way of managing that `#include` payload.

## Implementation

If you want to create a new include mechanism then you should sub-class the base class `cpip.core.IncludeHandler.CppIncludeStd` [reference documentation: [IncludeHandler](#)].

Sub-classing this requires implementing the following methods :

- **def initialTu(self, theTuIdentifier):** Given an Translation Unit Identifier this should return a class `FilePathOrigin` or `None` for the initial translation unit. As a precaution this should include code to check that the stack of current places is empty. For example:

```
if len(self._cpStack) != 0:
    raise ExceptionCppInclude('setTu() with CP stack: %s' % self._cpStack)
```

- **def \_searchFile(self, theCharSeq, theSearchPath):** Given an `HcharSeq/Qcharseq` and a searchpath this should return a class `FilePathOrigin` or `None`.

As examples there are a couple of reference implementations in `cpip.core.IncludeHandler`:

- `cpip.core.IncludeHandler.CppIncludeStdOs` - An implementation that behaves as most developers think the `#include` mechanism works.
- `cpip.core.IncludeHandler.CppIncludeStringIO` - An implementation that recovers content from a dictionary of in-memory files. This is used a lot within CPIP for unit testing.

## Pre-includes

The `PpLexer` can be supplied with an ordered list of file like objects that are pre-include files. These are processed in order before the ITU is processed. Macro redefinition rules apply.

For example `CPIPMain.py` can take a list of user defined macros on the command line. It then creates a list with a single pre-include file thus:

```
import io
from cpip.core import PpLexer

# defines is a list thus:
# ['spam(x)=x+4', 'eggs',]

myStr = '\n'.join(['#define '+' '.join(d.split('=')) for d in defines])+'\n'
myPreIncFiles = [io.StringIO(myStr), ]
# Create other constructor information here...
myLexer = PpLexer.PpLexer(
    anItu, # File to pre-process
    myIncH, # Include handler
```

```

        preIncFiles=myPreIncFiles,
    )

```

## Diagnostic

You can pass in to `PpLexer` a diagnostic object, this controls how the lexer responds to various conditions such as warning error etc. The default is for the lexer to create a `CppDiagnostic.PreprocessDiagnosticStd`.

If you want to create your own then sub-class the `cpip.core.CppDiagnostic.PreprocessDiagnosticStd` class in the module `cpip.ref.CppDiagnostic`.

Sub-classing `PreprocessDiagnosticStd` allows you to override any of the following that might be called by the `PpLexer`:

- `def undefined(self, msg, theLoc=None) :` Reports when an ‘undefined’ event happens.
- `def partialTokenStream(self, msg, theLoc=None) :` Reports when an partial token stream exists (e.g. an unclosed comment).
- `def implementationDefined(self, msg, theLoc=None) :` Reports when an ‘implementation defined’ event happens.
- `def error(self, msg, theLoc=None) :` Reports when an error event happens.
- `def warning(self, msg, theLoc=None) :` Reports when an warning event happens.
- `def handleUnclosedComment(self, msg, theLoc=None) :` Reports when an unclosed comment is seen at EOF.
- `def unspecified(self, msg, theLoc=None) :` Reports when unspecified behaviour is happening, For example order of evaluation of ‘#’ and ‘##’.
- `def debug(self, msg, theLoc=None) :` Reports a debug message.

There are a couple of implementations in the `CppDiagnostic` module that may be of interest:

- `cpip.core.CppDiagnostic.PreprocessDiagnosticKeepGoing`: Sub-class that does not raise exceptions.
- `cpip.core.CppDiagnostic.PreprocessDiagnosticRaiseOnError`: Sub-class that raises an exception on a `#error` directive.

## Pragma

You can pass in a specialised handler for `#pragma` statements [default: `None`]. This shall sub-class `cpip.core.PragmaHandler.PragmaHandlerABC` and can implement:

- The boolean attribute `replaceTokens` is to be implemented. If `True` then the tokens following the `#pragma` statement will be be macro replaced by the `PpLexer` using the current macro environment before being passed to this pragma handler.
- A method `def pragma(self, theTokS) :` that takes a non-zero length list of `PpTokens` the last of which will be a newline token. Any token this method returns will be yielded as part of the Translation Unit (and thus subject to macro replacement for example).

Have a look at the core module `cpip.core.PragmaHandler` for some example implementations.

## FileIncludeGraph Tutorial

The PpLexer module collects the file *include graph*. This tutorial shows you how to use it for your own ends.

### Creating a FileIncludeGraph

A FileIncludeGraph object is one of the artifacts produced by a PpLexer [see the tutorial here: [PpLexer Tutorial](#)].

Once the PpLexer has processed the Translation Unit it has and attribute fileIncludeGraphRoot which is an instance of the class FileIncludeGraph.FileIncludeGraphRoot.

Here is the code to create a file include graph:

```
1 import sys
2 from cpip.core import PpLexer
3 from cpip.core import IncludeHandler
4
5 def main():
6     print('Processing:', sys.argv[1])
7     myH = IncludeHandler.CppIncludeStdOs(
8         theUsrDirs=['../usr',],
9         theSysDirs=['../sys',],
10    )
11     myLex = PpLexer.PpLexer(sys.argv[1], myH)
12     tu = ''.join(tok.t for tok in myLex.ppTokens(minWs=True))
13     print(repr(myLex.fileIncludeGraphRoot))
14
15 if __name__ == "__main__":
16     main()
```

Invoking this code thus (in the manner of the [PpLexer Tutorial](#)):

```
$ python3 cpip_08.py ../src/main.cpp
```

Gives this output:

```
Processing: ../src/main.cpp
<cpip.core.FileIncludeGraph.FileIncludeGraphRoot object at 0x100753790>
```

### FileIncludeGraph Structure

The structure is a tree with each node being an included file, the root being the *Initial Translation Unit* i.e. the file being pre-processed. Source code order is ‘left-to-right’ and depth is the degree of #include statements.

The class FileIncludeGraph.FileIncludeGraphRoot has a fairly rich interface, reference documentation for the module is here: [FileIncludeGraph](#)

### A File Graph Visitor

The FileIncludeGraph.FileIncludeGraphRoot has a method def acceptVisitor(self, visitor): can accept a *visitor* object (that can inherit from FigVisitorBase) for traversing the graph. This takes the visitor object and calls visitor.visitGraph(self, theFigNode, theDepth, theLine) on that object where depth is the current depth in the graph as an integer and line the line that is a non-monotonic sibling node ordinal.



There are a number of visitor examples in the `FileIncludeGraph` test code. `CPIPMain` has a number of visitor implementations.

`visitGraph(self, theFigNode, theDepth, theLine)`

`theFigNode` is a `cpip.core.FileIncludeGraph.FileIncludeGraph` object. See [FileIncludeGraph](#)

## Example Visitor

Here we create a simple visitor [lines 6-9]. After processing the Translation Unit [line 18] we create a visitor and traverse the include graph [lines 19-20]. At each node in the graph the visitor merely prints out the file (node) name and the `findLogic` string i.e. how this file was found for inclusion [line 9].

```

1  import sys
2  from cpip.core import PpLexer
3  from cpip.core import IncludeHandler
4  from cpip.core import FileIncludeGraph
5
6  class Visitor(FileIncludeGraph.FigVisitorBase):
7
8      def visitGraph(self, theFigNode, theDepth, theLine):
9          print(theFigNode.fileName, theFigNode.findLogic)
10
11 def main():
12     print('Processing:', sys.argv[1])
13     myH = IncludeHandler.CppIncludeStdOs(
14         theUsrDirs=['../usr',],
15         theSysDirs=['../sys',],
16     )
17     myLex = PpLexer.PpLexer(sys.argv[1], myH)
18     tu = ''.join(tok.t for tok in myLex.ppTokens(minWs=True))
19     myVis = Visitor()
20     myLex.fileIncludeGraphRoot.acceptVisitor(myVis)
21
22 if __name__ == "__main__":
23     main()

```

Invoking this code thus (in the manner of the *PpLexer Tutorial*):

```
$ python3 cpip_09.py ../src/main.cpp
```

Gives this output:

```

Processing: ../src/main.cpp
../src/main.cpp
../usr/user.h ["user.h", 'CP=None', 'usr=../usr']
../sys/system.h ['<system.h>', 'sys=../sys']

```

For example, in line 3, this means that the file `../usr/user.h` was included with a `#include "user.h"` statement, first the “Current Place” (CP) was searched (unsuccessfully so result `None`), then the user include directories were searched and the file was found in the `../usr` directory.

## Creating a Bespoke Tree From a FileIncludeGraph

The use case here is, given a `FileIncludeGraph`, can I simply create a tree of objects of my own definition from the graph? An example would be creating a structure that makes it easy to plot an SVG graph. The class should sub-class

`cpip.core.FileIncludeGraph.FigVisitorTreeNodeBase`.

The solution is to create a `cpip.core.FileIncludeGraph.FigVisitorTree` object with a class definition for the node objects. This class definition must take in its constructor a file node (None for the root) and a line number.

Here is an example that is used to create a tree of file name and token counts. A class `MyVisitorTreeNode` is defined that on construction extracts file name and token count data from the file include graph node. The other requirement is to implement `finalise` at the the end of tree construction that updates the token count with those of the nodes children. Finally it supplies some string representation of itself.

The special code is on lines 40-43 where the `FileIncludeGraph.FigVisitorTree` visitor is created with a cls specification of `MyVisitorTreeNode`. The file include graph is then presented with the visitor (line 41). Finally a tree of `MyVisitorTreeNode` objects is retrieved with a call to `tree()`.

```

1  import sys
2  from cpip.core import PpLexer
3  from cpip.core import IncludeHandler
4  from cpip.core import FileIncludeGraph
5
6  class MyVisitorTreeNode(FileIncludeGraph.FigVisitorTreeNodeBase):
7      PAD = '  '
8      def __init__(self, theFig, theLineNum):
9          super(MyVisitorTreeNode, self).__init__(theLineNum)
10         if theFig is None:
11             self._name = None
12             self._t = 0
13         else:
14             self._name = theFig.fileName
15             self._t = theFig.numTokens
16
17     def finalise(self):
18         # Tot up tokens
19         for aChild in self._children:
20             aChild.finalise()
21             self._t += aChild._t
22
23     def __str__(self):
24         return self.retStr(0)
25
26     def retStr(self, d):
27         r = '%s%04d %s %d\n' % (self.PAD*d, self._lineNum, self._name, self._t)
28         for aC in self._children:
29             r += aC.retStr(d+1)
30         return r
31
32 def main():
33     print('Processing:', sys.argv[1])
34     myH = IncludeHandler.CppIncludeStdOs(
35         theUsrDirs=['../usr'],
36         theSysDirs=['../sys'],
37     )
38     myLex = PpLexer.PpLexer(sys.argv[1], myH)
39     tu = ''.join(tok.t for tok in myLex.ppTokens(minWs=True))
40     myVis = FileIncludeGraph.FigVisitorTree(MyVisitorTreeNode)
41     myLex.fileIncludeGraphRoot.acceptVisitor(myVis)
42     myTree = myVis.tree()
43     print(myTree)
44
45 if __name__ == "__main__":

```

46

```
main()
```

Invoking this so:

```
$ python3 cpip_10.py ../src/main.cpp
```

Gives this output:

```
Processing: ../src/main.cpp
-001 None 63
  -001 ../src/main.cpp 63
    0002 ../usr/user.h 20
      0004 ../sys/system.h 10
```

Further examples can be found in the code in `IncGraphSVGBase.py` and `IncGraphXML.py`



## CPIPMain

CPIPMain.py – Preprocess the file or the files in a directory.

**class** `cpip.CPIPMain.FigVisitorDot` (*lenPrefix=0*)

Simple visitor that collects parent/child links for plotting the graph with dot.

**visitGraph** (*theFigNode, theDepth, theLine*)

.

**class** `cpip.CPIPMain.FigVisitorLargestCommonPrefix`

Simple visitor that walks the tree and finds the largest common file name prefix.

**visitGraph** (*theFigNode, theDepth, theLine*)

Capture the file name.

**class** `cpip.CPIPMain.MainJobSpec` (*incHandler, preDefMacros, preIncFiles, diagnostic, pragmaHandler, keepGoing, conditionalLevel, dumpList, helpMap, includeDOT, cmdLine, gccExtensions*)

**cmdLine**

Alias for field number 10

**conditionalLevel**

Alias for field number 6

**diagnostic**

Alias for field number 3

**dumpList**

Alias for field number 7

**gccExtensions**

Alias for field number 11

**helpMap**

Alias for field number 8

**incHandler**

Alias for field number 0

**includeDOT**

Alias for field number 9

**keepGoing**

Alias for field number 5

**pragmaHandler**

Alias for field number 4

**preDefMacros**

Alias for field number 1

**preIncFiles**

Alias for field number 2

**class** cpip.CPIPMain.**PpProcessResult** (*ituPath, indexPath, tuIndexFileName, total\_files, total\_lines, total\_bytes*)

**indexPath**

Alias for field number 1

**ituPath**

Alias for field number 0

**total\_bytes**

Alias for field number 5

**total\_files**

Alias for field number 3

**total\_lines**

Alias for field number 4

**tuIndexFileName**

Alias for field number 2

**cpip.CPIPMain.main()**

Processes command line to preprocess a file or a directory.

**cpip.CPIPMain.preProcessFilesMP** (*dIn, dOut, jobSpec, glob, recursive, jobs*)

Multiprocessing code to preprocess directories. Returns a count of ITUs processed.

**cpip.CPIPMain.preprocessDirToOutput** (*inDir, outDir, jobSpec, globMatch, recursive, numJobs*)

Pre-process all the files in a directory. Returns a count of the TUs. This uses multiprocessing where possible. Any Exception (such as a KeyboardInterrupt) will terminate this function but write out an index of what has been achieved so far.

**cpip.CPIPMain.preprocessFileToOutput** (*ituPath, outDir, jobSpec*)

Preprocess a single file. May raise ExceptionCpip (or worse!). Returns a: PpProcessResult (ituPath, indexPath, tuIndexFileName(ituPath) total\_files, total\_lines, total\_bytes)

**cpip.CPIPMain.preprocessFileToOutputNoExcept** (*ituPath, \*args, \*\*kwargs*)

Preprocess a single file and catch all ExceptionCpip exceptions and log them.

**cpip.CPIPMain.retFileCountMap** (*theLexer*)

Visits the Lexers file include graph and returns a dict of: {file\_name : (inclusion\_count, line\_count, bytes\_count)}.

The `line_count`, `bytes_count` are obtained by reading the file.

`cpip.CPIPMain.retOptionMap (theOptParser, theOpts)`

Returns map of {`opt_name` : (value, help), ...} from the current options.

`cpip.CPIPMain.writeIndexHtml (theItuS, theOutDir, theJobSpec, time_start, total_files, total_lines, total_bytes)`

Writes the top level index.html page for a pre-processed file.

`theOutDir` - The output directory.

`theTuS` - The list of translation units processed.

`theCmdLine` - The command line as a string.

`theOptMap` is a map of {`opt_name` : (value, help), ...} from the command line options. TODO: This is fine but has too many levels of indent.

`cpip.CPIPMain.writeTuIndexHtml (theOutDir, theTuPath, theLexer, theFileCountMap, theTokenCntr, hasIncDot, macroHistoryIndexName)`

Write the index.html for a single TU.

***theOutDir*** The output directory to write to.

***theTuPath*** The path to the original ITU.

***theLexer*** The pre-processing Lexer that has pre-processed the ITU/TU.

***theFileCountMap*** dict of {`file_path` : data, ...} where data is things like inclusion count, lines, bytes and so on.

***theTokenCntr*** `cpip.core.PpTokenCount.PpTokenCount` containing the token counts.

***hasIncDot*** bool to emit graphviz .dot files.

***macroHistoryIndexName*** String of the filename of the macro history.

Returns: (total\_files, total\_lines, total\_bytes) as integers.

## CppCondGraphToHtml

Writes out the Cpp Conditional processing graph as HTML.

**class** `cpip.CppCondGraphToHtml.CcgVisitorToHtml (theHtmlStream)`

Writing CppCondGraph visitor object.

**visitPost** (*theCcgNode, theDepth*)

Post-traversal call with a CppCondGraphNode and the integer depth in the tree.

**visitPre** (*theCcgNode, theDepth*)

Pre-traversal call with a CppCondGraphNode and the integer depth in the tree.

`cpip.CppCondGraphToHtml.processCppCondGrphToHtml (theLex, theHtmlPath, theTitle, theIdxPath)`

Given the PpLexer write out the Cpp Cond Graph to the HTML file. `theLex` is a PpLexer. `theHtmlPath` is the file path of the output. `theTitle` is the page title. `theIdxPath` is the file name of the index page. `theTuIndexer` is a `TuIndexer.TuIndexer` object.

## FileStatus

Provides a command line tool for finding out information on files:

```
$ python3 src/cpip/FileStatus.py -r src/cpip/
Cmd: src/cpip/FileStatus.py -r src/cpip/
File                               SLOC      Size
↳MD5 Last modified
src/cpip/CPIPMMain.py              1072      44829
↳4dee8712b7d51f978689ef257cf1fd34 Wed Sep 27 08:57:00 2017
src/cpip/CppCondGraphToHtml.py     124       4862
↳4f0d5731ef6f3d47ec638f00e7646a9f Fri Sep 8 15:30:41 2017
src/cpip/DupeRelink.py              269      11795
↳914ed2149dce6584e6f3f55ec0e2b923 Wed Sep 27 11:35:32 2017
src/cpip/FileStatus.py             218       8015
↳6db0658622e82d32a9a9b4c8eb9e82e5 Thu Sep 28 11:13:40 2017
src/cpip/IncGraphSVG.py            1026     45049
↳7b82651dadd44eb4ed65d390f6c052df Fri Sep 8 15:30:41 2017
...
src/cpip/util/Tree.py              166       5719
↳cdb81dleaaaf6a1743e5182355f2e75bb Fri Sep 8 15:30:41 2017
src/cpip/util/XmlWrite.py           425      15114
↳48563685ace3ec0f6d734695cac17ede Tue Sep 12 15:38:55 2017
src/cpip/util/__init__.py           31       1161
↳208abac9edd9682f438945906a451473 Fri Sep 8 15:30:41 2017
Total [54]                        19475     789349
CPU time =      0.041 (S)
Bye, bye!
```

**class** cpip.FileStatus.**FileInfo** (*thePath*)

Holds information on a text file.

**count**

Files processed.

**size**

Size in bytes.

**sloc**

Lines in file.

**write** (*theS*=<*\_io.TextIOWrapper* name='<stdout>' mode='w' encoding='UTF-8'>, *incHash*=True)

Writes the number of lines and bytes (optionally MD5) to stream.

**writeHeader** (*theS*=<*\_io.TextIOWrapper* name='<stdout>' mode='w' encoding='UTF-8'>)

Writes header to stream.

**class** cpip.FileStatus.**FileInfoSet** (*thePath*, *glob*=None, *isRecursive*=False)

Contains information on a set of files.

**processDir** (*theDir*, *glob*, *isRecursive*)

Read a directory and return a map of {path : class FileInfo, ...}

**processPath** (*theP*, *glob*=None, *isRecursive*=False)

Process a file or directory.

**write** (*theS*=<*\_io.TextIOWrapper* name='<stdout>' mode='w' encoding='UTF-8'>)

Write summary to stream.

cpip.FileStatus.**main** ()

Prints out the status of files in a directory:

```
$ python ../src/cpip/FileStatus.py --help
Cmd: ../src/cpip/FileStatus.py --help
Usage: FileStatus.py [options] dir
```



Counts files and sizes.

Options:

```
--version          show program's version number and exit
-h, --help         show this help message and exit
-g GLOB, --glob=GLOB  Space separated list of file match patterns. [default:
                      *.py]
-l LOGLEVEL, --loglevel=LOGLEVEL
                      Log Level (debug=10, info=20, warning=30, error=40,
                      critical=50) [default: 30]
-r                Recursive. [default: False]
```

## IncGraphSVG

**class** `cpip.IncGraphSVG.SVGTreeNodeMain` (*theFig, theLineNum*)

This does most of the heavy lifting of plotting the include graph SVG. The challenges are plotting things in the ‘right’ order and with the ‘right’ JavaScript so that the DHTML does not look too hideous.

Basic principle here is that `plotInitialise()` writes static data. In our case just the pretty histogram pop-up (Ed. is this right??).

Then `SVGTreeNodeBase.plotToSVGStream()` is called - this is implemented in the base class.

Finally `plotFinalise()` is called - this overlays the DHTML text. This is a little tricky as our way of DHTML is to switch opacity on underlying objects the switching boundary being the overlying object (e.g. ‘?’). So `_all_` the underlying objects need to be written first so that the overlying objects are always ‘visible’ to trigger `onmouseover/onmouseout` on the underlying object.

**condComp**

A string of conditional tests.

**finalise()**

Finalisation this sets up all the bounding boxes of me and my children.

**findLogic**

The find logic as a string.

**plotFinalise** (*theSvg, theDatumL, theTpt*)

Finish the plot. In this case we write the text overlays.

**plotInitialise** (*theSvg, theDatumL, theTpt*)

Plot the histogram legend once only.

**tokenCounter**

This is the `PpTokenCount.PpTokenCount()` for me only.

**tokenCounterChildren**

This is the computed `PpTokenCount.PpTokenCount()` for all my descendents.

**tokenCounterTotal**

This is the computed `PpTokenCount.PpTokenCount()` me plus my descendents.

**writeAltTextAndMouseOverRect** (*theSvg, theId, theAltPt, theAltS, theTrigPt, theTrigRect*)

Composes and writes the (pop-up) alternate text. Also writes a trigger rectangle.

**writePreamble** (*theS*)

Write any preamble such as CSS or JavaScript. To be implemented by child classes.

## IncGraphSVGBase

Provides basic functionality to take the #include graph of a preprocessed file and plots it as a diagram in SVG.

### Event handlers for onmouseover/onmouseout

We would like to have more detailed information available to the user when they mouseover an object on the SVG image. After a lot of experiment the most cross browser way this is done by providing an event handler to switch the opacity of an element between 0 and 1. See `IncGraphSVG.writeAltTextAndMouseOverRect()`.

`cpip.IncGraphSVGBase.processIncGraphToSvg` (*theLex, theFilePath, theClass, tptPos, tptSweep*)  
Convert a Include graph from a PpLexer to SVG in theFilePath.

## IncGraphXML

Generates an XML file from an include graph.

This is implemented as a hierarchical visitor pattern. This could have be implemented as a non-hierarchical visitor pattern using less memory at the expense of more code.

`cpip.IncGraphXML.processIncGraphToXml` (*theLex, theFilePath*)  
Convert a Include graph from a PpLexer to SVG in theFilePath.

## IncList

## ItuToHtml

Converts an ITU to HTML.

**class** `cpip.ItuToHtml.ItuToHtml` (*theItu, theHtmlDir, keepGoing=False, macroRefMap=None, cpp-CondMap=None, ituToTuLineSet=None*)  
Converts an ITU to HTML and write it to the output directory.

## MacroHistoryHtml

Writes out a macro history in HTML.

Macros can be: Active - In scope at the end of processing a translation unit (one per identifier). Inactive - Not in scope at the end of processing a translation unit ( $\geq 0$  per identifier). And: Referenced - Have had some influence over the processing of the translation unit. Not Referenced - No influence over the processing of the translation unit.

Example test:

Macros with reference counts of zero are not that interesting so they are relegated to a page (`<file>_macros_noref.html`) that just describes their definition and where they where defined.

Macros `_with_` reference counts are presented on a page (`<file>_macros_ref.html`) with one section per macro. The section has: definition, where defined, [This macro depends on the following macros:], [Macros that depend on this macro:],

These two HTML pages are joined by a `<file>_macros.html` this lists (and links to) the identifiers in this order:

- Active, ref count >0
- Inactive, ref count >0
- Active, ref count =0
- Inactive, ref count =0

## Macro HTML IDs

This is identifier + ‘\_’ + n For any active macro the value of n is the number of previously defined macros. Current code is like this:

```
myUnDefIdxS, isDefined = myMacroMap[aMacroName]
# Write the undefined ones
for anIndex in myUnDefIdxS:
    myMacro = theEnv.getUnDefMacro(anIndex)
    startLetter = _writeTrMacro(theS, theHtmlPath, myMacro,
                                anIndex, startLetter, retVal)
# Now the defined one
if isDefined:
    myMacro = theEnv.macro(aMacroName)
    startLetter = _writeTrMacro(theS, theHtmlPath, myMacro,
                                len(myUnDefIdxS), startLetter, retVal)
```

`cpip.MacroHistoryHtml.processMacroHistoryToHtml` (*theLex, theHtmlPath, theItu, theIndex-Path*)

Write out the macro history from the PpLexer as HTML. Returns a map of: {identifier : [(fileId, lineNum, href\_name), ...], ...} which can be used by src->html generator for providing links to macro pages.

`cpip.MacroHistoryHtml.splitLine` (*theStr, splitLen=60, splitLenHard=80*)

Splits a long string into string that is a set of lines with continuation characters.

`cpip.MacroHistoryHtml.splitLineToList` (*sIn, splitLen=60, splitLenHard=80*)

Splits a long string into a list of lines. This tries to do it nicely at whitespaces but will force a split if necessary.

## TokenCss

CSS Support for ITU+TU files in HTML.

`cpip.TokenCss.writeCssForFile` (*theFile*)

Writes the CSS file into to the directory that the file is in.

`cpip.TokenCss.writeCssToDir` (*theDir*)

Writes the CSS file into to the directory.

## Tu2Html

Converts an initial translation unit to HTML.

TODO: For making anchors in the TU HTML that the conditional include graph can link to. If we put an <a name=”...” on every line most browsers can not handle that many. What we could do here is to keep a copy of the conditional include stack and for each token see if it has changed (like the file stack). If so that write a marker that the conditional graph can later link to.

`cpip.Tu2Html.processTuToHtml` (*theLex, theHtmlPath, theTitle, theCondLevel, theIdxPath, incItuAn-*  
*chors=True*)

Processes the PpLexer and writes the tokens to the HTML file.

***theHtmlPath*** The path to the HTML file to write.

***theTitle*** A string to go into the <title> element.

***theCondLevel*** The Conditional level to pass to theLex.ppTokens()

***theIdxPath*** Path to link back to the index page.

***incItuAnchors*** boolean, if True will write anchors for lines in the ITU that are in this TU. If True then setItu-  
LineNumbers returned is likely to be non-empty.

Returns a pair of (PpTokenCount.PpTokenCount(), set(int)) The latter is a set of integer line numbers in the ITU  
that are in the TU, these line numbers with have anchors in this HTML file of the form: <a name="%d" />.

## TuIndexer

Provides a means of linking to a translation unit to HTML.

**exception** `cpip.TuIndexer.ExceptionTuIndexer`  
Exception when handling PpLexer object.

**class** `cpip.TuIndexer.TuIndexer` (*tuFileName*)  
Provides a means of indexing into a TU html file.

**add** (*theTuIndex*)  
Adds an integer index to the list of markers, returns the href name.

**href** (*theTuIndex, isLB*)  
Returns an href string for the TuIndex. If isLB is true returns the nearest lower bound, otherwise the nearest  
upper bound.

## cpip.core

CPIP Core contains the core code for pre-processing. The architecture is described here: [CPIP Core Architecture](#).

## ConstantExpression

Handles the Python interpretation of a constant-expression. See *ISO/IEC 14882:1998(E)*

**class** `cpip.core.ConstantExpression.ConstantExpression` (*theTokTypeS*)  
Class that interpret a stream of pre-processing tokens (*cpip.core.PpToken.PpToken* objects) and evalu-  
ate it as a constant expression.

**evaluate** ()  
Evaluates the constant expression and returns 0 or 1.

**translateTokensToString** ()  
Returns a string to be evaluated as a constant-expression.

*ISO/IEC ISO/IEC 14882:1998(E) 16.1 Conditional inclusion sub-section 4 i.e. 16.1-4*

All remaining identifiers and keywords 137) , except for true and false, are replaced with the pp-number 0

**exception** `cpip.core.ConstantExpression.ExceptionConditionalExpression`

Exception when conditional expression e.g. ... ? ... : ... fails to evaluate.

**exception** `cpip.core.ConstantExpression.ExceptionConditionalExpressionInit`

Exception when initialising a ConstantExpression class.

**exception** `cpip.core.ConstantExpression.ExceptionConstantExpression`

Simple specialisation of an exception class for the ConstantExpression classes.

**exception** `cpip.core.ConstantExpression.ExceptionEvaluateExpression`

Exception when conditional expression e.g.  $1 < 2$  fails to evaluate.

## CppCond

Provides a state stack of booleans to facilitate conditional compilation as: *ISO/IEC 9899:1999(E) section 6.10.1* ('C') and *ISO/IEC 14882:1998(E) section 16.1* ('C++') [*cpp.cond*]

This does not interpret any semantics of either standard but instead provides a state class that callers that do interpret the language semantics can use.

In particular this provides state change operations that might be triggered by the following six pre-processing directives:

```
#if constant-expression new-line group opt
#ifdef identifier new-line group opt
#ifndef identifier new-line group opt
#elif constant-expression new-line group opt
#else new-line group opt
#endif new-line
```

In this module a single *CppCond* object has a stack of ConditionalState objects. The latter has both a boolean state and an 'explanation' of that state at any point in the translation. The latter is represented by a list of string representations of either constant-expression or identifier tokens.

The stack i.e. *CppCond* can also be queried for its net boolean state and its net 'explanation'.

Basic boolean stack operations:

Directive	Argument	Stack, s, boolean operation
-----	-----	-----
<code>#if</code>	<code>constant-expression</code>	<code>s.push(bool)</code>
<code>#ifdef</code>	<code>identifier</code>	<code>s.push(bool)</code>
<code>#ifndef</code>	<code>identifier</code>	<code>s.push(!bool)</code>
<code>#elif</code>	<code>constant-expression</code>	<code>s.pop(), s.push(bool)</code>
<code>#else</code>	N/A	<i>Either <code>s.push(!s.pop())</code> or <code>s.flip()</code></i>
<code>#endif</code>	N/A	<code>s.pop()</code>

Basic boolean 'explanation' string operations:

The '!' prefix is parameterised as `TOKEN_NEGATION` so that any subsequent processing can recognise '!!!' as '!' and '!!!!' as '!!':

Directive	Argument	Matrix, m, strings
-----	-----	-----
<code>#if</code>	<code>constant-expression</code>	<code>m.push(['%s' % tokens,])</code>
<code>#ifdef</code>	<code>identifier</code>	<code>m.push(['(defined %s)' % identifier])</code>
<code>#ifndef</code>	<code>identifier</code>	<code>m.push(['!(defined %s)' % identifier])</code>
<code>#elif</code>	<code>constant-expression</code>	<code>m[-1].push(['!%s' % m[-1].pop()),</code> <code>m[-1].push(['%s' % tokens,])</code>

		Note: Here we flip the existing state via a push(!pop()) then push the additional condition so that we have multiple conditions that are and'd together.
#else	N/A	m[-1].push('!%s' % m[-1].pop())
		Note: This is the negation of the sum of the previous #if, #elif statements.
#endif	N/A	m.pop()

---

**Note:** The above does not include error checking such as pop() from an empty stack.

---

Stringifying the matrix m:

```
flatList = []
for aList in m:
    assert(len(aList) > 0)
    if len(aList) > 1:
        # Add parenthesis so that when flatList is flattened then booleans are
        # correctly protected.
        flatList.append('(' + '%s' % ' && '.join(aList))
    else:
        flatList.append(aList[0])
return ' && '.join(flatList)
```

This returns for something like m is: [['a < 0'], ['!b', 'c > 45'], ['d < 27'], [],]

Then this gives: "a < 0 && (!b && c > 45) && d < 27"

**class** cpip.core.CppCond.**ConditionalState** (*theState, theIdOrCondExpr*)  
Holds a single conditional state.

**constexprStr** (*invert=False*)

Returns self as a string which is the concatenation of constant-expressions.

**flip** ()

Inverts the boolean such as for #else directive.

**flipAndAdd** (*theBool, theConstExpr*)

This handles an #elif command on this item in the stack. This flips the state (if theBool is True) and negates the last expression on the condition list then appends theConstExpr onto the condition list.

**hasBeenTrue**

Return True if the state has been True at any time in the lifetime of this object.

**negateLastState** ()

Inverts the state of the last item on the stack.

**state**

Returns boolean state of self.

**class** cpip.core.CppCond.**CppCond**

Provides a state stack to handle conditional compilation. This could be used by an implementation of conditional inclusion e.g. *ISO/IEC 14882:1998(E) section 16.1 Conditional inclusion [cpp.cond]*

Essentially this class provides a state machine that can be created altered and queried. The APIs available to the caller correspond to the if-section part of the applicable standard (i.e. #if #elif etc). Most APIs take two arguments;

**theBool** Is a boolean that is the result of the callers evaluation of a constant-expression.

**theIce** A string that represents the identifier or constant-expression in a way that the caller sees fit (i.e. this is not evaluated locally in any way). Combinations of such strings `_are_` merged by use of boolean logic (`!`) and `LPAREN` and `RPAREN`.

**close()**

Finalisation, may raise *ExceptionCppCond* is stack non-empty.

**hasBeenTrueAtCurrentDepth()**

Return True if the *ConditionalState* at the current depth has ever been True. This is used to decide whether to evaluate `#elif` expressions. They don't need to be if the *ConditionalState* has already been True, and in fact, the C Rationale (6.10) says that bogus `#elif` expressions should **not** be evaluated in this case - i.e. ignore syntax errors.

**isTrue()**

Returns True if all of the states in the stack are True, False otherwise.

**oElif (theBool, theConstExpr)**

Deal with the result of a `#elif`.

**theBool** Is a boolean that is the result of the callers evaluation of a constant-expression.

**theConstExpr** A string that represents the identifier or constant-expression in a way that the caller sees fit (i.e. this is not evaluated locally in any way). Combinations of such strings `_are_` merged by use of boolean logic (`!`) and `LPAREN` and `RPAREN`.

**oElse()**

Deal with the result of a `#else`.

**oEndif()**

Deal with the result of a `#endif`.

**oIf (theBool, theConstExpr)**

Deal with the result of a `#if`.

**theBool** Is a boolean that is the result of the callers evaluation of a constant-expression.

**theConstExpr** A string that represents the identifier or constant-expression in a way that the caller sees fit (i.e. this is not evaluated locally in any way). Combinations of such strings `_are_` merged by use of boolean logic (`!`) and `LPAREN` and `RPAREN`.

**oIfdef (theBool, theConstExpr)**

Deal with the result of a `#ifdef`.

**theBool** Is a boolean that is the result of the callers evaluation of a constant-expression.

**theConstExpr** A string that represents the identifier or constant-expression in a way that the caller sees fit (i.e. this is not evaluated locally in any way). Combinations of such strings `_are_` merged by use of boolean logic (`!`) and `LPAREN` and `RPAREN`.

**oIfndef (theBool, theConstExpr)**

Deal with the result of a `#ifndef`.

**theBool** Is a boolean that is the result of the callers evaluation of a constant-expression.

**theConstExpr** A string that represents the identifier or constant-expression in a way that the caller sees fit (i.e. this is not evaluated locally in any way). Combinations of such strings `_are_` merged by use of boolean logic (`!`) and `LPAREN` and `RPAREN`.

**stackDepth**

Returns the depth of the conditional stack as an integer.

**class** `cpip.core.CppCond.CppCondGraph`

Represents a graph of conditional preprocessing directives.

**isComplete**

True if the last if-section, if present is completed with an `#endif`.

**oElif** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a `#elif`.

**theFlc** A `cpip.core.FileLocation.FileLineColumn` object that identifies the position in the file.

**theTuIndex** An integer that represents the position in the translation unit.

**theBool** The current state of the conditional stack.

**theCe** The constant expression as a string (not evaluated).

**oElse** (*theFlc, theTuIdx, theBool*)

Deal with the result of a `#else`.

**theFlc** A `cpip.core.FileLocation.FileLineColumn` object that identifies the position in the file.

**theTuIndex** An integer that represents the position in the translation unit.

**theBool** The current state of the conditional stack.

**oEndif** (*theFlc, theTuIdx, theBool*)

Deal with the result of a `#endif`.

**theFlc** A `cpip.core.FileLocation.FileLineColumn` object that identifies the position in the file.

**theTuIndex** An integer that represents the position in the translation unit.

**theBool** The current state of the conditional stack.

**oIf** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a `#if`.

**theFlc** A `cpip.core.FileLocation.FileLineColumn` object that identifies the position in the file.

**theTuIndex** An integer that represents the position in the translation unit.

**theBool** The current state of the conditional stack.

**theCe** The constant expression as a string (not evaluated).

**oIfdef** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a `#ifdef`.

**theFlc** A `cpip.core.FileLocation.FileLineColumn` object that identifies the position in the file.

**theTuIndex** An integer that represents the position in the translation unit.

**theBool** The current state of the conditional stack.

**theCe** The constant expression as a string (not evaluated).

**oIfndef** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a `#ifndef`.

**theFlc** A `cpip.core.FileLocation.FileLineColumn` object that identifies the position in the file.

**theTuIndex** An integer that represents the position in the translation unit.



**theBool** The current state of the conditional stack.

**theCe** The constant expression as a string (not evaluated).

**visit** (*theVisitor*)

Take a visitor object and pass it around giving it each *CppCondGraphNode* object.

**class** `cpip.core.CppCond.CppCondGraphIfSection` (*theIfCppD, theFlc, theTuIdx, theBool, theCe*)

Class that represents a conditionally compiled section starting with #if... and ending with #endif.

**theIfCppD** A string, one of 'if', 'ifdef', 'ifndef'.

**theFlc** A `cpip.core.FileLocation.FileLineColumn` object that identifies the position in the file.

**theTuIndex** An integer that represents the position in the translation unit.

**theBool** The current state of the conditional stack.

**theCe** The constant expression as a string (not evaluated).

**oElif** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a #elif.

**oElse** (*theFlc, theTuIdx, theBool*)

Deal with the result of a #else.

**oEndif** (*theFlc, theTuIdx, theBool*)

Deal with the result of a #endif.

**oIf** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a #if.

**oIfdef** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a #ifdef.

**oIfndef** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a #ifndef.

**visit** (*theVisitor, theDepth*)

Take a visitor object make the pre/post calls.

**class** `cpip.core.CppCond.CppCondGraphNode` (*theCppDirective, theFileLineCol, theTuIdx, theBool, theConstExpr=None*)

Base class for all nodes in the *CppCondGraph*.

**canAccept** (*theCppD*)

True if I can accept a Preprocessing Directive; theCppD.

**oElif** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a #elif.

**oElse** (*theFlc, theTuIdx, theBool*)

Deal with the result of a #else.

**oEndif** (*theFlc, theTuIdx, theBool*)

Deal with the result of a #endif.

**oIf** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a #if.

**oIfdef** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a #ifdef.

**oIfndef** (*theFlc, theTuIdx, theBool, theCe*)

Deal with the result of a #ifndef.

**retStrList** (*theDepth*)

Returns a list of string representation.

**visit** (*theVisitor, theDepth*)

Take a visitor object make the pre/post calls.

**class** `cpip.core.CppCond.CppCondGraphVisitorBase`

Base class for a CppCondGraph visitor object.

**visitPost** (*theCcgNode, theDepth*)

Post-traversal call with a [CppCondGraphNode](#) and the integer depth in the tree.

**visitPre** (*theCcgNode, theDepth*)

Pre-traversal call with a [CppCondGraphNode](#) and the integer depth in the tree.

**class** `cpip.core.CppCond.CppCondGraphVisitorConditionalLines`

Allows you to find out if any particular line in a file is compiled or not. This is useful to be handed to the ITU to HTML generator that can colourize the HTML depending if any line is compiled or not.

This is a visitor class that walks the graph creating a dict of: `{file_id : [(line_num, boolean), ...], ...}` It then decomposes those into a map of `{file_id : LineConditionalInterpretation(), ...}` which can perform the actual conditional state determination.

API is really [isCompiled\(\)](#) and this returns -1 or 0 or 1. 0 means NO. 1 means YES and -1 means sometimes - for re-included files in a different macro environment perhaps.

**fileIds**

An unordered list of file IDs.

**isCompiled** (*fileId, lineNum*)

Returns 1 if this line is compiled, 0 if not or -1 if it is ambiguous i.e. sometimes it is and sometimes not when multiple inclusions.

**visitPre** (*theCcgNode, theDepth*)

Capture the fileID, line number and state.

**exception** `cpip.core.CppCond.ExceptionCppCond`

Simple specialisation of an exception class for the CppCond.

**exception** `cpip.core.CppCond.ExceptionCppCondGraph`

Simple specialisation of an exception class for the CppCondGraph.

**exception** `cpip.core.CppCond.ExceptionCppCondGraphElif`

When the CppCondGraph sees an `#elif` preprocessing directive in the wrong sequence.

**exception** `cpip.core.CppCond.ExceptionCppCondGraphElse`

When the CppCondGraph sees an `#endif` preprocessing directive in the wrong sequence.

**exception** `cpip.core.CppCond.ExceptionCppCondGraphIfSection`

Exception for a [CppCondGraphIfSection](#).

**exception** `cpip.core.CppCond.ExceptionCppCondGraphNode`

When the [CppCondGraphNode](#) sees an preprocessing directive in the wrong sequence.

**class** `cpip.core.CppCond.LineConditionalInterpretation` (*theList*)

Class that represents the conditional compilation state of every line in a file. This takes a list of `[(line_num, boolean), ...]` and interprets individual line numbers as to whether they are compiled or not.

If the same file is included twice with a different macro environment then it is entirely possible that `line_num` is not monotonic. In any case not every line number is present, the state of any unmentioned line is the state of the last mentioned line. Thus a simple dict is not useful.

We have to sort theList by line\_num and if there are duplicate line\_num with different boolean values then the conditional compilation state at that point is ambiguous.

**isCompiled** (*lineNum*)

Returns 1 if this line is compiled, 0 if not or -1 if it is ambiguous i.e. sometimes it is and sometimes not when multiply included.

This requires a search for the previously mentioned line state.

Will raise a ValueError if no prior state can be found, for example if there are no conditional compilation directives in the file. In this case it is up to the caller to handle this. CppCondGraphVisitorConditionalLines does this during visitPre() by artificially inserting line 1. See CppCondGraphVisitorConditionalLines.isCompiled()

cpip.core.CppCond.**StateConstExprFileLine**  
alias of StateConstExprLoc

## CppDiagnostic

Describes how a preprocessor class behaves under abnormal conditions.

**exception** cpip.core.CppDiagnostic.**ExceptionCppDiagnostic**  
Exception class for representing CppDiagnostic.

**exception** cpip.core.CppDiagnostic.**ExceptionCppDiagnosticPartialTokenStream**  
Exception class for representing partial remaining tokens.

**exception** cpip.core.CppDiagnostic.**ExceptionCppDiagnosticUndefined**  
Exception class for representing undefined behaviour.

**class** cpip.core.CppDiagnostic.**PreprocessDiagnosticKeepGoing**  
Sub-class that does not raise exceptions.

**partialTokenStream** (*msg, theLoc=None*)

Reports when an partial token stream exists (e.g. an unclosed comment).

*msg* The main message, a string.

*theLoc* The file locator e.g. FileLocation.FileLineCol. If present it must have: (fileId, lineNum colNum) attributes.

**undefined** (*msg, theLoc=None*)

Reports when an *undefined* event happens.

*msg* The main message, a string.

*theLoc* The file locator e.g. FileLocation.FileLineCol. If present it must have: (fileId, lineNum colNum) attributes.

**class** cpip.core.CppDiagnostic.**PreprocessDiagnosticRaiseOnError**  
Sub-class that raises an exception on a #error directive.

**error** (*msg, theLoc=None*)

Reports when an error event happens.

*msg* The main message, a string.

*theLoc* The file locator e.g. FileLocation.FileLineCol. If present it must have: (fileId, lineNum colNum) attributes.

**class** cpip.core.CppDiagnostic.**PreprocessDiagnosticStd**  
Describes how a preprocessor class behaves under abnormal conditions.

**debug** (*msg, theLoc=None*)

Reports a debug message.

*msg* The main message, a string.

*theLoc* The file locator e.g. `FileLocation.FileLineCol`. If present it must have: (*fileId*, *lineNum* *colNum*) attributes.

**error** (*msg, theLoc=None*)

Reports when an error event happens.

*msg* The main message, a string.

*theLoc* The file locator e.g. `FileLocation.FileLineCol`. If present it must have: (*fileId*, *lineNum* *colNum*) attributes.

**eventList**

A list of events in the order that they appear. An event is a pair of strings: (*type*, *message*)

**handleUnclosedComment** (*msg, theLoc=None*)

Reports when an unclosed comment is seen at EOF.

*msg* The main message, a string.

*theLoc* The file locator e.g. `FileLocation.FileLineCol`. If present it must have: (*fileId*, *lineNum* *colNum*) attributes.

**implementationDefined** (*msg, theLoc=None*)

Reports when an *implementation defined* event happens.

*msg* The main message, a string.

*theLoc* The file locator e.g. `FileLocation.FileLineCol`. If present it must have: (*fileId*, *lineNum* *colNum*) attributes.

**isDebug**

Whether a call to `debug()` will result in any log output.

**partialTokenStream** (*msg, theLoc=None*)

Reports when an partial token stream exists (e.g. an unclosed comment).

*msg* The main message, a string.

*theLoc* The file locator e.g. `FileLocation.FileLineCol`. If present it must have: (*fileId*, *lineNum* *colNum*) attributes.

**undefined** (*msg, theLoc=None*)

Reports when an *undefined* event happens.

*msg* The main message, a string.

*theLoc* The file locator e.g. `FileLocation.FileLineCol`. If present it must have: (*fileId*, *lineNum* *colNum*) attributes.

**unspecified** (*msg, theLoc=None*)

Reports when unspecified behaviour is happening. For example order of evaluation of '#' and '##'.

*msg* The main message, a string.

*theLoc* The file locator e.g. `FileLocation.FileLineCol`. If present it must have: (*fileId*, *lineNum* *colNum*) attributes.

**warning** (*msg, theLoc=None*)

Reports when an warning event happens.

*msg* The main message, a string.

*theLoc* The file locator e.g. `FileLocation.FileLineCol`. If present it must have: (`fileId`, `lineNum` `colNum`) attributes.

## FileIncludeGraph

Captures the `#include` graph of a preprocessed file.

`cpip.core.FileIncludeGraph.DUMMY_FILE_LINENUM = -1`

In the graph the line number is ignored for dummy roots and this one used instead

`cpip.core.FileIncludeGraph.DUMMY_FILE_NAME = None`

The file ID for a ‘dummy’ file. This is used as the artificial root node for all the pre-includes and the ITU

**exception** `cpip.core.FileIncludeGraph.ExceptionFileIncludeGraph`

Simple specialisation of an exception class for the *FileIncludeGraph* classes.

**exception** `cpip.core.FileIncludeGraph.ExceptionFileIncludeGraphRoot`

Exception for issues for dummy file ID’s.

**exception** `cpip.core.FileIncludeGraph.ExceptionFileIncludeGraphTokenCounter`

Exception for issues for token counters.

**class** `cpip.core.FileIncludeGraph.FigVisitorBase`

Base class for visitors, see *FigVisitorTreeNodeBase* for base class for tree visitors.

**visitGraph** (*theFigNode*, *theDepth*, *theLine*)

Hierarchical visitor pattern. This is given a *FileIncludeGraph* as a graph node. *theDepth* is the current depth in the graph as an integer, *theLine* the line that is a non-monotonic sibling node ordinal.

**class** `cpip.core.FileIncludeGraph.FigVisitorFileSet`

Simple visitor that just collects the set of file IDs in the include graph and a count of how often they are seen.

**fileNameMap**

Dictionary of number of times each file is seen: {file : count, ...}.

**fileNameSet**

The set of file names seen.

**visitGraph** (*theFigNode*, *theDepth*, *theLine*)

Hierarchical visitor pattern.

*theFigNode* A *FileIncludeGraph* as a graph node.

*theDepth* The current depth in the graph as an integer.

*theLine* The line that is a non-monotonic sibling node ordinal.

**class** `cpip.core.FileIncludeGraph.FigVisitorTree` (*theNodeClass*)

This visitor can visit a graph of *FileIncludeGraphRoot* and *FileIncludeGraph* that recreates a tree of Node(s) the type of which are supplied by the user. Each node instance will be constructed with either an instance of a *FileIncludeGraphRoot* or *FileIncludeGraph* or, in the case of a pseudo root node then `None`.

**depth**

Returns the current depth in this graph representation. Changes to this determine if the node is a child, sibling or ancestor.

**tree** ()

Returns the top level node object as the only copy. This also finalises the tree.

**visitGraph** (*theFigNode*, *depth*, *line*)

Visit the give node.

**class** `cpip.core.FileIncludeGraph.FigVisitorTreeNodeBase` (*theLineNum*)

Base class for nodes created by a tree visitor. See *FigVisitorBase* for the base class for non-tree visitors.

**addChild** (*theObj*)

Add the object as a child.

**finalise** ()

This will be called on finalisation. This is an opportunity for the root (None) not to accumulate properties from its immediate children for example. For depth first finalisation the child class should call finalise on each child first as this function does.

**lineNum**

The line number of the parent file that included me.

**class** `cpip.core.FileIncludeGraph.FileIncludeGraph` (*theFile*, *theState*, *theCondition*, *theLogic*)

Recursive class that holds a graph of include files and and line numbers of the file that included them.

This class builds up a graph (actually a tree) of file includes. The insertion order is significant in that it is expected to be the order experienced by a translation unit processor. `addBranch()` is the way to add to the data structure.

*theFile* - a file ID (e.g. a path)

*theState* - a boolean conditional compilation state.

*theCondition* - a conditional compilation condition string e.g. "a >= b+2".

*theLogic* - a string explanation of how that the file was found.

If *theLogic* is taken from an IncludeHandler as a list of items. e.g. [`<foo.h>`, `CP=None`, `sys=None`, `usr=include/foo.h`] Each string after item[0] is of the form: `key=value` Where:

`key` is a key in `self.INCLUDE_ORIGIN_CODES` = is the '=' character. `value` is the result, or 'None' if not found.

[0] is the invocation [-1] is the final resolution.

The intermediate ones are various tries in order. So [`<foo.h>`, `CP=None`, `sys=None`, `usr=include/foo.h`] would mean:

0. '`<foo.h>`' the include directive was: `#include <foo.h>`

1. '`CP=None`' the Current place was searched and nothing found.

2. '`sys=None`' the system include(s) were searched and nothing found.

3. '`usr=include/foo.h`' the user include(s) were searched and `include/foo.h` was found.

This class does not distinguish between conditional compilation states that are True or False. Nor does this class evaluate the *Condition* in any way, it is merely stored for representation.

**acceptVisitor** (*visitor*, *depth*, *line*)

Hierarchical visitor pattern. This accepts a visitor object and calls `visitor.visitGraph(self, depth, line)` on that object where *depth* is the current depth in the graph as an integer and *line* the line that is a non-monotonic sibling node ordinal.

**addBranch** (*theFileS*, *theLine*, *theIncFile*, *theState*, *theCondition*, *theLogic*)

Adds a branch to the graph.

*theFileS* is a list of files that form the branch.

*theLine* is an integer value of the line number of the `#include` statement of the last named file in *theFileS*.

*theIncFile* is the file that is included.

theState is a boolean that describes the conditional compilation state.

theCondition is the conditional compilation test e.g. '1>0'

theLogic is a string representing how the branch was obtained.

May raise *ExceptionFileIncludeGraph* if:

- 0.The branch is zero length.
- 1.The branch does not match the existing graph (this function just immediately checks the first item on the branch but the others are done recursively).
- 2.theLine is a duplicate of an existing line.
- 3.The branch has missing nodes.

#### **condComp**

Returns the condition, as a string, under which this file was included e.g. "(a > b) && (1 > 0)".

#### **condCompState**

Returns the recorded conditional compilation state as a boolean.

#### **dumpGraph** (theS=<\_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>, theI='')

Writes out the graph to a stream.

#### **fileName**

Returns the current file name.

#### **findLogic**

Returns the findLogic string passed in in the constructor.

#### **genChildNodes** ()

Yields each child node as a *FileIncludeGraph* object.

#### **numTokens**

The total number of tokens seen by the PpLexer. Returns None if not initialised. Note: This is the number of tokens for this file only, it does not include the tokens that this file might include.

#### **numTokensIncChildren**

The total number of tokens seen by the PpLexer including tokens from files included by this one. Returns None if not initialised.

May raise *ExceptionFileIncludeGraphTokenCounter* is the token counters have been loaded inconsistently (i.e. the children have not been loaded).

#### **numTokensSig**

The number of significant tokens seen by the PpLexer. A significant token is a non-whitespace, non-conditionally compiled token. Returns None if not initialised.

---

**Note:** This is the number of tokens for this file only, it does not include the tokens that this file might include.

---

#### **numTokensSigIncChildren**

The number of significant tokens seen by the PpLexer including tokens from files included by this one. A significant token is a non-whitespace, non-conditionally compiled token. Returns None if not initialised.

May raise *ExceptionFileIncludeGraphTokenCounter* is the token counters have been loaded inconsistently (i.e. the children have not been loaded).

#### **retBranches** ()

Returns a list of lists of the branches with '#' then the line number.

**retLatestBranch ()**

Returns the branch to the last inserted leaf as a list of branch strings.

**retLatestBranchDepth ()**

Walks the graph and returns an integer that is the depth of the latest branch.

**retLatestBranchPairs ()**

Returns the branch to the last inserted leaf as a list of pairs (filename, integer\_line).

**retLatestLeaf ()**

Returns the last inserted leaf, a *FileIncludeGraph* object.

**retLatestNode (theBranch)**

Returns the last inserted node, a *FileIncludeGraph* object on the supplied branch.

This is generally used during dynamic construction by a caller that understands the state of the file include branch.

**setTokenCounter (theTokCounter)**

Sets the token counter for this node which is a PpTokenCount object. The PpLexer sets this as the token count for this file only. This files #includes are a separate token counter.

**tokenCounter**

Gets the token counter for this node, a PpTokenCount object.

**class** cpip.core.FileIncludeGraph.**FileIncludeGraphRoot**

Root class of the file include graph. This is used when there is a virtual or dummy root. It contains a list of *FileIncludeGraph* objects. In this way it can represent the list of graphs that would result from a list of pre-includes followed by the ITU itself.

In practice this is used by the PpLexer for this purpose where the dummy root is represented by None.

**acceptVisitor (visitor)**

Hierarchical visitor pattern. This accepts a visitor object and calls visitor.visitGraph(self, depth, line) on that object where depth is the current depth in the graph as an integer and line the line that is a non-monotonic sibling node ordinal.

**addGraph (theGraph)**

Add a *FileIncludeGraph* object.

**dumpGraph (theS=<\_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>)**

Dump the node for debug/trace.

**graph**

The latest *FileIncludeGraph* object I have. Will raise a *ExceptionFileIncludeGraphRoot* if nothing there.

**numTrees ()**

Returns the number of *FileIncludeGraph* objects.

## FileIncludeStack

This module represents a stack of file includes as used by the *PpLexer.PpLexer*

**exception** cpip.core.FileIncludeStack.**ExceptionFileIncludeStack**

Exception for FileIncludeStack object.

**class** cpip.core.FileIncludeStack.**FileInclude** (theFpo, theDiag)

Represents a single TU fragment with a PpTokeniser and a token counter.

*theFpo* A FilePathOrigin object that identifies the file.



*theDiag* A CppDiagnostic object to give to the PpTokeniser.

**tokenCountInc** (*tok, isUnCond, num=1*)  
Increment the token counter.

**tokenCounterAdd** (*theC*)  
Add a token counter to my token counter (used when a macro is declared).

**class** `cpip.core.FileIncludeStack`.**FileIncludeStack** (*theDiagnostic*)  
This maintains information about the stack of file includes. This holds several stacks (or representations of them):

*self.\_ppts* A stack of *PpTokeniser.PpTokeniser* objects.

*self.\_figr* A *FileIncludeGraph.FileIncludeGraphRoot* for tracking the #include graph.

*self.\_fns* A stack of file IDs as strings (e.g. the file path).

*self.\_tcs* A *PpTokenCount.PpTokenCountStack* object for counting tokens.

**currentFile**  
Returns the file ID from the top of the stack.

**depth**  
Returns the current include depth as an integer.

**fileIncludeGraphRoot**  
The *FileIncludeGraph.FileIncludeGraphRoot* object.

**fileLineCol**  
Return an instance of FileLineCol from the current physical line column.

**fileStack**  
Returns a copy of the stack of file IDs.

**finalise** ()  
Finalisation, may raise an ExceptionFileIncludeStack.

**includeFinish** ()  
End an #include file, returns the file ID that has been finished.

**includeStart** (*theFpo, theLineNum, isUncond, condStr, incLogic*)  
Start an #include file.

*theFpo* A *FileLocation.FilePathOrigin* object that identifies the file.

*theLineNum* The integer line number of the file that includes (None if Root).

*isUncond* A boolean that is the conditional compilation state.

*condStr* A string of the conditional compilation stack.

*incLogic* A string that describes the find include logic.

**ppt**  
Returns the PpTokeniser from the top of the stack.

**tokenCountInc** (*tok, isUnCond, num=1*)  
Increment the token counter.

**tokenCounter** ()  
Returns the Token Counter object at the tip of the stack.

**tokenCounterAdd** (*theC*)  
Add a token counter to my token counter (used when a macro is declared).

## IncludeHandler

Provides handlers for #including files.

**class** `cpip.core.IncludeHandler.CppIncludeStd` (*theUsrDirs, theSysDirs*)

Class that applies search rules for #include statements.

Search tactics based on RVCT and Berkeley UNIX search rules:

I <b>is</b> the usr includes.			
J <b>is</b> the sys includes.			
Size of I	Size of J	<code>#include &lt;...&gt;</code>	<code>#include "..."</code>
0	0	None	CP
0	>0	SYSTEMINCLUDEdirs	CP, SYSTEMINCLUDEdirs
>0	0	USERINCLUDEdirs	CP, USERINCLUDEdirs
>0	>0	SYSTEMINCLUDEdirs, USERINCLUDEdirs	CP, USERINCLUDEdirs, SYSTEMINCLUDEdirs

ISO/IEC 9899:1999 (E) 6.10.2-3 means that a failure of q-char must be retried as if it was a h-char. i.e. A failure of a q-char-sequence thus: `#include "..."`

Is to be retried as if it was written as a h-char-sequence thus: `#include <...>`

See: `_includeQcharseq()`

**INCLUDE\_ORIGIN\_CODES** = {'usr': 'User include directories', 'TU': 'Translation unit', 'comp': 'Compiler specific direct

Codes for the results of a search for an include

**canInclude** ()

Returns True if the last include succeeded.

**clearFindLogic** ()

Clears the list of find results for a single #include statement.

**clearHistory** ()

Clears the CP stack. This needed if you use this class as a persistent one and it encounters an exception. You need to call this function before you can reuse it.

**cpStack**

Returns the current stack of current places.

**cpStackPop** ()

Pops and returns the CP string off the current place stack. This is public so that the PpLexer can use it when processing pre-include files that might themselves include other files.

**cpStackPush** (*theFpo*)

Appends the CP from the FilePathOrigin to the current place stack. This is public so that the PpLexer can use it when processing pre-include files that might themselves include other files.

**cpStackSize**

Returns the size of the current stack of current places.

**currentPlace**

Returns the last current place or None if #include failed.

**endInclude** ()

Notify end of #include'd file. This pops the CP stack.

**finalise()**

Finalise at the end of the translation unit. Might raise a `ExceptionCppInclude`.

**findLogic**

Returns a list of strings that describe `_how_` the file was found For example:

```
['<foo.h>', 'CP=None', 'sys=None', 'usr=include/foo.h']
```

Each string after [0] is of the form: `key=value` Where:

1. `key` is a key in `self.INCLUDE_ORIGIN_CODES`
2. `=` is the '=' character.
3. `value` is the result, or 'None' if not found.
4. Item [0] is the invocation
5. Item [-1] is the final resolution.

The intermediate ones are various tries in order. So:

```
['<foo.h>', 'CP=None', 'sys=None', 'usr=include/foo.h']
```

Would mean:

- [0]: '<foo.h>' the include directive was: `#include <foo.h>`
- [1]: 'CP=None' the Current place was searched and nothing found.
- [2]: 'sys=None' the system include(s) were searched and nothing found.
- [3]: 'usr=include/foo.h' the user include(s) were searched and `include/foo.h` was found.

**includeHeaderName** (*theStr*)

Return the file location of a `#include` header-name where the header-name is a pp-token either a `<h-char-sequence>` or a "q-char-sequence" (including delimiters). If not None return value this also records the CP for the file.

**includeNextHeaderName** (*theStr*)

Return the file location of a `#include_next` header-name where the header-name is a pp-token either a `<h-char-sequence>` or a "q-char-sequence" (including delimiters).

This is a GCC extension, see: <https://gcc.gnu.org/onlinedocs/cpp/Wrapper-Headers.html>

This never records the CP for the found file (if any).

**initialTu** (*theTuIdentifier*)

Given an Translation Unit Identifier this should return a class `FilePathOrigin` or None for the initial translation unit. As a precaution this should include code to check that the stack of current places is empty. For example:

```
if len(self._cpStack) != 0:
    raise ExceptionCppInclude('setTu() with CP stack: %s' % self._cpStack)
```

**validateCpStack()**

Tests the coherence of the CP stack. A None can not be followed by a non-None.

**class** `cpip.core.IncludeHandler.CppIncludeStdOs` (*theUsrDirs, theSysDirs*)

This implements `_searchFile()` based on an OS file system call.

**initialTu** (*theTuPath*)

Given an path as a string this returns the class `FilePathOrigin` or None for the initial translation unit

**class** `cpip.core.IncludeHandler.CppIncludeStdin` (*theUsrDirs, theSysDirs*)

This reads stdin for the ITU but delegates `_searchFile()` to the OS file system call.

**initialTu** (*theTuPath*)

Given an path as a string this returns the class FilePathOrigin or None for the initial translation unit

**class** cpip.core.IncludeHandler.**CppIncludeStringIO** (*theUsrDirs, theSysDirs, theInitialTuContent, theFilePathToContent*)

This implements `_searchFile()` based on a lookup of stings that returns StringIO file-like object.

**initialTu** (*theTuIdentifier*)

Given an path as a string this returns the class FilePathOrigin or None for the initial translation unit

**exception** cpip.core.IncludeHandler.**ExceptionCppInclude**

Simple specialisation of an exception class for the CppInclude.

**class** cpip.core.IncludeHandler.**FilePathOrigin** (*fileObj, filePath, currentPlace, origin*)

FilePathOrigin is a class used externally to collect:

- An open file object that can be read by the caller.
- The file path of that object, wherever found.
- The ‘current place’ of that file, wherever found. This will affect subsequent calls.
- The origin code, i.e. how it was found.

Any or all of these attributes may be None as the methods `_searchFile()`, `_includeQcharseq()` and `_includeHcharseq()` return such an object (or None).

**currentPlace**

Alias for field number 2

**fileObj**

Alias for field number 0

**filePath**

Alias for field number 1

**origin**

Alias for field number 3

## ItuToTokens

Converts an ITU (i.e. a file like object and tokenises it into extended preprocessor tokens. This does not act on any preprocessing directives.

**class** cpip.core.ItuToTokens.**ItuToTokens** (*theFileObj=None, theFileId=None, theDiagnostic=None*)

Tokenises a file like object.

**genTokensKeywordPpDirective** ()

Process the file and generate tokens. This changes the type to a keyword or preprocessing-directive if it can do so.

## MacroEnv

This an environment of macro declarations

It implements *ISO/IEC 9899:1999(E) section 6 (aka ‘C’)* and *ISO/IEC 14882:1998(E) section 16 (aka ‘C++’)*

**exception** cpip.core.MacroEnv.**ExceptionMacroEnv**

Exception when handling MacroEnv object.

**exception** `cpip.core.MacroEnv.ExceptionMacroEnvInvalidRedefinition`

Exception for a invalid redefinition of a macro. NOTE: Under C rules (C Rationale 6.10.3) callers should merely issue a suitable diagnostic.

**exception** `cpip.core.MacroEnv.ExceptionMacroEnvNoMacroDefined`

Exception when trying to access a PpDefine that is not currently defined.

**exception** `cpip.core.MacroEnv.ExceptionMacroIndexError`

Exception when an access to a PpDefine that generates a IndexError.

**exception** `cpip.core.MacroEnv.ExceptionMacroReplacementInit`

Exception in the constructor.

**exception** `cpip.core.MacroEnv.ExceptionMacroReplacementPredefinedRedefinition`

Exception for a redefinition of a macro id that is predefined.

**class** `cpip.core.MacroEnv.MacroEnv` (*enableTrace=False, stdPredefMacros=None*)

Represents a set of #define directives that represent a macro processing environment. This provides support for #define and #undef directives. It also provides support for macro replacement see: *ISO/IEC 9899:1999 (E) 6.10.3 Macro replacement*.

**enableTrace** Allows calls to `_debugTokenStream()` that may or may not produce log output (depending on logging level). If True this makes this code run slower, typically 3x slower

**stdPredefMacros** If present should be a dictionary of: `{identifier : replacement_string_terminated, ...}` For example:

```
{
    '__DATE__' : 'First of June\n',
    '__TIME__' : 'Just before lunchtime.\n',
}
```

Each identifier must be in `STD_PREDEFINED_NAMES`

**allStaticMacroDependencies()**

Returns a `DuplexAdjacencyList()` of macro dependencies for the Macro environment. All objects in the `cpip.util.Tree.DuplexAdjacencyList` are macro identifiers as strings.

A `cpip.util.Tree.DuplexAdjacencyList` can be converted to a `cpip.util.Tree.Tree` and that can be converted to a `cpip.util.DictTree.DictTree`

**clear()**

Clears the macro environment.

**define** (*theGen, theFile, theLine*)

Defines a macro. theGen should be in the state immediately after the #define i.e. this will consume leading whitespace and the trailing newline.

Will raise a `ExceptionMacroEnvInvalidRedefinition` if the redefinition is not valid. May raise a `PpDefine.ExceptionCpipDefineInit` (or sub class) on failure.

On success it returns the identifier of the macro as a string.. The insertion is stable i.e. a valid re-definition does not replace the existing definition so that the existing state of the macro definition (file, line, reference count etc. are preserved.

**defined** (*theTtt, flagInvert, theFileLineCol=None*)

If the PpToken theTtt is an identifier that is currently defined then this returns 1 as a PpToken, 0 as a PpToken otherwise. If the macro exists in the environment its reference count is incremented.

**theFileLineCol** Is a `FileLocation.FileLineCol` object.

See: *ISO/IEC 9899:1999 (E) 6.10.1*.

**genMacros** (*theIdentifier=None*)

Generates PpDefine objects encountered during my existence. Macros that have been undefined will be generated first in order of un-definition followed by the currently defined macros in identifier order.

Macros that have been #undef'd will have the attribute isCurrentlyDefined as False.

**genMacrosInScope** (*theIdent=None*)

Generates PpDefine objects encountered during my existence and still in scope i.e. not yet un-defined.

If theIdent is not None then only that named macros will be yielded.

**genMacrosOutOfScope** (*theIdent=None*)

Generates PpDefine objects encountered during my existence but then undefined in the order of un-definition.

If theIdent is not None then only that named macros will be yielded.

**getUndefMacro** (*theIdx*)

Returns the PpDefine object from the undef list for the given index. Will raise an *ExceptionMacroIndexError* if the index is out of range.

**hasMacro** (*theIdentifier*)

Returns True if the environment has the macro.

NOTE: This does `_not_` increment the reference count so should not be used when processing `#ifdef ...`, `#if defined ...` or `#if !defined ...` for those use `isDefined()` and `defined()` instead.

**isDefined** (*theTtt, theFileLineCol=None*)

Returns True theTtt is an identifier that is currently defined, False otherwise. If True this increments the macro reference.

*theFileLineCol* Is a `FileLocation.FileLineCol` object.

See: *ISO/IEC 9899:1999 (E) 6.10.1.*

**macro** (*theIdentifier*)

Returns the macro identified by the identifier. Will raise a *ExceptionMacroEnvNoMacroDefined* is undefined.

**macroHistory** (*incEnv=True, onlyRef=True*)

Returns the macro history as a multi-line string

**macroHistoryMap** ()

Returns a map of {`ident` : ([`ints`, ...], `True/False`), ...} Where the macro identifier is mapped to a pair where: `pair[0]` is a list of indexes into `getUndefMacro()`. `pair[1]` is boolean, True if the identifier is currently defined i.e. it is the value of `self.hasMacro(ident)`. The macro can be obtained by `self.macro()`.

**macroNotDefinedDependencies** ()

Returns a map of {`identifier` : [`class FileLineColumn`, ...], ...} where there has been an `#ifdef` and nothing is defined. Thus these macros, if present, could alter the outcome i.e. it is dependency on them NOT being defined.

**macroNotDefinedDependencyNames** ()

Returns an unsorted list of identifies where there has been an `#ifdef` and nothing is defined. Thus these macros, if present, could alter the outcome i.e. it is dependency on them NOT being defined.

**macroNotDefinedDependencyReferences** (*theIdentifier*)

Returns an ordered list of class `FileLineColumn` for an identifier where there has been an `#ifdef` and nothing is defined. Thus these macros, if present, could alter the outcome i.e. it is dependency on them NOT being defined.

**macros ()**

Returns and unsorted list of strings of current macro identifiers.

**mightReplace (theTtt)**

Returns True if theTok might be able to be expanded. ‘Might’ is not ‘can’ or ‘will’ because of this:

```
#define FUNC(a,b) a-b
FUNC FUNC(45,3)
```

Becomes:

```
FUNC 45 -3
```

Thus `mightReplace('FUNC', ...)` is True in both cases but actual replacement only occurs once for the second FUNC.

**referencedMacroIdentifiers (sortedByRefCount=False)**

Returns an unsorted list of macro identifiers that have a reference count > 0. If `sortedByRefCount` is True the list will be in increasing order of reference count then by name. Use `reverse()` on the result to get decreasing order. If `sortedByRefCount` is False the return value is unsorted.

**replace (theTtt, theGen, theFileLineCol=None)**

Given a PpToken this returns the replacement as a list of `[class PpToken, ...]` that is the result of the substitution of macro definitions.

**theGen** Is a generator that might be used in the case of function-like macros to consume their argument lists.

**theFileLineCol** Is a `FileLocation.FileLineCol` object.

**set\_\_FILE\_\_ (theStr)**

This sets the `__FILE__` macro directly.

**set\_\_LINE\_\_ (theStr)**

This sets the `__LINE__` macro directly.

**undef (theGen, theFile, theLine)**

Removes a definition from the map and adds the PpDefine to `self._undefS`. It returns None. If no definition exists this has no side-effects on the internal representation.

## PpDefine

This handles definition, undefinition, redefinition, replacement and rescanning of macro declarations

It implements: *ISO/IEC 9899:1999(E) section 6 (aka ‘C99’)* and/or: *ISO/IEC 14882:1998(E) section 16 (aka ‘C++98’)*

**exception `cpip.core.PpDefine.ExceptionCpipDefine`**

Exception when handling PpDefine object.

**exception `cpip.core.PpDefine.ExceptionCpipDefineBadArguments`**

Exception when scanning an argument list for a function style macro fails. NOTE: This is only raised during replacement not during initialisation.

**exception `cpip.core.PpDefine.ExceptionCpipDefineBadWs`**

Exception when calling bad whitespace is in a define statement. See: *ISO/IEC 9899:1999(E) Section 6.10-f* and *ISO/IEC 14882:1998(E) 16-2*

**exception `cpip.core.PpDefine.ExceptionCpipDefineDupeId`**

Exception for a function-like macro has duplicates in the identifier-list.

**exception** `cpip.core.PpDefine.ExceptionCpipDefineInit`

Exception when creating PpDefine object fails.

**exception** `cpip.core.PpDefine.ExceptionCpipDefineInitBadLine`

Exception for a bad line number given as argument.

**exception** `cpip.core.PpDefine.ExceptionCpipDefineInvalidCmp`

Exception for a redefinition where the identifiers are different.

**exception** `cpip.core.PpDefine.ExceptionCpipDefineMissingWs`

Exception when calling missing ws between identifier and replacement tokens.

See: *ISO/IEC 9899:1999(E) Section 6.10.3-3* and *ISO/IEC 14882:1998(E) Section ???*

---

**Note:** The executable, `cpp`, says for `#define PLUS+`

```
src.h:1:13: warning: ISO C requires whitespace after the macro name
```

---

**exception** `cpip.core.PpDefine.ExceptionCpipDefineReplace`

Exception when replacing a macro definition fails.

**class** `cpip.core.PpDefine.PpDefine` (*theTokGen*, *theFileId*, *theLine*)

Represents a single `#define` directive and performs *ISO/IEC 9899:1999 (E) 6.10.3 Macro replacement*.

***theTokGen*** A PpToken generator that is expected to generate pp-tokens that appear after the start of the `#define` directive from the first non-whitespace token onwards i.e. the `__init__` will, itself, consume leading whitespace.

***theFileId*** A string that represents the file ID.

***theLine*** A positive integer that represents the line in theFile that the `#define` statement occurred.

Definition example, object-like macros:

```
[identifier, [replacement-list (opt)], new-line, ...]
```

Or function-like macros:

```
[
    identifier,
    lparen,
    [identifier-list (opt)],
    ')',
    replacement-list,
    new-line,
    ...
]
```

---

**Note:** No whitespace is allowed between the identifier and the lparen of function-like macros.

---

The `identifier-list` of parameters is stored as a list of names. The `replacement-list` is stored as a list of preprocessor tokens. Leading and trailing whitespace in the replacement list is removed to facilitate redefinition comparison.

**CPP\_CONCAT\_OP = '##'**

C standard definition of concatenation operator



**CPP\_STRINGIZE\_OP = '#'**

C standard definition of string'izing operator

**IDENTIFIER\_SEPERATOR = ','**

C standard definition of identifier separator in function-like macros

**INITIAL\_REF\_COUNT = 0**

This is what the reference count is set to on construction

**LPAREN = '('**

C standard definition of left parenthesis

**PLACEMARKER = None**

Our representation of a placemark token

**RPAREN = ')'**

C standard definition of right parenthesis

**STRINGIZE\_WHITESPACE\_CHAR = ' '**

Whitespace runs are replaced by a single space ISO/IEC 9899:1999 (E) 6.10.3.2-2

**VARIABLE\_ARGUMENT\_IDENTIFIER = '...'**

Variable argument (variadic) macro definitions

**VARIABLE\_ARGUMENT\_SUBSTITUTE = '\_\_VA\_ARGS\_\_'**

Variable argument (variadic) macro substitution

**assertReplListIntegrity()**

Tests that any identifier tokens in the replacement list are actually replaceable. This will raise an assertion failure if not. It is really an integrity tests to see if an external entity has grabbed a reference to the replacement list and set a token to be not replaceable.

**consumeFunctionPreamble** (*theGen*)

This consumes tokens to the preamble of a Function style macro invocation. This really means consuming whitespace and the opening LPAREN.

This will return either:

- None - Tokens including the leading LPAREN have been consumed.
- List of (token, token\_type) if the LPAREN is not found.

For example given this:

```
#define t(a) a+2
t    (21) - t    ;
```

For the first `t` this would consume ' (' and return None leaving the next token to be ('21', 'pp-number').

For the second `t` this would consume ' ; ' and return:

```
[
    (' ', 'whitespace'),
    (';', 'preprocessing-op-or-punc'),
]
```

This allows the MacroReplacementEnv to generate the correct result:

```
21 +2 - t ;
```

**expandArguments**

The flag that says whether arguments should be expanded. For object like macros this will be False. For

function like macros this will be False if there is a stringize (#) or a token pasting operator (##). True otherwise.

**fileId**

The file ID given as an argument in the constructor.

**identifier**

The macro identifier i.e. the name as a string.

**incRefCount** (*theFileLineCol=None*)

Increment the reference count. Typically callers do this when replacement is certain of in the event of definition testing

*theFileLineCol* A FileLocation.FileLineCol object.

For example:

```
#ifdef SPAM or defined(SPAM) etc.
```

Or if the macro is expanded e.g. #define SPAM\_N\_EGGS spam and eggs

The menu is SPAM\_N\_EGGS.

**isCurrentlyDefined**

Returns True if the current instance is a valid definition i.e. it has not been #undef'd.

**isObjectTypeMacro**

True if this is an object type macro and False if it is a function type macro.

**isReferenced**

Returns True if the reference count has been incremented since construction.

**isSame** (*other*)

Tests 'sameness'. Returns: -1 if the identifiers are different. 1 if the identifiers are the same but redefinition is NOT allowed. 0 if the identifiers are the same but redefinition is allowed i.e. the macros are equivalent.

**isValidRefefinition** (*other*)

Returns True if this is a valid redefinition of *other*, False otherwise. Will raise an *ExceptionCpipDefineInvalidCmp* if the identifiers are different. Will raise an *ExceptionCpipDefine* if either is not currently defined.

From: **ISO/IEC 9899:1999 (E) 6.10.3:**

1. **Two replacement lists are identical if and only if the preprocessing** tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.
2. **An identifier currently defined as a macro without use of lparen** (an object-like macro) may be redefined by another #define preprocessing directive provided that the second definition is an object-like macro definition and the two replacement lists are identical, otherwise the program is ill-formed.
3. **An identifier currently defined as a macro using lparen (a** function-like macro) may be redefined by another #define preprocessing directive provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical, otherwise the program is ill-formed.

See also: **ISO/IEC 14882:1998(E) 16.3 Macro replacement [cpp.replace]**

**line**

The line number given as an argument in the constructor.

**parameters**

The list of parameter names as strings for a function like macros or None if this is an object type Macro.

**refCount**

Returns the current reference count as an integer less its initial value on construction.

**refFileLineCols**

Returns the list of FileLineCol objects where this macro was referenced.

**replaceArgumentList** (*theArgList*)

Given an list of arguments this does argument substitution and returns the replacement token list. The argument list is of the form given by *retArgumentListTokens()*. The caller must have replaced any macro invocations in theArgList before calling this method.

---

**Note:** For function style macros only.

---

**replaceObjectStyleMacro** ()

Returns a list of [ (token, token\_type), ... ] from the replacement of an object style macro.

**replacementTokens**

The list of zero or more replacement token as a list of *PpToken.PpToken*

**replacements**

The list of zero or more replacement tokens as strings.

**retArgumentListTokens** (*theGen*)

For a function macro this reads the tokens following a LPAREN and returns a list of arguments where each argument is a list of PpToken objects.

Thus this function returns a list of lists of *PpToken.PpToken* objects, for example given this:

```
#define f(x,y) ...
f(a,b)
```

This function, then passed a,b) returns:

```
[
  [
    PpToken.PpToken('a', 'identifier'),
  ],
  [
    PpToken.PpToken('b', 'identifier'),
  ],
]
```

And an invocation of: *f* (1 (, ) 2, 3) i.e. this gets passed via the generator "1 (, ) 2, 3 " and returns two arguments:

```
[
  [
    PpToken('1', 'pp-number'),
    PpToken('(', 'preprocessing-op-or-punc'),
    PpToken(',', 'preprocessing-op-or-punc'),
    PpToken(')', 'preprocessing-op-or-punc'),
    PpToken('2', 'pp-number'),
  ],
  [
    PpToken('3', 'pp-number'),
  ],
]
```

So this function supports two cases:

1. Parsing function style macro declarations.
2. Interpreting function style macro invocations where the argument list is subject to replacement before invoking the macro.

In the case that an argument is missing a `PpDefine.PLACEMARKER` token is inserted. For example:

```
#define FUNCTION_STYLE(a,b,c) ...  
FUNCTION_STYLE(,2,3)
```

Gives:

```
[  
  PpDefine.PLACEMARKER,  
  [  
    PpToken.PpToken('2',      'pp-number'),  
  ],  
  [  
    PpToken.PpToken('3',      'pp-number'),  
  ],  
]
```

Placemark tokens are not used if the macro is defined with no arguments. This might raise a *ExceptionCpipDefineBadArguments* if the list does not match the prototype or a *StopIteration* if the token list is too short. This ignores leading and trailing whitespace for each argument.

TODO: Raise an *ExceptionCpipDefineBadArguments* if there is a `#define` statement. e.g.:

```
#define f(x) x x  
f (1  
#undef f  
#define f 2  
f)
```

**strIdentPlusParam()**

Returns the identifier name and parameters if a function-like macro as a string.

**strReplacements()**

Returns the replacements tokens with minimised whitespace as a string.

**tokenCounter**

The `PpTokenCount` object that counts tokens that have been consumed from the input.

**tokensConsumed**

The total number of tokens consumed by the class.

**undef(theFileId, theLineNum)**

Records this instance of a macro `#undef'd` at a particular file and line number. May raise an *ExceptionCpipDefine* if already undefined or the line number is bad.

**undefFileId**

The file ID where this macro was `undef'd` or `None`.

**undefLine**

The line number where this macro was `undef'd` or `None`.

## PpLexer

Generates tokens from a C or C++ translation unit.

TODO: Fix accidental token pasting. See: TestFromCppInternalsTokenspacing and, connected is: TODO: Set setPrevWs flag on the token where necessary.

TODO: Preprocessor statements in arguments of function like macros. Sect. 3.9 of cpp.pdf and existing MacroEnv tests.

**exception** `cpip.core.PpLexer.ExceptionConditionalExpression`

Exception when eval() conditional expressions.

**exception** `cpip.core.PpLexer.ExceptionPpLexer`

Exception when handling PpLexer object.

**exception** `cpip.core.PpLexer.ExceptionPpLexerAlreadyGenerating`

Exception when two generators are created then the internal state will become inconsistent.

**exception** `cpip.core.PpLexer.ExceptionPpLexerCallStack`

Exception when finding issues with the call stack or nested includes.

**exception** `cpip.core.PpLexer.ExceptionPpLexerCallStackTooSmall`

Exception when `sys.getrecursionlimit()` is too small.

**exception** `cpip.core.PpLexer.ExceptionPpLexerCondLevelOutOfRange`

Exception when handling a conditional token generation level.

**exception** `cpip.core.PpLexer.ExceptionPpLexerDefine`

Exception when loading predefined macro definitions.

**exception** `cpip.core.PpLexer.ExceptionPpLexerNestedIncludeLimit`

Exception when nested `#include` limit exceeded.

**exception** `cpip.core.PpLexer.ExceptionPpLexerNoFile`

Exception when can not find file.

**exception** `cpip.core.PpLexer.ExceptionPpLexerPreInclude`

Exception when loading pre-include files.

**exception** `cpip.core.PpLexer.ExceptionPpLexerPreIncludeIncNoCp`

Exception when loading a pre-include file that has no current place (e.g. a StringIO object) and the pre-include then has an `#include` statement.

**exception** `cpip.core.PpLexer.ExceptionPpLexerPredefine`

Exception when loading predefined macro definitions.

`cpip.core.PpLexer.PREPROCESSING_DIRECTIVES = ['if', 'ifdef', 'ifndef', 'elif', 'else', 'endif', 'include', 'define', 'undefine']`

Allowable preprocessing directives

**class** `cpip.core.PpLexer.PpLexer` (*tuFileId*, *includeHandler*, *preIncFiles=None*, *diagnostic=None*, *pragmaHandler=None*, *stdPredefMacros=None*, *autoDefineDate-Time=True*, *gccExtensions=False*, *annotateLineFile=False*)

Create a translation unit tokeniser that applies *ISO/IEC 9899:1999(E) Section 6* and/or *ISO/IEC 14882:1998(E) section 16*.

**tuFileId** A file ID that will be given to the include handler to find the translation unit. Typically this will be the file path (as a string) to the file that is the Initial Translation Unit (ITU) i.e. the file being preprocessed.

**includeHandler** A handler to file `#include'd` files typically a `IncludeHandler.IncludeHandlerStd`. This might have user and system include path information and a means of resolving file references.

**preIncFiles** An ordered list of file like objects that are pre-include files. These are processed in order before the ITU is processed. Macro redefinition rules apply.

**diagnostic** A diagnostic object, defaults to a `CppDiagnostic.PreprocessDiagnosticStd`.

**pragmaHandler** A handler for #pragma statements.

This must have the attribute `replaceTokens` is to be implemented, if True then the tokens stream will be macro replaced before being passed to the pragma handler.

This must have a function `pragma()` defined that takes a non-zero length list of `PpToken.PpToken` the last of which will be a newline token. The tokens returned will be yielded.

**stdPredefMacros** A dictionary of Standard pre-defined macros. See for example: *ISO/IEC 9899:1999 (E) 6.10.8 Predefined macro names ISO/IEC 14882:1998 (E) 16.8 Predefined macro names N2800=08-0310 16.8 Predefined macro names*

The macros `__DATE__` and `__TIME__` will be automatically updated to current locale date/time (see `autoDefineDateTime`).

**autoDefineDateTime** If True then the macros `__DATE__` and `__TIME__` will be automatically updated to current locale date/time. Mostly this is used for testing.

**gccExtensions** Support GCC extensions. Currently just `#include_next` is supported.

**annotateLineFile** - if True then **PpToken** will output line number and file as **cpp**. For example:

```
# 22 "/usr/include/stdio.h" 3 4
# 59 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
```

TODO: Set flags here rather than supplying them to a generator? This would make the API simply the ctor and `pptokens/next()`. Flags would be: `incWs` - Include whitespace tokens. `condLevel` - (0, 1, 2) thus:

**0: No conditionally compiled tokens. The `fileIncludeGraphRoot` will not have any information about conditionally included files.**

**1: Conditionally compiled tokens are generated but not from** conditionally included files. The `fileIncludeGraphRoot` will have a reference to a conditionally included file but not that included file's includes.

**2: Conditionally compiled tokens including tokens from conditionally** included files. The `fileIncludeGraphRoot` will have all the information about conditionally included files recursively.

**CALL\_STACK\_DEPTH\_ASSUMED\_PPTOKENS = 10**

Each include The call stack depth,  $D = A + B + C * L$  Where L is the number of levels of nested includes and A is the call stack A above:

**CALL\_STACK\_DEPTH\_FIRST\_INCLUDE = 3**

B above:

**CALL\_STACK\_DEPTH\_PER\_INCLUDE = 3**

C above:

**COND\_LEVEL\_DEFAULT = 0**

Conditionality settings for token generation

**COND\_LEVEL\_OPTIONS = range(0, 3)**

Conditionality level (0, 1, 2)

**MAX\_INCLUDE\_DEPTH = 200**

The maximum value of nested #include's

**colNum**

Returns the current column number as an integer during processing.

**condCompGraph**

The conditional compilation graph as a *CppCond.CppCondGraph* object.

**condState**

The conditional state as (boolean, string).

**currentFile**

Returns the file ID on the top of the file stack.

**definedMacros**

Returns a string representing the currently defined macros.

**fileIncludeGraphRoot**

Returns the *FileIncludeGraph.FileIncludeGraphRoot* object.

**fileLineCol**

Returns a FileLineCol object or None

**fileName**

Returns the current file name during processing.

**fileStack**

Returns the file stack.

**finalise()**

Finalisation, may raise any Exception.

**includeDepth**

Returns the integer depth of the include stack.

**lineNum**

Returns the current line number as an integer during processing or None.

**macroEnvironment**

The current Macro environment as a *MacroEnv.MacroEnv* object.

**Caution:** Write to this at your own risk. Your write might be ignored or cause undefined behaviour.

**ppTokens** (*incWs=True, minWs=False, condLevel=0*)

A generator for providing a sequence of *PpToken.PpToken* in accordance with section 16 of *ISO/IEC 14882:1998(E)*.

*incWs* - if True than whitespace tokens are included (i.e. `tok.isWs() == True`).

*minWs* - if True then whitespace runs will be minimised to a single space or, if newline is in the whitespace run, a single newline

*condLevel* - if !=0 then conditionally compiled tokens will be yielded and they will have `tok.isCond == True`. The *fileIncludeGraphRoot* will be marked up with the appropriate conditionality. Levels are:

```
0: No conditionally compiled tokens. The fileIncludeGraphRoot will
not have any information about conditionally included files.

1: Conditionally compiled tokens are generated but not from
conditionally included files. The fileIncludeGraphRoot will have
a reference to a conditionally included file but not that
included file's includes.

2: Conditionally compiled tokens including tokens from conditionally
```

```
included files. The fileIncludeGraphRoot will have all the
information about conditionally included files recursively.
```

(see `_cppInclude` where we check if `self._condStack.isTrue()`).

#### **tuFileId**

Returns the user supplied ID of the translation unit.

`cpip.core.PpLexer.UNNAMED_FILE_NAME = 'Unnamed Pre-include'`

Used when file objects have no name

## PpToken

Represents a preprocessing Token in C/C++ source code.

`cpip.core.PpToken.ENUM_NAME = {0: 'header-name', 1: 'identifier', 2: 'pp-number', 3: 'character-literal', 4: 'string-literal'}`

Map of {integer : PREPROCESS\_TOKEN\_TYPE, ...} So this can be used thus:

```
if ENUM_NAME[token_type] == 'header-name':
```

**exception** `cpip.core.PpToken.ExceptionCpipToken`

Used by *PpToken*.

**exception** `cpip.core.PpToken.ExceptionCpipTokenIllegalMerge`

Used by *PpToken* when *PpToken.merge()* is called illegally.

**exception** `cpip.core.PpToken.ExceptionCpipTokenIllegalOperation`

Used by *PpToken* when an illegal operation is performed.

**exception** `cpip.core.PpToken.ExceptionCpipTokenReopenForExpansion`

Used by *PpToken* when a non-expandable token is made available for expansion.

**exception** `cpip.core.PpToken.ExceptionCpipTokenUnknownType`

Used by *PpToken* when the token type is out of range.

`cpip.core.PpToken.LEX_PPTOKEN_TYPES = ['header-name', 'identifier', 'pp-number', 'character-literal', 'string-literal']`

Types of preprocessing-token From: *ISO/IEC 14882:1998(E) 2.4 Preprocessing tokens [lex.pptoken]* and *ISO/IEC 9899:1999(E) 6.4.7 Header names* .. note:

```
Para 3 of the latter says that: "A header name preprocessing token is
recognized only within a ``#include`` preprocessing directive."
```

```
So in other contexts a header-name that is a q-char-sequence should be treated
as a string-literal
```

This produces interesting issues in this case:

```
#define str(s) # s
#include str(foo.h)
```

The stringise operator creates a string-literal token but the `#include` directive expects a header-name. So in certain contexts (macro stringising followed by `#include` instruction) we need to 'downcast' a string-literal to a header-name.

See *`cpip.core.PpLexer.PpLexer`* for how this is done

`cpip.core.PpToken.LEX_PPTOKEN_TYPE_ENUM_RANGE = range(0, 9)`

Range of allowable enum values



`cpip.core.PpToken.NAME_ENUM = {'preprocessing-op-or-punc': 5, 'character-literal': 3, 'concat': 8, 'pp-number': 2, 'header-name': 1}`  
 Map of {PREPROCESS\_TOKEN\_TYPE : integer, ...} So this can be used thus:

```
self._cppTokType = NAME_ENUM['header-name']
```

**class** `cpip.core.PpToken.PpToken(t, tt, lineNum=0, colNum=0, isReplacement=False)`

Holds a preprocessor token, its type and whether the token can be replaced.

`t` is the token (a string) and `tt` is either an enumerated integer or a string. Internally `tt` is stored as an enumerated integer. If the token is an identifier then it is eligible for replacement unless marked otherwise.

**SINGLE\_SPACE** = ' '

Representation of a single whitespace

**WORD\_REPLACE\_MAP** = {'false': 'False', '/': '//', 'true': 'True', '&&': 'and', '||': 'or'}

Operators that are replaced directly by Python equivalents for constant evaluation

**canReplace**

Flag to control whether this token is eligible for replacement

**colNum**

Returns the column number of the start of the token as an integer.

**copy()**

Returns a shallow copy of `self`. This is useful where the same token is added to multiple lists and then a `merge()` operation on one list will be seen by the others. To avoid this insert `self.copy()` in all but one of the lists.

**evalConstExpr()**

Returns a string value suitable for eval'ing in a constant expression. For numbers this removes such tiresome trivia as 'u', 'L' etc. For others it replaces '&&' with 'and' and so on.

See *ISO/IEC 14882:1998(E) 16.1 Conditional inclusion sub-section 4* i.e. section 16.1-4

and: *ISO/IEC 9899:1999 (E) 6.10.1 Conditional inclusion sub-section 3* i.e. section 6.10.1-3

**getIsReplacement()**

Gets the flag that records that this token is the result of macro replacement

**getPrevWs()**

Gets the flag that records prior whitespace.

**getReplace()**

Gets the flag that controls whether this can be replaced.

**isCond**

Flag that if True indicates that the token appeared within a section that was conditionally compiled. This is False on construction and can only be set True by `setIsCond()`

**isIdentifier()**

Returns True if the token type is 'identifier'.

**isReplacement**

Flag that records that this token is the result of macro replacement

**isUnCond**

Flag that if True indicates that the token appeared within a section that was un-conditionally compiled. This is the negation of `isCond`.

**isWs()**

Returns True if the token type is 'whitespace'.

**lineNum**

Returns the line number of the start of the token as an integer.

**merge** (*other*)

This will merge by appending the other token if they are different token types the type becomes 'concat'.

**prevWs**

Flag to indicate whether this token is preceded by whitespace

**replaceNewLine** ()

Replace any newline with a single whitespace character in-place.

See: *ISO/IEC 9899:1999(E) 6.10-3 and C++ ISO/IEC 14882:1998(E) 16.3-9*

This will raise a *ExceptionCpipTokenIllegalOperation* if I am not a whitespace token.

**setIsCond** ()

Sets self.\_isCond to be True.

**setIsReplacement** (*val*)

Sets the flag that records that this token is the result of macro replacement.

**setPrevWs** (*val*)

Sets the flag that records prior whitespace.

**setReplace** (*val*)

Setter, will raise if I am not an identifier or val is True and if I am otherwise not expandable.

**shrinkWs** ()

Replace all whitespace with a single ' '

This will raise a *ExceptionCpipTokenIllegalOperation* if I am not a whitespace token.

**subst** (*t, tt*)

Substitutes token value and type.

**t**

Returns the token as a string.

**tokenEnumToktype**

Returns the token and the enumerated token type as a tuple.

**tokToktype**

Returns the token and the token type (as a string) as a tuple.

**tt**

Returns the token type as a string.

`cpip.core.PpToken.tokensStr` (*theTokens, shortForm=True*)

Given a list of tokens this returns them as a string. If shortForm is True then the lexical string is returned. If False then the *PpToken* representations separated by '|' is returned. e.g. `PpToken(t="f", tt=identifier, line=True, prev=False, ?=False) | ...`

## PpTokenCount

Keeps a count of Preprocessing tokens.

**exception** `cpip.core.PpTokenCount.ExceptionPpTokenCount`

Exception when handling PpTokenCount object.

**exception** `cpip.core.PpTokenCount.ExceptionPpTokenCountStack`

Exception when handling PpTokenCountStack object.

---

```

class cpip.core.PpTokenCount.PpTokenCount
    Maps of {token_type : integer_count, ...} self._cntrTokAll is all tokens.

    __iadd__ (other)
        In-place add of the contents of another PpTokenCount object.

    __weakref__
        list of weak references to the object (if defined)

    inc (tok, isUnCond, num=1)
        Increment the count. tok is a PpToken, isUnCond is a boolean that is True if this is not conditionally
        compiled. num is the number of tokens to increment.

    tokenCount (theType, isAll)
        Returns the token count of a particular type. If isAll is true then the count of all tokens is returned, if False
        the count of unconditional tokens is returned.

    tokenCountNonWs (isAll)
        Returns the token count of a particular type. If isAll is true then the count of all tokens is returned, if False
        the count of unconditional tokens is returned.

    tokenTypesAndCounts (isAll, allPossibleTypes=True)
        Generator the yields (type, count) in PpToken.LEX_PPTOKEN_TYPES order where type is a
        string and count an integer.

        If isAll is true then the count of all tokens is returned, if False the count of unconditional tokens is returned.

        If allPossibleTypes is True the counts of all token types are yielded even if zero, if False then only token
        types encountered will be yielded i.e. all counts will be non-zero.

    totalAll
        The total token count.

    totalAllConditional
        The token count of conditional tokens.

    totalAllUnconditional
        The token count of unconditional tokens.

class cpip.core.PpTokenCount.PpTokenCountStack
    This simply holds a stack of PpTokenCount objects that can be created and popped of the stack.

    __init__ ()
        ctor with empty stack.

    __weakref__
        list of weak references to the object (if defined)

    close ()
        Finalisation, will raise a ExceptionPpTokenCountStack if there is anything on the stack.

    counter ()
        Returns a reference to the current PpTokenCount object.

    pop ()
        Pops the current PpTokenCount object off the stack and returns it.

    push ()
        Add a new counter object to the stack.

```

## PpTokeniser

Performs translation phases 0, 1, 2, 3 on C/C++ source code.

Translation phases from *ISO/IEC 9899:1999 (E)*:

5.1.1.2 Translation phases 5.1.1.2-1 The precedence among the syntax rules of translation is specified by the following phases.

Phase 1. Physical source file multibyte characters are mapped, in an implementation defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences are replaced by corresponding single-character internal representations.

Phase 2. Each instance of a backslash character ( ) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place.

Phase 3. The source file is decomposed into preprocessing tokens<sup>6)</sup> and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

TODO: Do phases 0,1,2 as generators i.e. not in memory?

TODO: Check coverage with a complete but minimal example of every token

TODO: remove self.\_cppTokType and have it as a return value?

TODO: Remove commented out code.

TODO: Performance of phase 1 processing.

TODO: rename next() as genPpTokens()?

TODO: Perf rewrite slice functions to take an integer argument of where in the array to start inspecting for a slice. This avoids calls to ...[x:] e.g. myCharS = myCharS[sliceIdx:] in genLexPptokenAndSeqWs.

```
cpip.core.PpTokeniser.COMMENT_REPLACEMENT = ‘ ‘  
    Comments are replaced by a single space
```

```
cpip.core.PpTokeniser.C_KEYWORDS = (‘auto’, ‘break’, ‘case’, ‘char’, ‘const’, ‘continue’, ‘default’, ‘do’, ‘double’, ‘else’,  
    ISO/IEC 9899:1999 (E) 6.4.1 Keywords
```

```
cpip.core.PpTokeniser.DIGRAPH_TABLE = {‘>’: ‘]’, ‘%: %’: ‘##’, ‘and_eq’: ‘&=’, ‘bitor’: ‘|’, ‘%:’: ‘#’, ‘compl’: ‘~’, ‘  
    Map of Digraph alternates
```

```
exception cpip.core.PpTokeniser.ExceptionCpipTokeniser  
    Simple specialisation of an exception class for the preprocessor.
```

```
exception cpip.core.PpTokeniser.ExceptionCpipTokeniserUcnConstraint  
    Specialisation for when universal character name exceeds constraints.
```

```
cpip.core.PpTokeniser.LEN_SOURCE_CHARACTER_SET = 96  
    Size of the source code character set
```

```
class cpip.core.PpTokeniser.PpTokeniser (theFileObj=None, theFileId=None, theDiagnos-  
    tic=None)  
    Imitates a Preprocessor that conforms to ISO/IEC 14882:1998(E).
```

Takes an optional file like object. If theFileObj has a ‘name’ attribute then that will be use as the name otherwise theFileId will be used as the file name.

**Implementation note:** On all `_slice...()` and `__slice...()` functions: A `_slice...()` function takes a buffer-like object and an integer offset as arguments. The buffer-like object will be accessed by index

so just needs to implement `__getitem__()`. On overrun or other out of bounds index an `IndexError` must be caught by the `_slice...()` function. i.e. `len()` should not be called on the buffer-like object, or rather, if `len()` (i.e. `__len__()`) is called a `TypeError` will be raised and propagated out of this class to the caller.

`StrTree`, for example, conforms to these requirements.

The function is expected to return an integer that represents the number of objects that can be consumed from the buffer-like object. If the return value is non-zero the `PpTokeniser` is side-affected in that `self._cppTokType` is set to a non-None value. Before doing that a test is made and if `self._cppTokType` is already non-None then an assertion error is raised.

The buffer-like object should not be side-affected by the `_slice...()` function regardless of the return value.

So a `_slice...()` function pattern is:

```
def _slice...(self, theBuf, theOfs):
    i = theOfs
    try:
        # Only access theBuf with [i] so that __getitem__() is called
        ...theBuf[i]...
        # Success as the absence of an IndexError!
        # So return the length of objects that pass
        # First test and set for type of slice found
        if i > theOfs:
            assert(self._cppTokType is None), '_cppTokType was %s now %s' % (self.
→_cppTokType, ...)
            self._cppTokType = ...
            # NOTE: Return size of slice not the index of the end of the slice
            return i - theOfs
        except IndexError:
            pass
        # Here either return 0 on IndexError or i-theOfs
        return ...
```

NOTE: Functions starting with `__slice...` do not trap the `IndexError`, the caller must do that.

TODO: ISO/IEC 14882:1998(E) Escape sequences Table 5?

#### **cppTokType**

Returns the type of the last preprocessing-token found by `_sliceLexPptoken()`.

#### **fileLineCol**

Return an instance of `FileLineCol` from the current physical line column.

#### **fileLocator**

Returns the `FileLocation` object.

#### **fileName**

Returns the ID of the file.

#### **filterHeaderNames** (*theToks*)

Returns a list of 'header-name' tokens from the supplied stream. May raise `ExceptionCpipTokeniser` if un-parsable or theToks has non-(whitespace, header-name).

#### **genLexPptokenAndSeqWs** (*theCharS*)

Generates a sequence of `PpToken` objects. Either:

- a sequence of whitespace (comments are replaces with a single whitespace).
- a pre-processing token.

This performs translation phase 3.

NOTE: Whitespace sequences are not merged so ' /\\*\\*/ ' will generate three tokens each of `PpToken.PpToken(' ', 'whitespace')` i.e. leading whitespace, comment replced by single space, trailing whitespace.

So this yields the tokens from translation phase 3 if supplied with the results of translation phase 2.

NOTE: This does not generate 'header-name' tokens as these are context dependent i.e. they are only valid in the context of a `#include` directive.

*ISO/IEC 9899:1999 (E) 6.4.7 Header names Para 3* says that: "A header name preprocessing token is recognised only within a `#include` preprocessing directive."

#### **initLexPhase12 ()**

Process phases one and two and returns the result as a string.

#### **lexPhases\_0 ()**

An non-standard phase that just reads the file and returns its contents as a list of lines (including EOL characters). May raise an `ExceptionCpipTokeniser` if self has been created with `None` or the file is unreadable

#### **lexPhases\_1 (theLineS)**

*ISO/IEC 14882:1998(E) 2.1 Phases of translation [lex.phases] - Phase one* Takes a list of lines (including EOL characters), replaces trigraphs and returns the new list of lines.

#### **lexPhases\_2 (theLineS)**

*ISO/IEC 14882:1998(E) 2.1 Phases of translation [lex.phases] - Phase two* This joins physical to logical lines. NOTE: This side-effects the supplied lines and returns `None`.

#### **next ()**

The token generator. On being called this performs translations phases 1, 2 and 3 (unless already done) and then generates pairs of: (preprocessing token, token type) Token type is an enumerated integer from `LEX_PPTOKEN_TYPES`. Proprocessing tokens include sequences of whitespace characters and these are not necessarily concatenated i.e. this generator can produce more than one whitespace token in sequence. TODO: Rename this to `ppTokens()` or something

#### **pLineCol**

Returns the current physical (line, column) as integers.

#### **reduceToksToHeaderName (theToks)**

This takes a list of `PpTokens` and retuns a list of `PpTokens` that might have a header-name token type in them. May raise an `ExceptionCpipTokeniser` if tokens are not all consumed. This is used at lexer level for re-interpreting `PpTokens` in the context of a `#include` directive.

#### **resetTokType ()**

Erases the memory of the previously seen token type.

#### **substAltToken (tok)**

If a `PpToken` is a Digraph this alters its value to its alternative. If not the supplied token is returned unchanged. There are no side effects on self.

```
cpip.core.PpTokeniser.TRIGRAPH_PREFIX = '?'
```

Note: This is redoubled

```
cpip.core.PpTokeniser.TRIGRAPH_SIZE = 3
```

Well it is a Trigraph

```
cpip.core.PpTokeniser.TRIGRAPH_TABLE = {'(': '[', '"': '^', '-': '~', '!': '|', '/': '\\', '>': '}', '<': '{', ')': ']', '=': '#'}
Map of Trigraph alternates after the ?? prefix
```

## PpWhitespace

Understands whitespacey things about source code character streams.

```

cpip.core.PpWhitespace.DDEFINE_WHITESPACE = {'\n', '\t', ' '
    Whitespace characters that are significant in define statements ISO/IEC 14882:1998(E) 16-2 only ' ' and 't' as
    ws

cpip.core.PpWhitespace.LEN_WHITESPACE_CHARACTER_SET = 5
    Number of whitespace characters

cpip.core.PpWhitespace.LEX_NEWLINE = '\n'
    Whitespace newline

cpip.core.PpWhitespace.LEX_WHITESPACE = {'\n', '\x0c', '\t', ' ', '\x0b'}
    Whitespace characters

class cpip.core.PpWhitespace.PpWhitespace
    A class that does whitespacey type things in accordance with ISO/IEC 9899:1999(E) Section 6 and ISO/IEC
    14882:1998(E).

    hasLeadingWhitespace(theCharS)
        Returns True if any leading whitespace, False if zero length or starts with non-whitespace.

    isAllMacroWhitespace(theCharS)
        "Return True if theCharS is zero length or only has allowable whitespace for preprocessing macros.
        ISO/IEC 14882:1998(E) 16-2 only ' ' and ' ' as whitespace.

    isAllWhitespace(theCharS)
        Returns True if the supplied string is all whitespace.

    isBreakingWhitespace(theCharS)
        Returns True if whitespace leads theChars and that whitespace contains a newline.

    preceedsNewline(theCharS)
        Returns True if theChars ends with a newline. i.e. this immediately precedes a new line.

    sliceNonWhitespace(theBuf, theOfs=0)
        Returns the length of non-whitespace characters that are in theBuf from position theOfs.

    sliceWhitespace(theBuf, theOfs=0)
        Returns the length of whitespace characters that are in theBuf from position theOfs.

```

## PragmaHandler

```

exception cpip.core.PragmaHandler.ExceptionPragmaHandler
    Simple specialisation of an exception class for the PragmaHandler. If raised this will cause the PpLexer to
    register undefined behaviour.

exception cpip.core.PragmaHandler.ExceptionPragmaHandlerStopParsing
    Exception class for the PragmaHandler to stop parsing token stream.

class cpip.core.PragmaHandler.PragmaHandlerABC
    Abstract base class for a pragma handler.

    isLiteral
        Treat the result of pragma() literally so no further processing required.

    pragma(theTokS)
        Takes a list of PpTokens, processes then and should return a newline terminated string that will be prepro-
        cessed in the current environment.

    replaceTokens
        An boolean attribute that says whether the supplied tokens should be macro replaced before being passed
        to self.

```

**class** `cpip.core.PragmaHandler.PragmaHandlerEcho`

A pragma handler that retains the `#pragma` line verbatim.

**isLiteral**

This class is just going to echo the line back complete with the `'#pragma'` prefix. If the PpLexer re-interpreted this it would be an infinite loop.

**pragma** (*theTokS*)

Consume and return.

**replaceTokens**

Tokens do not require macro replacement.

**class** `cpip.core.PragmaHandler.PragmaHandlerNull`

A pragma handler that does nothing.

**pragma** (*theTokS*)

Consume and return.

**replaceTokens**

Tokens do not require macro replacement.

**class** `cpip.core.PragmaHandler.PragmaHandlerSTDC`

Base class for a pragma handler that implements ISO/IEC 9899:1999 (E) 6.10.5 Error directive para. 2.

**DIRECTIVES** = ('FP\_CONTRACT', 'FENV\_ACCESS', 'CX\_LIMITED\_RANGE')

Standard C acceptable macro directives

**ON\_OFF\_SWITCH\_STATES** = ('ON', 'OFF', 'DEFAULT')

Standard C macro states

**STDC** = 'STDC'

Standard C macro

**pragma** (*theTokS*)

Inject a macro declaration into the environment.

See ISO/IEC 9899:1999 (E) 6.10.5 Error directive para. 2.

**replaceTokens**

STDC lines do not require macro replacement.

## cpip.util

### BufGen

A generator class with a buffer. This allows multiple inspections of the stream issued by a generator. For example this is used by MaxMunchGen.

**class** `cpip.util.BufGen.BufGen` (*theGen*)

A generator class with a buffer.

**gen** ()

Yield objects from the generator via the buffer.

**lenBuf**

Returns the length of the existing buffer. NOTE: This may not be the final length as the generator might not be exhausted just yet.



**replace** (*theIdx, theLen, theValueS*)

Replaces within the buffer starting at theIdx removing theLen objects and replacing them with theValueS.

**slice** (*sliceLen*)

Returns a buffer slice of length sliceLen.

**exception** `cpip.util.BufGen.ExceptionBufGen`

Exception specialisation for BufGen.

## CommonPrefix

Created on 23 Feb 2014

@author: paulross

`cpip.util.CommonPrefix.lenCommonPrefix` (*iterable*)

Returns the length of the common prefix of a list of file names. The prefix is limited to directory names.

## DictTree

A dictionary that takes a list of hashables as a key and behaves like a tree.

**class** `cpip.util.DictTree.DictTree` (*valIterable=None*)

A dictionary that takes a list of hashables as a key and behaves like a tree

**add** (*k, v*)

Add a key/value. k is a list of hashables.

**depth** ()

Returns the maximum tree depth as an integer.

**keys** ()

Return a list of keys where each key is a list of hashables.

**remove** (*k, v=None*)

Remove a key/value. k is a list of hashables.

**value** (*k*)

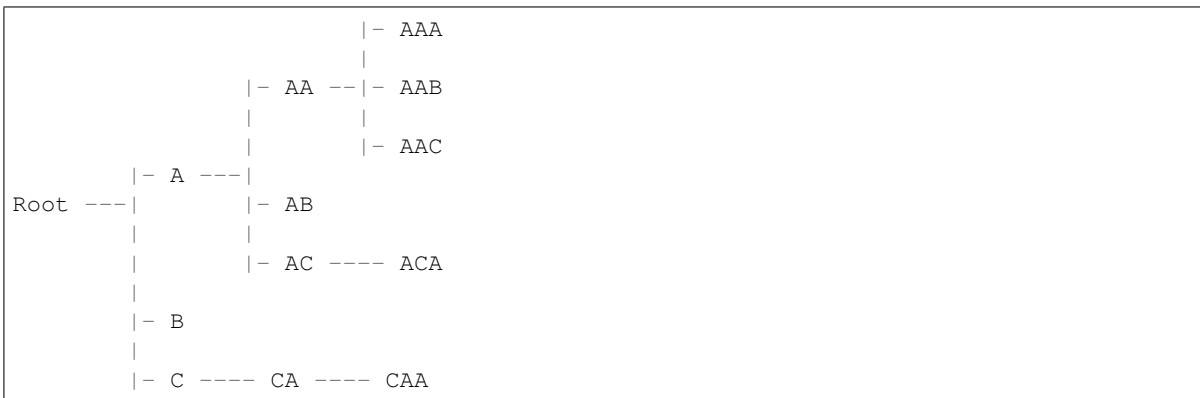
Value corresponding to a key or None. k is a list of hashables.

**values** ()

Returns a list of all values.

**class** `cpip.util.DictTree.DictTreeHtmlTable` (*\*args*)

A sub-class of DictTree that helps writing HTML row/col span tables Suppose we have a tree like this:



And we want to represent the tree like this when laid out as an HTML table:

-----		
A	AA	AAA
		-----
		AAB
		-----
		AAC
	-----	
	AB	
	-----	
	AC	ACA
-----		
B		
-----		
C	CA	CAA
-----		

In this example the tree is loaded branch by branch thus:

```
myTree = DictTreeHtmlTable()
myTree.add(('A', 'AA', 'AAA'), None)
myTree.add(('A', 'AA', 'AAB'), None)
myTree.add(('A', 'AA', 'AAC'), None)
myTree.add(('A', 'AB',), None)
myTree.add(('A', 'AC', 'ACA'), None)
myTree.add(('B',), None)
myTree.add(('C', 'CA', 'CAA'), None)
```

The HTML code generator can be used like this:

```
# Write: <table border="2" width="100%">
for anEvent in myTree.genColRowEvents():
    if anEvent == myTree.ROW_OPEN:
        # Write out the '<tr>' element
    elif anEvent == myTree.ROW_CLOSE:
        # Write out the '</tr>' element
    else:
        k, v, r, c = anEvent
        # Write '<td rowspan="%d" colspan="%d">%s</td>' % (r, c, v)
# Write: </table>
```

And the HTML code will look like this:

```
<table border="2" width="100%">
  <tr valign="top">
    <td rowspan="5">A</td>
    <td rowspan="3">AA</td>
    <td>AAA</td>
  </tr>
  <tr valign="top">
    <td>AAB</td>
  </tr>
  <tr valign="top">
    <td>AAC</td>
  </tr>
  <tr valign="top">
    <td colspan="2">AB</td>
  </tr>
```

```

<tr valign="top">
  <td>AC</td>
  <td>ACA</td>
</tr>
<tr valign="top">
  <td colspan="3">B</td>
</tr>
<tr valign="top">
  <td>C</td>
  <td>CA</td>
  <td>CAA</td>
</tr>
</table>

```

**genColRowEvents ()**

Returns a set of events that are quadruples. (key\_branch, value, rowspan\_int, colspan\_int) The branch is a list of keys the from the branch of the tree. The rowspan and colspan are both integers. At the start of the a <tr> there will be a ROW\_OPEN and at row end (</tr> a ROW\_CLOSE will be yielded

**setColRowSpan ()**

Top level call that sets colspan and rowspan attributes.

**exception cpip.util.DictTree.ExceptionDictTree**

Exception when handling a DictTree object.

## DirWalk

Provides various ways of walking a directory tree

Created on Jun 9, 2011

**exception cpip.util.DirWalk.ExceptionDirWalk**

Exception class for this module.

**class cpip.util.DirWalk.FileInOut (filePathIn, filePathOut)**

A pair of (in, out) file paths

**filePathIn**

Alias for field number 0

**filePathOut**

Alias for field number 1

**cpip.util.DirWalk.dirWalk** (*theIn*, *theOut*=None, *theFnMatch*=None, *recursive*=False, *bigFirst*=False)

Walks a directory tree generating file paths.

**theIn** The input directory.

**theOut** The output directory. If None then input file paths as strings will be generated If non-None this function will yield FileInOut(in, out) objects. NOTE: This does not create the output directory structure, it is up to the caller to do that.

**theFnMatch** A glob like match pattern for file names (not tested for directory names). Can be a list of strings any of which can match. If None or empty list then all files match.

**recursive** Boolean to recurse into directories or not.

**bigFirst** If True then the largest files in directory are given first. If False it is alphabetical.

`cpip.util.DirWalk.genBigFirst` (*d*)

Generator that yields the biggest files (name not path) first. This is fairly simple in that it only looks the current directory not only sub-directories. Useful for multiprocessing.

## HtmlUtils

HTML utility functions.

`cpip.util.HtmlUtils.pathSplit` (*p*)

Split a path into its components.

`cpip.util.HtmlUtils.retHtmlFileLink` (*theSrcPath*, *theLineNum*)

Returns a string that is a link to a HTML file.

***theSrcPath* : str** The path of the original source, whis will be encoded with `retHtmlFileName()`.

***theLineNum* : int** An integer line number in the target.

`cpip.util.HtmlUtils.retHtmlFileName` (*thePath*)

Creates a unique, short, human readable file name base on the input file path.

`cpip.util.HtmlUtils.writeCharsAndSpan` (*theS*, *theText*, *theSpan*)

Write theText to the stream theS. If theSpan is not None the text is enclosed in a `<span class=theSpan>` element.

***theS*** The XHTML stream.

***theText* : str** The text to write, must be non-empty.

***theClass* : str, optional** CSS class for the text.

`cpip.util.HtmlUtils.writeDictTreeAsTable` (*theS*, *theDt*, *tableAttrs*, *includeKeyTail*)

Writes a DictTreeHtmlTable object as a table, for example as a directory structure.

The key list in the DictTreeHtmlTable object is the path to the file i.e. `os.path.abspath(p).split(os.sep)` and the value is expected to be a pair of (*link*, *nav\_text*) or None.

`cpip.util.HtmlUtils.writeFileListAsTable` (*theS*, *theFileLinkS*, *tableAttrs*, *includeKeyTail*)

Writes a list of file names as an HTML table looking like a directory structure. *theFileLinkS* is a list of pairs (*file\_path*, *href*). The navigation text in the cell will be the basename of the *file\_path*.

`cpip.util.HtmlUtils.writeFileListTrippleAsTable` (*theS*, *theFileLinkS*, *tableAttrs*, *includeKeyTail*)

Writes a list of file names as an HTML table looking like a directory structure. *theFileLinkS* is a list of triples (*file\_name*, *href*, *nav\_text*).

`cpip.util.HtmlUtils.writeFilePathsAsTable` (*valueType*, *theS*, *theKvS*, *tableStyle*, *fnTd*, *fnTrTh=None*)

Writes file paths as a table, for example as a directory structure.

***valueType*** The type of the value: None, `'list'` | `'set'`

***theS*** The HTML stream.

***theKvS* : list** A list of pairs (*file\_path*, *value*).

***tableStyle* : str** The style used for the table.

***fnTd*** A callback function that is executed for a `<td>` element when there is a non-None value. This is called with the following arguments:

***theS*** The HTML stream.

***attrs* : dict** A map of attrs that include the rowspan/colspan for the `<td>`

***k*** : *list* The key as a list of path components.

***v*** The value given by the caller.

***fnTrTh*** Callback function for the header that will be called with the following arguments:

***theS*** The HTML stream.

***pathDepth*** Maximum depth of the largest path, this can be used for <th colspan="...">File path</th>.

```
cpip.util.HtmlUtils.writeHtmlFileAnchor(theS, theLineNum, theText='', theClass=None,
                                         theHref=None)
```

Writes an anchor.

***theS*** The XHTML stream.

***theLineNum*** : *int* An integer line number in the target.

***theText*** : *str, optional* Navigation text.

***theClass*** : *str, optional* CSS class for the navigation text.

***theHref*** : *str, optional* The href=.

```
cpip.util.HtmlUtils.writeHtmlFileLink(theS, theSrcPath, theLineNum, theText='', the-
                                       Class=None)
```

Writes a link to another HTML file that represents source code.

***theS*** The XHTML stream.

***theSrcPath*** : *str* The path of the original source, this will be encoded with retHtmlFileName().

***theLineNum*** : *int* An integer line number in the target.

***theText*** : *str, optional* Navigation text.

***theClass*** : *obj, optional* CSS class for the navigation text.

## ListGen

Treats a list as a generator with an optional additional generator. This is used for macro replacement for example.

```
class cpip.util.ListGen.ListAsGenerator(theList, theGen=None)
```

Class that takes a list and provides a generator on that list. If the list is exhausted and call for another object is made then it is pulled of the generator (if available).

The attribute `listIsEmpty` is True if the immediate list is empty.

Iterating through the result and stopping when the list is exhausted using the flag `listIsEmpty`:

To be clear: when this flag is set, for example if we have a list [0,1,2,3] followed by ['A', 'B', 'C'] thus:

```
myObj = ListAsGenerator(range(3), ListAsGenerator(list('ABC')).next())
```

And we try to iterate over it with list comprehension:

```
myGen = myObj.next()
myResult = [x for x in myGen if not myObj.listIsEmpty]
```

`myResult` will be [0, 1,] because when 3 is yielded the flag is False as it refers to the `_next_` item.

Similarly the list comprehension:

```
myResult = [x for x in myGen if myObj.listIsEmpty]
```

Will be [3, 'A', 'B', 'C']

If you want to recover the then this the technique:

```
myResult = []
if not myObj.listIsEmpty:
    for aVal in myGen:
        myResult.append(aVal)
        if myObj.listIsEmpty:
            break
```

Or exclude the list then this the technique:

```
if not myObj.listIsEmpty:
    for aVal in myGen:
        if myObj.listIsEmpty:
            break
myResult = [x for x in myGen]
```

The rationale for this behaviour is for generating macro replacement tokens in that the list contains tokens for re-examination and the last token may turn out to be a function like macro that needs the generator to (possibly) complete the expansion. Once that last token has been re-examined we do not want to consume any more tokens than necessary.

#### **listIsEmpty**

True if the next yield would come from the generator, not the list.

#### **next ()**

yield the next value. The attribute listIsEmpty will be set True immediately before yielding the last value.

## MatrixRep

Makes replacements in a list of lines.

**exception** `cpip.util.MatrixRep.ExceptionMatrixRep`

Simple specialisation of an exception class for MatrixRep.

**class** `cpip.util.MatrixRep.MatrixRep`

Makes replacements in a list of lines.

**addLineColRep** (*l, c, was, now*)

Adds to the IR. No test is made to see if there is an existing or pre-existing conflicting entry or if a sequence of entries makes sense. It is expected that callers call this in line/column order of the original matrix. If not the results of a subsequent call to `sideEffect()` are undefined.

**sideEffect** (*theMat*)

Makes the replacement, if line/col is out of range and `ExceptionMatrixRep` will be raised and the state of the *theMat* argument is undefined.

## MaxMunchGen

Generic Maximal Munch generator.

**exception** `cpip.util.MaxMunchGen.ExceptionMaxMunchGen`

Exception specialisation for MaxMunchGen.

**class** `cpip.util.MaxMunchGen.MaxMunchGen` (*theGen*, *theFnS*, *isExclusive=False*, *yieldReplacement=False*)

Provides a generator that applies Maximal munch rules.

**gen** ()

Yields a maximal munch. If `yieldReplacement` is `False` these will be pairs of (iterable, kind) where kind is from the function, any replacement will be done on the fly. If `yieldReplacement` is `True` these will be triples of (iterable, kind, repl) where kind and repl are from the function with repl being `None` if no replacement. No replacement will have been done.

TODO: Reconsider this design. Really `yieldReplacement` decides if the underlying generator buffer contains the replacement rather than whether self yields the replacement.

`cpip.util.MaxMunchGen.anyToken` (*theGen*)

A function that always reads one token. This can be used as the last registered function to ensure that the token stream is read to completion. The kind returned is `None`.

## OaS

Various utility functions etc. that don't obviously fit elsewhere.

**exception** `cpip.util.OaS.ExceptionOaS`

Simple specialisation of an exception class for this module.

`cpip.util.OaS.indexLB` (*l*, *v*)

Returns the lower bound index in a sorted list *l* of the value that is equal to *v* or the nearest lower value to *v*. Returns -1 if *l* empty or all values higher than *v*.

`cpip.util.OaS.indexMatch` (*l*, *v*)

Returns the index of *v* in sorted list *l* or -1. This uses Jon Bentley's binary search algorithm. This uses operators `>` and `<`.

`cpip.util.OaS.indexUB` (*l*, *v*)

Returns the upper bound index in a sorted list *l* of the value that is equal to *v* or the nearest upper value to *v*. Returns -1 if *l* empty or all values lower than *v*.

## StrTree

Treats a string as a tree.

**class** `cpip.util.StrTree.StrTree` (*theIterable=None*)

Initialise the class with a optional list of strings.

**add** (*s*)

Add a string.

**has** (*s*, *i=0*)

Returns the index of the end of *s* that match a complete word in the tree. i.e. [*i*:return\_value] is in the dictionary. Note `IndexError` and `KeyError` are trapped here.

**values** ()

Returns all values.

## Tree

Represents a simple tree.

Created on 6 Mar 2014

@author: paulross

**class** `cpip.util.Tree.DuplexAdjacencyList`

Represents a set of parent/child relationships (and their inverse) as Adjacency Lists.

**allChildren**

Returns an unordered list of objects that have at least one parent.

**allParents**

Returns an unordered list of objects that have at least one child.

**children** (*parent*)

Returns all immediate children of a given parent.

**hasChild** (*child*)

Returns True if the given child has any parents.

**hasParent** (*parent*)

Returns True if the given parent has any children.

**parents** (*child*)

Returns all immediate parents of a given child.

**treeChildParent** (*theObj*)

Returns a Tree() object where the links are the relationships between child and parent. Cycles are not reproduced i.e. if a -> b and b -> c and c -> a then treeChildParent('a') returns ['a', 'c', 'b',] treeChildParent('b') returns ['b', 'a', 'c',] treeChildParent('c') returns ['c', 'b', 'a',]

**treeParentChild** (*theObj*)

Returns a Tree() object where the links are the relationships between parent and child. Cycles are not reproduced i.e. if a -> b and b -> c and c -> a then treeParentChild('a') returns ['a', 'b', 'c',] treeParentChild('b') returns ['b', 'c', 'a',] treeParentChild('c') returns ['c', 'a', 'b',]

**class** `cpip.util.Tree.Tree` (*obj*)

Represents a simple tree of objects.

**branches** ()

Returns all the possible branches through the tree as a list of lists of self.\_obj.

**youngestChild**

The latest child to be added, may raise IndexError if no children.

## XmlWrite

Writes XML and XHTML.

**class** `cpip.util.XmlWrite.Element` (*theXmlStream, theElemName, theAttrs=None*)

Represents an element in a markup stream.

**exception** `cpip.util.XmlWrite.ExceptionXml`

Exception specialisation for the XML writer.

**exception** `cpip.util.XmlWrite.ExceptionXmlEndElement`

Exception specialisation for end of element.

`cpip.util.XmlWrite.RAISE_ON_ERROR = True`

Global flag that sets the error behaviour If True then this module may raise an ExceptionXml and that might mask other exceptions. If False no ExceptionXml will be raised but a logging.error(...) will be written. These will not mask other Exceptions.



**class** `cpip.util.XmlWrite.XmlStream` (*theFout*, *theEnc*='utf-8', *theDtdLocal*=None, *theId*=0, *mustIndent*=True)

Creates and maintains an XML output stream.

**characters** (*theString*)

Encodes the string and writes it to the output.

**comment** (*theS*, *newLine*=False)

Writes a comment to the output stream.

**endElement** (*name*)

Ends an element.

**id**

A unique ID in this stream. The ID is incremented on each call.

**literal** (*theString*)

Writes theString to the output without encoding.

**PI** (*theS*)

Writes a Processing Instruction to the output stream.

**startElement** (*name*, *attrs*)

Opens a named element with attributes.

**writeCDATA** (*theData*)

Writes a CDATA section.

Example:

**writeCSS** (*theCSSMap*)

Writes a style sheet as a CDATA section. Expects a dict of dicts.

Example:

**writeECMAScript** (*theScript*)

Writes the ECMA script.

Example:

**xmlSpacePreserve** ()

Suspends indentation for this element and its descendants.

`cpip.util.XmlWrite.decodeString` (*theS*)

Returns a string that is the argument decoded. May raise a TypeError.

`cpip.util.XmlWrite.encodeString` (*theS*, *theCharPrefix*='\_')

Returns a string that is the argument encoded. RFC3548:

See section 3 of : <http://www.faqs.org/rfcs/rfc3548.html>

`cpip.util.XmlWrite.nameFromString` (*theStr*)

Returns a name from a string.

See <http://www.w3.org/TR/1999/REC-html401-19991224/types.html#type-cdata>

“ID and NAME tokens must begin with a letter ([A-Za-z]) and may be followed by any number of letters, digits ([0-9]), hyphens (“-”), underscores (“\_”), colons (“:”), and periods (“.”).

This also works for in namespaces as ‘:’ is not used in the encoding.

## cpip.plot

### Coord

#### Main Classes

Most classes in this module are `collections.namedtuple` objects.

Class	Description	Attributes
<code>Dim</code>	Linear dimension	value units
<code>Box</code>	A Box	width depth
<code>Pad</code>	Padding around a tree object	prev next, parent child
<code>Margin</code>	Padding around an object	left right top bottom
<code>Pt</code>	A point in Cartesian space	x y

#### Reference

**exception** `cpip.plot.Coord.ExceptionCoord`

Exception class for representing Coordinates.

**exception** `cpip.plot.Coord.ExceptionCoordUnitConvert`

Exception raised when converting units.

`cpip.plot.Coord.units()`

Returns the unsorted list of acceptable units.

`cpip.plot.Coord.convert(val, unitFrom, unitTo)`

Convert a value from one set of units to another.

**class** `cpip.plot.Coord.Dim`

Represents a dimension as an engineering value i.e. a number and units.

**scale** (*factor*)

Returns a new `Dim()` scaled by a factor, units are unchanged.

**convert** (*u*)

Returns a new `Dim()` with units changed and value converted.

**\_\_add\_\_** (*other*)

Overload self+other, returned result has the sum of self and other. The units chosen are self's unless self's units are None in which case other's units are used (if not None).

**\_\_sub\_\_** (*other*)

Overload self-other, returned result has the difference of self and other. The units chosen are self's unless self's units are None in which case other's units are used (if not None).

**\_\_iadd\_\_** (*other*)

Addition in place, value of other is converted to my units and added.

**\_\_isub\_\_** (*other*)

Subtraction in place, value of other is subtracted.

**\_\_lt\_\_** (*other*)

Returns true if self value < other value after unit conversion.

**\_\_le\_\_** (*other*)

Returns true if self value <= other value after unit conversion.

**\_\_eq\_\_** (*other*)  
Returns true if self value == other value after unit conversion.

**\_\_ne\_\_** (*other*)  
Returns true if self value != other value after unit conversion.

**\_\_gt\_\_** (*other*)  
Returns true if self value > other value after unit conversion.

**\_\_ge\_\_** (*other*)  
Returns true if self value >= other value after unit conversion.

**class** `cpip.plot.Coord.Pad`

Padding around another object that forms the Bounding Box. All 4 attributes are Dim() objects

**\_\_str\_\_** ()  
Stringifying.

**class** `cpip.plot.Coord.Pt`

A point, an absolute x/y position on the plot area. Members are Coord.Dim().

**\_\_eq\_\_** (*other*)  
Comparison.

**\_\_str\_\_** ()  
Stringifying.

**convert** (*u*)  
Returns a new Pt() with units changed and value converted.

**scale** (*factor*)  
Returns a new Pt() scaled by a factor, units are unchanged.

`cpip.plot.Coord.baseUnitsDim` (*theLen*)  
Returns a Coord.Dim() of length and units BASE\_UNITS.

`cpip.plot.Coord.zeroBaseUnitsDim` ()  
Returns a Coord.Dim() of zero length and units BASE\_UNITS.

`cpip.plot.Coord.zeroBaseUnitsBox` ()  
Returns a Coord.Box() of zero dimensions and units BASE\_UNITS.

`cpip.plot.Coord.zeroBaseUnitsPad` ()  
Returns a Coord.Pad() of zero dimensions and units BASE\_UNITS.

`cpip.plot.Coord.zeroBaseUnitsPt` ()  
Returns a Coord.Dim() of zero length and units BASE\_UNITS.

`cpip.plot.Coord.newPt` (*theP, incX=None, incY=None*)  
Returns a new Pt object by incrementing existing point incX, incY that are both Dim() objects or None.

`cpip.plot.Coord.convertPt` (*theP, theUnits*)  
Returns a new point with the dimensions of theP converted to theUnits.

## Examples

### `Coord.Dim()`

Creation, addition and subtraction:

```
d = Coord.Dim(1, 'in') + Coord.Dim(18, 'px')
# d is 1.25 inches
d = Coord.Dim(1, 'in') - Coord.Dim(18, 'px')
# d is 0.75 inches
d += Coord.Dim(25.4, 'mm')
# d is 1.75 inches
```

Scaling and unit conversion returns a new object:

```
a = Coord.Dim(12, 'px')
b = myObj.scale(6.0)
# b is 72 pixels
c = b.convert('in')
# 1 is 1 inch
```

Comparison:

```
assert(Coord.Dim(1, 'in') == Coord.Dim(72, 'px'))
assert(Coord.Dim(1, 'in') >= Coord.Dim(72, 'px'))
assert(Coord.Dim(1, 'in') <= Coord.Dim(72, 'px'))
assert(Coord.Dim(1, 'in') > Coord.Dim(71, 'px'))
assert(Coord.Dim(1, 'in') < Coord.Dim(73, 'px'))
```

## **Coord.Pt ()**

Creation:

```
p = Coord.Pt (
    Coord.Dim(12, 'px'),
    Coord.Dim(24, 'px'),
)
print(p)
# Prints: 'Pt(x=Dim(12px), y=Dim(24px))'
p.x # Coord.Dim(12, 'px')
p.y # Coord.Dim(24, 'px')
# Scale up by 6 and convert units
pIn = p.scale(6).convert('in')
# pIn now 'Pt(x=Dim(1in), y=Dim(2in))'
```

## **Testing**

The unit tests are in `test/TestCoord.py`.

## **PlotNode**

### **Bounding Boxes**

Legend for the drawing below:

```
**** - Self sigma BB.
~~~~ - Self pad box
#### - Self width and depth.
```

```
.... - All children
++++ - Child[n] sigma BB.
```

i.e. For a child its ++++ is equivalent to my \*\*\*\*.

Points in the drawing below:

- D - Self datum point.
- S - Self plot datum point.
- x[n] - Child datum point.
- Pl - Parent landing point to self.
- Pt - Parent take-off point from self.
- P[n] - Self take off point and landing point to child n.
- pl[n] - Child n landing point from self.
- pt[n] - Child n take-off point to self.
- tdc - Top dead centre.

Box .... has depth of `max(Boxes(++++) .width)` and width `max(Box(~~~~), sum(Boxes(++++) .depth))`.

Each instance of class knows about the following:

Boxes:

- \*\*\*\* - Self sigma BB as computed `Dim()` objects: `self.bbSigmaDepth` and `self.bbSigmaWidth`. Or as computed `Box()` object `self.bbSigma`
- ~~~~ - As computed `Dim()` objects: `self.bbSelfWidth`, `self.bbSelfDepth`
- ##### - Self width and depth as `Dim()` objects: `self.width` and `self.depth`
- .... - All children as a `Box()` object: `self.bbChildren`

And padding between ~~~~ and .... as `Dim()` object `self.bbSpaceChildren`

i.e. not ++++ - Child[n] sigma BB. That the caller knows about its children.

Points: given D each instance of this class knows:

```
S, Pl, Pt, P[0] to P[N-1], x[0], tdc (only).
```

In the following diagram where lines are adjacent that means that there is no spacing between them. This diagram shows the root at top left and the children from left to right. The default plot of the include graph is to have the root at top left with the processed file centre left with the children running from top to bottom. It is felt that this is more intuitive for source code.

```
-|-----> x increases
|
|
\ /
y increases

D *****
*                                     *
*          ~~~~~~*
*          ~~~~~~*
*          ~      S ### Pl ###tdc### Pt #####      ~      *
*          ~      #                                     #      ~      *
```

```

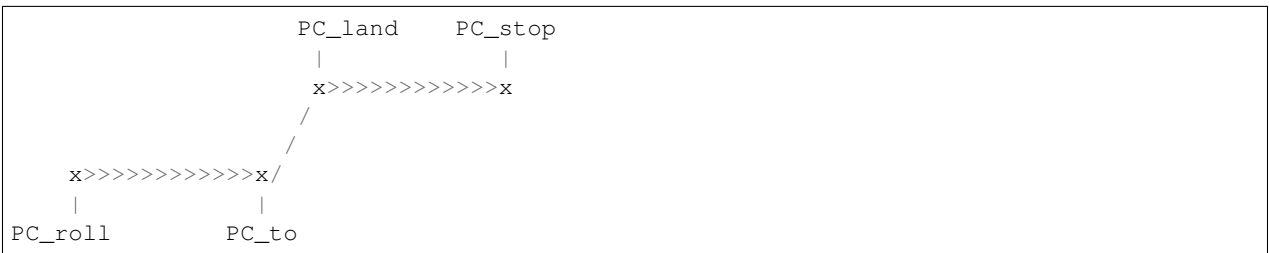
*          ~      #          #          ~          *
*          ~      #          #          ~          *
*          ~      #          #          ~          *
*          ~      ## P[0] ## P[c] ## P[C-1] ## ~          *
*          ~                                          ~          *
*          ~~~~~~^~~~~~                                          *
*                  | == self._bbSpaceChildren                                          *
*                  |                                          *
* .....*
*.x[0] + pl[0] + pt[0] +x[c] + pl[c] + pt[c] ++++++x[C-1]+pl/pt[C-1]+.*
*.+          ++          ++          +.*
*.+      Child[0]      ++          ++      Child[C-1]      +.*
*.+          ++          ++          +.*
*.+++++          Child[c]          +.*
*.          +          ++++++.*
*.          +          +.*
*.          ++++++.*
* .....*
*****

```

Note: . . . . can be narrower than ~~~~

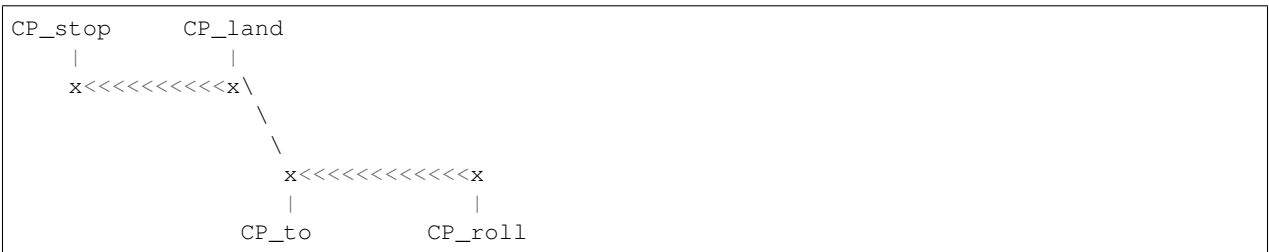
## Vertices

The following show root at the left. Linking parent to child:



PC\_roll and PC\_to are determined by the parent. PC\_land and PC\_stop are determined by the child.

And child to parent:



CP\_roll and CP\_to are determined by the child. CP\_land and CP\_stop are determined by the parent.

**exception** `cpip.plot.PlotNode.ExceptionPlotNode`  
Exception when handling PlotNodeBbox object.

**class** `cpip.plot.PlotNode.PlotNodeBbox`

This is a class that can hold the width and depth of an object and the bounding box of self and the children. This can then compute various dimensions of self and children.

**bbChildren**

The bounding box of children as a Coord.Box() or None. i.e. the box ....

**bbChildrenDepth**

The bounding box depth of children as a Coord.Dim() or None. i.e. the depth of box ....

**bbChildrenWidth**

The bounding box width of children as a Coord.Dim() or None. i.e. the width of box ....

**bbSelfDepth**

The depth of self plus padding as a Coord.Dim(). i.e. the depth of box ~~~~

**bbSelfPadding**

The immediate padding around self as a Coord.Pad().

**bbSelfWidth**

The width of self plus padding as a Coord.Dim() or None. i.e. the width of box ~~~~

**bbSigma**

Bounding box of self and my children as a Coord.Box().

**bbSigmaDepth**

The depth of self+children as a Coord.Dim() or None in the case that I don't exist and I have no children.  
i.e. the depth of box \*\*\*\*

**bbSigmaWidth**

The depth of self+children as a Coord.Dim() or None in the case that I don't exist and I have no children.  
i.e. the width of box \*\*\*\*

**bbSpaceChildren**

The additional distance to give to the children as a Coord.Dim().

**box**

The Coord.Box() of #####.

**childBboxDatum** (*theDatum*)

The point x[0] as a Coord.Pt() given theDatum as Coord.Pt() or None if no children.

**depth**

The immediate depth of the node, if None then no BB depth or bbSpaceChildrend is allocated. i.e. the depth of box #####

**extendChildBbox** (*theChildBbox*)

Extends the child bounding box by the amount theChildBbox which should be a Coord.Box(). This extends the .... line.

**hasSetArea**

Returns True if width and depth are set, False otherwise.

**plotPointCentre** (*theLd*)

Returns the logical point at the centre of the box shown as ##### above.

**plotPointSelf** (*theDatum*)

The point S as a Coord.Pt() given theDatum as Coord.Pt().

**width**

The immediate width of the node, if None then no BB width is allocated. i.e. the width of box #####

**class** cpip.plot.PlotNode.**PlotNodeBboxBoxy**

Sub-class parent child edges that contact the corners of the box shown as ##### above.

**cpLand** (*theLd, childIndex*)

The me-as-parent-from-child landing point given the logical datum as a Coord.Pt.

**cpRoll** (*theLd*)

The me-as-child-to-parent start point given the logical datum as a Coord.Pt.

**cpStop** (*theLd, childIndex*)

The me-as-parent-from-child stop point given the logical datum as a Coord.Pt.

**cpTo** (*theLd*)

The me-as-child-to-parent take off point given the logical datum as a Coord.Pt.

**pcLand** (*theLd*)

The parent-to-me-as-child landing point given the logical datum as a Coord.Pt.

**pcRoll** (*theDatum, childIndex*)

The me-as-parent-to-child logical start point given the logical datum as a Coord.Pt and the child ordinal.  
This gives equispaced points along the lower edge.

**pcStop** (*theLd*)

The parent-to-me-as-child stop point given the logical datum as a Coord.Pt.

**pcTo** (*theDatum, childIndex*)

The me-as-parent-to-child logical take off point given the logical datum as a Coord.Pt and the child ordinal.  
This gives equispaced points along the lower edge.

**class** `cpip.plot.PlotNode.PlotNodeBboxRoundy`

Sub-class for parent child edges that contact the centre of the box shown as ##### above.

**cpLand** (*theDatumL, childIndex*)

The me-as-parent-from-child landing point given the logical datum as a Coord.Pt.

**cpRoll** (*theDatumL*)

The me-as-child-to-parent start point given the logical datum as a Coord.Pt.

**cpStop** (*theDatumL, childIndex*)

The me-as-parent-from-child stop point given the logical datum as a Coord.Pt.

**cpTo** (*theDatumL*)

The me-as-child-to-parent take off point given the logical datum as a Coord.Pt.

**pcLand** (*theDatumL*)

The parent-to-me-as-child landing point given the logical datum as a Coord.Pt.

**pcRoll** (*theDatumL, childIndex*)

The me-as-parent-to-child logical start point given the logical datum as a Coord.Pt and the child ordinal.  
This gives equispaced points along the lower edge.

**pcStop** (*theDatumL*)

The parent-to-me-as-child stop point given the logical datum as a Coord.Pt.

**pcTo** (*theDatumL, childIndex*)

The me-as-parent-to-child logical take off point given the logical datum as a Coord.Pt and the child ordinal.  
This gives equispaced points along the lower edge.

## SVGWriter

An SVG writer.

**exception** `cpip.plot.SVGWriter.ExceptionSVGWriter`

Exception class for SVGWriter.

**class** `cpip.plot.SVGWriter.SVGCircle` (*theXmlStream, thePoint, theRadius, attrs=None*)

A circle in SVG. See: <http://www.w3.org/TR/2003/REC-SVG11-20030114/shapes.html#CircleElement>



Initialise the circle with a stream, a `Coord.Pt()` and a `Coord.Dim()` objects.

**class** `cpip.plot.SVGWriter.SVGEllipse` (*theXmlStream, ptFrom, theRadX, theRadY, attrs=None*)  
An ellipse in SVG. See: <http://www.w3.org/TR/2003/REC-SVG11-20030114/shapes.html#EllipseElement>

Initialise the ellipse with a stream, a `Coord.Pt()` and a `Coord.Dim()` objects.

**class** `cpip.plot.SVGWriter.SVGGroup` (*theXmlStream, attrs=None*)  
Initialise the group with a stream.

See: <http://www.w3.org/TR/2003/REC-SVG11-20030114/struct.html#GElement>

Sadly we can't use `**kwargs` because of Python restrictions on keyword names. For example `stroke-width` that is not a valid keyword argument (although `stroke_width` would be). So instead we pass in an optional dictionary `{string : string, ...}`

**class** `cpip.plot.SVGWriter.SVGLine` (*theXmlStream, ptFrom, ptTo, attrs=None*)  
A line in SVG. See: <http://www.w3.org/TR/2003/REC-SVG11-20030114/shapes.html#LineElement>

Initialise the line with a stream, and two `Coord.Pt()` objects.

**class** `cpip.plot.SVGWriter.SVGPointList` (*theXmlStream, name, pointS, attrs*)  
An abstract class that takes a list of points, derived by polyline and polygon.

Initialise the element with a stream, a name, and a list of `Coord.Pt()` objects.

NOTE: The units of the points are ignored, it is up to the caller to convert them to the User Coordinate System.

**class** `cpip.plot.SVGWriter.SVGPolygon` (*theXmlStream, pointS, attrs=None*)  
A polygon in SVG. See: <http://www.w3.org/TR/2003/REC-SVG11-20030114/shapes.html#PolygonElement>

Initialise the polygon with a stream, and a list of `Coord.Pt()` objects.

NOTE: The units of the points are ignored, it is up to the caller to convert them to the User Coordinate System.

**class** `cpip.plot.SVGWriter.SVGPolyline` (*theXmlStream, pointS, attrs=None*)  
A polyline in SVG. See: <http://www.w3.org/TR/2003/REC-SVG11-20030114/shapes.html#PolylineElement>

Initialise the polyline with a stream, and a list of `Coord.Pt()` objects.

NOTE: The units of the points are ignored, it is up to the caller to convert them to the User Coordinate System.

**class** `cpip.plot.SVGWriter.SVGRect` (*theXmlStream, thePoint, theBox, attrs=None*)  
Initialise the rectangle with a stream, a `Coord.Pt()` and a `Coord.Box()` objects. See: <http://www.w3.org/TR/2003/REC-SVG11-20030114/shapes.html#RectElement> Typical attributes: `{'fill' : "blue", 'stroke' : "black", 'stroke-width' : "2", }`

**class** `cpip.plot.SVGWriter.SVGText` (*theXmlStream, thePoint, theFont, theSize, attrs=None*)  
Text in SVG. See: <http://www.w3.org/TR/2003/REC-SVG11-20030114/text.html#TextElement>

Initialise the text with a stream, a `Coord.Pt()` and font as a string and size as an integer. If thePoint is None then no location will be specified (for example for use inside a `<defs>` element).

**class** `cpip.plot.SVGWriter.SVGWriter` (*theFile, theViewPort, rootAttrs=None, mustIndent=True*)  
Initialise the stream with a file and `Coord.Box()` object. The view port units must be the same for width and depth.

`cpip.plot.SVGWriter.dimToTxt` (*theDim*)  
Converts a `Coord.Dim()` object to text for SVG units.

## TreePlotTransform

Provides a means of re-interpreting the coordinate system when plotting trees so that the the tree root can be top/left/bottom/right and the child order plotted anti-clockwise or clockwise.

This can convert ‘logical’ positions into ‘physical’ positions. Where a ‘logical’ position is one with the root of the tree at the top and the child nodes below in left-to-right (i.e. anti-clockwise) order. A ‘physical’ position is a plot-able position where the root of the tree is top/left/bottom or right and the child nodes are in anti-clockwise or clockwise order.

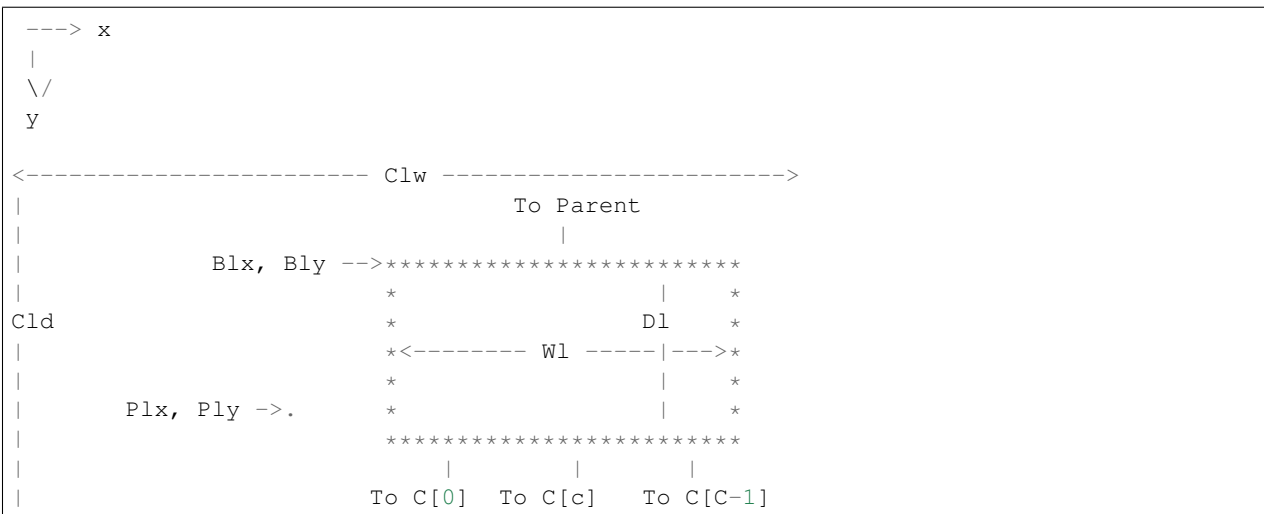
### Transforming sizes and positions

If the first suffix is ‘l’ this is the “logical” coordinate system. If the first suffix is ‘p’ this is the “physical” coordinate system.

Then:

- C - The canvas dimension, Cpw is “Canvas physical width”
- W - Width dimension, physical and logical.
- D - Depth dimension, physical and logical.
- B - Box datum position (“top-left”), physical and logical, x and y.
- P - Arbitrary point, physical and logical, x and y.

So this “logical view” of the tree graph (‘top’ and ‘-’): i.e. Root(s) is a top and children are written in an anti-clockwise.



Or:

Origin	Cpw	Cpd	Wp	Dp	Bpx	Bpy	Ppx	Ppy
top	Clw	Cld	Wl	Dl	Blx	Bly	Plx	Ply
left	Cld	Clw	Dl	Wl	Bly	(Clw-Plx-Wl)	Ply	Clw-Plx
bottom	Clw	Cld	Wl	Dl	(Clw-Plx-Wl)	(Cld-Ply-Dl)	Clw-Plx	Cld-Ply
right	Cld	Clw	Dl	Wl	(Cld-Ply-Dl)	Blx	Cld-Ply	Plx

Note the diagonal top-right to bottom-left transference between each pair of columns. That is because with each successive line we are doing a 90 degree rotation (anti-clockwise) plus a +ve y translation by Clw (top->left or bottom->right) or Cld (left->bottom or right->top).

### Incrementing child positions

Moving from one child to another is done in the following combinations:

Origin	'-'	'+'
top	right	left
left	up	down
bottom	left	right
right	down	up

**exception** `cpip.plot.TreePlotTransform.ExceptionTreePlotTransform`

Exception class for TreePlotTransform.

**exception** `cpip.plot.TreePlotTransform.ExceptionTreePlotTransformRangeCtor`

Exception class for out of range input on construction.

**class** `cpip.plot.TreePlotTransform.TreePlotTransform` (*theLogicalCanvas*, *rootPos='top'*,  
*sweepDir='-'*)

Provides a means of re-interpreting the coordinate system when plotting trees.

`rootPosition = frozenset(['top', 'bottom', 'left', 'right'])` default: 'top'

`sweepDirection = frozenset(['+', '-'])` default: '-'

Has functionality for interpreting width/depth to actual postions given rootPostion.

**bdcL** (*theBlxy*, *theBl*)

Given a logical datum (logical top left) and a logical box this returns logical bottom dead centre of a box.

**bdcP** (*theBlxy*, *theBl*)

Given a logical datum (logical top left) and a logical box this returns physical bottom dead centre of a box.

**boxDatumP** (*theBlxy*, *theBl*)

Given a logical point and logical box this returns a physical point that is the box datum ("upper left").

**boxP** (*theBl*)

Given a logical box this returns a Coord.Box that describes the physical box.

**canvasP** ()

Returns a Coord.Box that describes the physical canvass.

**genRootPos** ()

Yield all possible root positions.

**genSweepDir** ()

Yield all possible root positions.

**incPhysicalChildPos** (*thePt*, *theDim*)

Given a child physical datum point and a distance to next child this returns the next childs physical datum point. TODO: Remove this as redundant?

**nextdcL** (*theBlxy*, *theBl*)

Given a logical datum (logical top left) and a logical box this returns logical 'next' dead centre of a box.

**positiveSweepDir**

True if positive sweep, false otherwise.

**postIncChildLogicalPos** (*thePt*, *theBox*)

Post-incrempents the child logical datum point ('top-left') given the child logical datum point and the child.bbSigma. Returns a Coord.Pt(). This takes into account the sweep direction.

**preIncChildLogicalPos** (*thePt*, *theBox*)

Pre-incrempents the child logical datum point ('top-left') given the child logical datum point and the child.bbSigma. Returns a Coord.Pt(). This takes into account the sweep direction.

**prevdcL** (*theBlxy*, *theBl*)

Given a logical datum (logical top left) and a logical box this returns logical 'previous' dead centre of a box.

**pt** (*thePt*, *units=None*)

Given an arbitrary logical point as a `Coord.Pt()`, this returns the physical point as a `Coord.Pt()`. If *units* is supplied then the return value will be in those units.

**startChildrenLogicalPos** (*thePt*, *theBox*)

Returns the starting child logical datum point ('top-left') given the children logical datum point and the `children.bbSigma`. Returns a `Coord.Pt()`. This takes into account the sweep direction.

**tdcL** (*theBlxy*, *theBl*)

Given a logical datum (logical top left) and a logical box this returns logical top dead centre of a box.

**tdcP** (*theBlxy*, *theBl*)

Given a logical datum (logical top left) and a logical box this returns physical top dead centre of a box.

---

### Usage

---

To use cpip in a project:

```
import cpip
```

Have a read of the [CPIP Tutorials](#) for how you can use CPIP programatically. In there is a tutorial on how to use the `PpLexer` that is the equivalent of `cpp`: [PpLexer Tutorial](#). There is also the [FileIncludeGraph Tutorial](#) showing how you can analyse how `#include`'d files are processed. This is very useful for understanding file dependencies.



Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways:

### Types of Contributions

#### Report Bugs

Report bugs at <https://github.com/paulross/cpip/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

#### Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

#### Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

## Write Documentation

cpip could always use more documentation, whether as part of the official cpip docs, in docstrings, or even on the web in blog posts, articles, and such.

## Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/paulross/cpip/issues>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## Get Started!

Ready to contribute? Here's how to set up *cpip* for local development.

1. Fork the *cpip* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/cpip.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv cpip
$ cd cpip/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 cpip tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.



## Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check [https://travis-ci.org/paulross/cpip/pull\\_requests](https://travis-ci.org/paulross/cpip/pull_requests) and make sure that the tests pass for all supported Python versions.

## Tips

To run a subset of tests:

```
$ py.test tests.test_cpip
```



# CHAPTER 11

---

## Credits

---

### Development Lead

- Paul Ross <apaulross@gmail.com>

### Contributors

None yet. Why not be the first?



#### **0.9.7 Beta Release (2017-10-04)**

- Minor fixes.
- Performance optimisations.
- Builds the CPython source tree in 5 hours with 2 CPUs.
- DOcumentation improvements.

#### **0.9.5 Beta Release (2017-10-03)**

- Migrate from sourceforge to GitHub.

#### **0.9.1 (2014-09-03)**

Version 0.9.1, various minor fixes. Tested on Python 2.7 and 3.3.

#### **Alpha Plus Release (2014-09-04)**

Fairly thorough refactor. CPIP now tested on Python 2.7, 3.3. Version 0.9.1. Updated documentation.

#### **Alpha Release (2012-03-25)**

Very little functional change. CPIP now tested on Python 2.6, 2.7, 3.2. Added loads of documentation.

## Alpha Release (2011-07-14)

This is a pre-release of CPIP. It is tested on BSD/Linux, it will probably work on Windows (although some unit tests will fail on that platform).

Project started in 2008.

## CHAPTER 13

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`





### C

`cpip.core.ConstantExpression`, 72  
`cpip.core.CppCond`, 73  
`cpip.core.CppDiagnostic`, 79  
`cpip.core.FileIncludeGraph`, 81  
`cpip.core.FileIncludeStack`, 84  
`cpip.core.IncludeHandler`, 86  
`cpip.core.ItuToTokens`, 88  
`cpip.core.MacroEnv`, 88  
`cpip.core.PpDefine`, 91  
`cpip.core.PpLexer`, 97  
`cpip.core.PpToken`, 100  
`cpip.core.PpTokenCount`, 102  
`cpip.core.PpTokeniser`, 104  
`cpip.core.PpWhitespace`, 106  
`cpip.core.PragmaHandler`, 107  
`cpip.CPIPMain`, 65  
`cpip.CppCondGraphToHtml`, 67  
`cpip.FileStatus`, 67  
`cpip.IncGraphSVG`, 69  
`cpip.IncGraphSVGBase`, 70  
`cpip.IncGraphXML`, 70  
`cpip.IncList`, 70  
`cpip.ItuToHtml`, 70  
`cpip.MacroHistoryHtml`, 70  
`cpip.plot.Coord`, 118  
`cpip.plot.PlotNode`, 120  
`cpip.plot.SVGWriter`, 124  
`cpip.plot.TreePlotTransform`, 125  
`cpip.TokenCss`, 71  
`cpip.Tu2Html`, 71  
`cpip.TuIndexer`, 72  
`cpip.util.BufGen`, 108  
`cpip.util.CommonPrefix`, 109  
`cpip.util.DictTree`, 109  
`cpip.util.DirWalk`, 111  
`cpip.util.HtmlUtils`, 112  
`cpip.util.ListGen`, 113  
`cpip.util.MatrixRep`, 114  
`cpip.util.MaxMunchGen`, 114  
`cpip.util.OaS`, 115  
`cpip.util.StrTree`, 115  
`cpip.util.Tree`, 115  
`cpip.util.XmlWrite`, 116



## Symbols

\_\_add\_\_() (cpip.plot.Coord.Dim method), 118  
 \_\_eq\_\_() (cpip.plot.Coord.Dim method), 118  
 \_\_eq\_\_() (cpip.plot.Coord.Pt method), 119  
 \_\_ge\_\_() (cpip.plot.Coord.Dim method), 119  
 \_\_gt\_\_() (cpip.plot.Coord.Dim method), 119  
 \_\_iadd\_\_() (cpip.core.PpTokenCount.PpTokenCount  
 method), 103  
 \_\_iadd\_\_() (cpip.plot.Coord.Dim method), 118  
 \_\_init\_\_() (cpip.core.PpTokenCount.PpTokenCountStack  
 method), 103  
 \_\_isub\_\_() (cpip.plot.Coord.Dim method), 118  
 \_\_le\_\_() (cpip.plot.Coord.Dim method), 118  
 \_\_lt\_\_() (cpip.plot.Coord.Dim method), 118  
 \_\_ne\_\_() (cpip.plot.Coord.Dim method), 119  
 \_\_str\_\_() (cpip.plot.Coord.Pad method), 119  
 \_\_str\_\_() (cpip.plot.Coord.Pt method), 119  
 \_\_sub\_\_() (cpip.plot.Coord.Dim method), 118  
 \_\_weakref\_\_ (cpip.core.PpTokenCount.PpTokenCount  
 attribute), 103  
 \_\_weakref\_\_ (cpip.core.PpTokenCount.PpTokenCountStack  
 attribute), 103

## A

acceptVisitor() (cpip.core.FileIncludeGraph.FileIncludeGraph  
 method), 82  
 acceptVisitor() (cpip.core.FileIncludeGraph.FileIncludeGraphRoot  
 method), 84  
 add() (cpip.TuIndexer.TuIndexer method), 72  
 add() (cpip.util.DictTree.DictTree method), 109  
 add() (cpip.util.StrTree.StrTree method), 115  
 addBranch() (cpip.core.FileIncludeGraph.FileIncludeGraph  
 method), 82  
 addChild() (cpip.core.FileIncludeGraph.FigVisitorTreeNodeBase  
 method), 82  
 addGraph() (cpip.core.FileIncludeGraph.FileIncludeGraphRoot  
 method), 84  
 addLineColRep() (cpip.util.MatrixRep.MatrixRep  
 method), 114

allChildren (cpip.util.Tree.DuplexAdjacencyList at-  
 tribute), 116  
 allParents (cpip.util.Tree.DuplexAdjacencyList attribute),  
 116  
 allStaticMacroDependencies()  
 (cpip.core.MacroEnv.MacroEnv method),  
 89  
 anyToken() (in module cpip.util.MaxMunchGen), 115  
 assertReplListIntegrity() (cpip.core.PpDefine.PpDefine  
 method), 93

## B

baseUnitsDim() (in module cpip.plot.Coord), 119  
 bbChildren (cpip.plot.PlotNode.PlotNodeBbox attribute),  
 122  
 bbChildrenDepth (cpip.plot.PlotNode.PlotNodeBbox at-  
 tribute), 123  
 bbChildrenWidth (cpip.plot.PlotNode.PlotNodeBbox at-  
 tribute), 123  
 bbSelfDepth (cpip.plot.PlotNode.PlotNodeBbox at-  
 tribute), 123  
 bbSelfPadding (cpip.plot.PlotNode.PlotNodeBbox  
 attribute), 123  
 bbSelfWidth (cpip.plot.PlotNode.PlotNodeBbox at-  
 tribute), 123  
 bbSigma (cpip.plot.PlotNode.PlotNodeBbox attribute),  
 123  
 bbSigmaDepth (cpip.plot.PlotNode.PlotNodeBbox  
 attribute), 123  
 bbSigmaWidth (cpip.plot.PlotNode.PlotNodeBbox  
 attribute), 123  
 bbSpaceChildren (cpip.plot.PlotNode.PlotNodeBbox at-  
 tribute), 123  
 bdcL() (cpip.plot.TreePlotTransform.TreePlotTransform  
 method), 127  
 bdcP() (cpip.plot.TreePlotTransform.TreePlotTransform  
 method), 127  
 box (cpip.plot.PlotNode.PlotNodeBbox attribute), 123  
 boxDatumP() (cpip.plot.TreePlotTransform.TreePlotTransform  
 method), 127

boxP() (cpip.plot.TreePlotTransform.TreePlotTransform method), 127

branches() (cpip.util.Tree.Tree method), 116

BufGen (class in cpip.util.BufGen), 108

## C

C\_KEYWORDS (in module cpip.core.PpTokeniser), 104

CALL\_STACK\_DEPTH\_ASSUMED\_PPTOKENS (cpip.core.PpLexer.PpLexer attribute), 98

CALL\_STACK\_DEPTH\_FIRST\_INCLUDE (cpip.core.PpLexer.PpLexer attribute), 98

CALL\_STACK\_DEPTH\_PER\_INCLUDE (cpip.core.PpLexer.PpLexer attribute), 98

canAccept() (cpip.core.CppCond.CppCondGraphNode method), 77

canInclude() (cpip.core.IncludeHandler.CppIncludeStd method), 86

canReplace (cpip.core.PpToken.PpToken attribute), 101

canvasP() (cpip.plot.TreePlotTransform.TreePlotTransform method), 127

CcgVisitorToHtml (class in cpip.CppCondGraphToHtml), 67

characters() (cpip.util.XmlWrite.XmlStream method), 117

childBboxDatum() (cpip.plot.PlotNode.PlotNodeBbox method), 123

children() (cpip.util.Tree.DuplexAdjacencyList method), 116

clear() (cpip.core.MacroEnv.MacroEnv method), 89

clearFindLogic() (cpip.core.IncludeHandler.CppIncludeStd method), 86

clearHistory() (cpip.core.IncludeHandler.CppIncludeStd method), 86

close() (cpip.core.CppCond.CppCond method), 75

close() (cpip.core.PpTokenCount.PpTokenCountStack method), 103

cmdLine (cpip.CPIPMain.MainJobSpec attribute), 65

colNum (cpip.core.PpLexer.PpLexer attribute), 98

colNum (cpip.core.PpToken.PpToken attribute), 101

comment() (cpip.util.XmlWrite.XmlStream method), 117

COMMENT\_REPLACEMENT (in module cpip.core.PpTokeniser), 104

COND\_LEVEL\_DEFAULT (cpip.core.PpLexer.PpLexer attribute), 98

COND\_LEVEL\_OPTIONS (cpip.core.PpLexer.PpLexer attribute), 98

condComp (cpip.core.FileIncludeGraph.FileIncludeGraph attribute), 83

condComp (cpip.IncGraphSVG.SVGTreeNodeMain attribute), 69

condCompGraph (cpip.core.PpLexer.PpLexer attribute), 98

condCompState (cpip.core.FileIncludeGraph.FileIncludeGraph attribute), 83

conditionalLevel (cpip.CPIPMain.MainJobSpec attribute), 65

ConditionalState (class in cpip.core.CppCond), 74

condState (cpip.core.PpLexer.PpLexer attribute), 99

ConstantExpression (class in cpip.core.ConstantExpression), 72

constExprStr() (cpip.core.CppCond.ConditionalState method), 74

consumeFunctionPreamble() (cpip.core.PpDefine.PpDefine method), 93

convert() (cpip.plot.Coord.Dim method), 118

convert() (cpip.plot.Coord.Pt method), 119

convert() (in module cpip.plot.Coord), 118

convertPt() (in module cpip.plot.Coord), 119

copy() (cpip.core.PpToken.PpToken method), 101

count (cpip.FileStatus.FileInfo attribute), 68

counter() (cpip.core.PpTokenCount.PpTokenCountStack method), 103

cpip.core.ConstantExpression (module), 72

cpip.core.CppCond (module), 73

cpip.core.CppDiagnostic (module), 79

cpip.core.FileIncludeGraph (module), 81

cpip.core.FileIncludeStack (module), 84

cpip.core.IncludeHandler (module), 86

cpip.core.ItuToTokens (module), 88

cpip.core.MacroEnv (module), 88

cpip.core.PpDefine (module), 91

cpip.core.PpLexer (module), 97

cpip.core.PpToken (module), 100

cpip.core.PpTokenCount (module), 102

cpip.core.PpTokeniser (module), 104

cpip.core.PpWhitespace (module), 106

cpip.core.PragmaHandler (module), 107

cpip.CPIPMain (module), 65

cpip.CppCondGraphToHtml (module), 67

cpip.FileStatus (module), 67

cpip.IncGraphSVG (module), 69

cpip.IncGraphSVGBase (module), 70

cpip.IncGraphXML (module), 70

cpip.IncList (module), 70

cpip.ItuToHtml (module), 70

cpip.MacroHistoryHtml (module), 70

cpip.plot.Coord (module), 118

cpip.plot.PlotNode (module), 120

cpip.plot.SVGWriter (module), 124

cpip.plot.TreePlotTransform (module), 125

cpip.TokenCss (module), 71

cpip.Tu2Html (module), 71

cpip.TuIndexer (module), 72

cpip.util.BufGen (module), 108

cpip.util.CommonPrefix (module), 109

cpip.util.DictTree (module), 109

cpip.util.DirWalk (module), 111

cpip.util.HtmlUtils (module), 112

- `cpip.util.ListGen` (module), 113
  - `cpip.util.MatrixRep` (module), 114
  - `cpip.util.MaxMunchGen` (module), 114
  - `cpip.util.OaS` (module), 115
  - `cpip.util.StrTree` (module), 115
  - `cpip.util.Tree` (module), 115
  - `cpip.util.XmlWrite` (module), 116
  - `cpLand()` (`cpip.plot.PlotNode.PlotNodeBboxBoxy` method), 123
  - `cpLand()` (`cpip.plot.PlotNode.PlotNodeBboxRoundy` method), 124
  - `CPP_CONCAT_OP` (`cpip.core.PpDefine.PpDefine` attribute), 92
  - `CPP_STRINGIZE_OP` (`cpip.core.PpDefine.PpDefine` attribute), 92
  - `CppCond` (class in `cpip.core.CppCond`), 74
  - `CppCondGraph` (class in `cpip.core.CppCond`), 75
  - `CppCondGraphIfSection` (class in `cpip.core.CppCond`), 77
  - `CppCondGraphNode` (class in `cpip.core.CppCond`), 77
  - `CppCondGraphVisitorBase` (class in `cpip.core.CppCond`), 78
  - `CppCondGraphVisitorConditionalLines` (class in `cpip.core.CppCond`), 78
  - `CppIncludeStd` (class in `cpip.core.IncludeHandler`), 86
  - `CppIncludeStdin` (class in `cpip.core.IncludeHandler`), 87
  - `CppIncludeStdOs` (class in `cpip.core.IncludeHandler`), 87
  - `CppIncludeStringIO` (class in `cpip.core.IncludeHandler`), 88
  - `cppTokType` (`cpip.core.PpTokeniser.PpTokeniser` attribute), 105
  - `cpRoll()` (`cpip.plot.PlotNode.PlotNodeBboxBoxy` method), 123
  - `cpRoll()` (`cpip.plot.PlotNode.PlotNodeBboxRoundy` method), 124
  - `cpStack` (`cpip.core.IncludeHandler.CppIncludeStd` attribute), 86
  - `cpStackPop()` (`cpip.core.IncludeHandler.CppIncludeStd` method), 86
  - `cpStackPush()` (`cpip.core.IncludeHandler.CppIncludeStd` method), 86
  - `cpStackSize` (`cpip.core.IncludeHandler.CppIncludeStd` attribute), 86
  - `cpStop()` (`cpip.plot.PlotNode.PlotNodeBboxBoxy` method), 124
  - `cpStop()` (`cpip.plot.PlotNode.PlotNodeBboxRoundy` method), 124
  - `cpTo()` (`cpip.plot.PlotNode.PlotNodeBboxBoxy` method), 124
  - `cpTo()` (`cpip.plot.PlotNode.PlotNodeBboxRoundy` method), 124
  - `currentFile` (`cpip.core.FileIncludeStack.FileIncludeStack` attribute), 85
  - `currentFile` (`cpip.core.PpLexer.PpLexer` attribute), 99
  - `currentPlace` (`cpip.core.IncludeHandler.CppIncludeStd` attribute), 86
  - `currentPlace` (`cpip.core.IncludeHandler.FilePathOrigin` attribute), 88
- ## D
- `debug()` (`cpip.core.CppDiagnostic.PreprocessDiagnosticStd` method), 79
  - `decodeString()` (in module `cpip.util.XmlWrite`), 117
  - `define()` (`cpip.core.MacroEnv.MacroEnv` method), 89
  - `DEFINE_WHITESPACE` (in module `cpip.core.PpWhitespace`), 106
  - `defined()` (`cpip.core.MacroEnv.MacroEnv` method), 89
  - `definedMacros` (`cpip.core.PpLexer.PpLexer` attribute), 99
  - `depth` (`cpip.core.FileIncludeGraph.FigVisitorTree` attribute), 81
  - `depth` (`cpip.core.FileIncludeStack.FileIncludeStack` attribute), 85
  - `depth` (`cpip.plot.PlotNode.PlotNodeBbox` attribute), 123
  - `depth()` (`cpip.util.DictTree.DictTree` method), 109
  - `diagnostic` (`cpip.CPIPMain.MainJobSpec` attribute), 65
  - `DictTree` (class in `cpip.util.DictTree`), 109
  - `DictTreeHtmlTable` (class in `cpip.util.DictTree`), 109
  - `DIGRAPH_TABLE` (in module `cpip.core.PpTokeniser`), 104
  - `Dim` (class in `cpip.plot.Coord`), 118
  - `dimToTxt()` (in module `cpip.plot.SVGWriter`), 125
  - `DIRECTIVES` (`cpip.core.PragmaHandler.PragmaHandlerSTDC` attribute), 108
  - `dirWalk()` (in module `cpip.util.DirWalk`), 111
  - `DUMMY_FILE_LINENUM` (in module `cpip.core.FileIncludeGraph`), 81
  - `DUMMY_FILE_NAME` (in module `cpip.core.FileIncludeGraph`), 81
  - `dumpGraph()` (`cpip.core.FileIncludeGraph.FileIncludeGraph` method), 83
  - `dumpGraph()` (`cpip.core.FileIncludeGraph.FileIncludeGraphRoot` method), 84
  - `dumpList` (`cpip.CPIPMain.MainJobSpec` attribute), 65
  - `DuplexAdjacencyList` (class in `cpip.util.Tree`), 116
- ## E
- `Element` (class in `cpip.util.XmlWrite`), 116
  - `encodeString()` (in module `cpip.util.XmlWrite`), 117
  - `endElement()` (`cpip.util.XmlWrite.XmlStream` method), 117
  - `endInclude()` (`cpip.core.IncludeHandler.CppIncludeStd` method), 86
  - `ENUM_NAME` (in module `cpip.core.PpToken`), 100
  - `error()` (`cpip.core.CppDiagnostic.PreprocessDiagnosticRaiseOnError` method), 79
  - `error()` (`cpip.core.CppDiagnostic.PreprocessDiagnosticStd` method), 80

`evalConstExpr()` (`cpip.core.PpToken.PpToken` method), 101

`evaluate()` (`cpip.core.ConstantExpression.ConstantExpression` method), 72

`eventList` (`cpip.core.CppDiagnostic.PreprocessDiagnosticStream` attribute), 80

`ExceptionBufGen`, 109

`ExceptionConditionalExpression`, 72, 97

`ExceptionConditionalExpressionInit`, 73

`ExceptionConstantExpression`, 73

`ExceptionCoord`, 118

`ExceptionCoordUnitConvert`, 118

`ExceptionCpipDefine`, 91

`ExceptionCpipDefineBadArguments`, 91

`ExceptionCpipDefineBadWs`, 91

`ExceptionCpipDefineDupeId`, 91

`ExceptionCpipDefineInit`, 91

`ExceptionCpipDefineInitBadLine`, 92

`ExceptionCpipDefineInvalidCmp`, 92

`ExceptionCpipDefineMissingWs`, 92

`ExceptionCpipDefineReplace`, 92

`ExceptionCpipToken`, 100

`ExceptionCpipTokenIllegalMerge`, 100

`ExceptionCpipTokenIllegalOperation`, 100

`ExceptionCpipTokeniser`, 104

`ExceptionCpipTokeniserUcnConstraint`, 104

`ExceptionCpipTokenReopenForExpansion`, 100

`ExceptionCpipTokenUnknownType`, 100

`ExceptionCppCond`, 78

`ExceptionCppCondGraph`, 78

`ExceptionCppCondGraphElif`, 78

`ExceptionCppCondGraphElse`, 78

`ExceptionCppCondGraphIfSection`, 78

`ExceptionCppCondGraphNode`, 78

`ExceptionCppDiagnostic`, 79

`ExceptionCppDiagnosticPartialTokenStream`, 79

`ExceptionCppDiagnosticUndefined`, 79

`ExceptionCppInclude`, 88

`ExceptionDictTree`, 111

`ExceptionDirWalk`, 111

`ExceptionEvaluateExpression`, 73

`ExceptionFileIncludeGraph`, 81

`ExceptionFileIncludeGraphRoot`, 81

`ExceptionFileIncludeGraphTokenCounter`, 81

`ExceptionFileIncludeStack`, 84

`ExceptionMacroEnv`, 88

`ExceptionMacroEnvInvalidRedefinition`, 88

`ExceptionMacroEnvNoMacroDefined`, 89

`ExceptionMacroIndexError`, 89

`ExceptionMacroReplacementInit`, 89

`ExceptionMacroReplacementPredefinedRedefinition`, 89

`ExceptionMatrixRep`, 114

`ExceptionMaxMunchGen`, 114

`ExceptionOas`, 115

`ExceptionPlotNode`, 122

`ExceptionPpLexer`, 97

`ExceptionPpLexerAlreadyGenerating`, 97

`ExceptionPpLexerCallStack`, 97

`ExceptionPpLexerCallStackTooSmall`, 97

`ExceptionPpLexerCondLevelOutOfRange`, 97

`ExceptionPpLexerDefine`, 97

`ExceptionPpLexerNestedIncludeLimit`, 97

`ExceptionPpLexerNoFile`, 97

`ExceptionPpLexerPredefine`, 97

`ExceptionPpLexerPreInclude`, 97

`ExceptionPpLexerPreIncludeIncNoCp`, 97

`ExceptionPpTokenCount`, 102

`ExceptionPpTokenCountStack`, 102

`ExceptionPragmaHandler`, 107

`ExceptionPragmaHandlerStopParsing`, 107

`ExceptionSVGWriter`, 124

`ExceptionTreePlotTransform`, 127

`ExceptionTreePlotTransformRangeCtor`, 127

`ExceptionTuIndexer`, 72

`ExceptionXml`, 116

`ExceptionXmlEndElement`, 116

`expandArguments` (`cpip.core.PpDefine.PpDefine` attribute), 93

`extendChildBbox()` (`cpip.plot.PlotNode.PlotNodeBbox` method), 123

## F

`FigVisitorBase` (class in `cpip.core.FileIncludeGraph`), 81

`FigVisitorDot` (class in `cpip.CPIPMain`), 65

`FigVisitorFileSet` (class in `cpip.core.FileIncludeGraph`), 81

`FigVisitorLargestCommonPrefix` (class in `cpip.CPIPMain`), 65

`FigVisitorTree` (class in `cpip.core.FileIncludeGraph`), 81

`FigVisitorTreeNodeBase` (class in `cpip.core.FileIncludeGraph`), 81

`fileId` (`cpip.core.PpDefine.PpDefine` attribute), 94

`fileIdS` (`cpip.core.CppCond.CppCondGraphVisitorConditionalLines` attribute), 78

`FileInclude` (class in `cpip.core.FileIncludeStack`), 84

`FileIncludeGraph` (class in `cpip.core.FileIncludeGraph`), 82

`FileIncludeGraphRoot` (class in `cpip.core.FileIncludeGraph`), 84

`fileIncludeGraphRoot` (`cpip.core.FileIncludeStack.FileIncludeStack` attribute), 85

`fileIncludeGraphRoot` (`cpip.core.PpLexer.PpLexer` attribute), 99

`FileIncludeStack` (class in `cpip.core.FileIncludeStack`), 85

`FileInfo` (class in `cpip.FileStatus`), 68

`FileInfoSet` (class in `cpip.FileStatus`), 68

`FileInOut` (class in `cpip.util.DirWalk`), 111

- fileLineCol (cpip.core.FileIncludeStack.FileIncludeStack attribute), 85
- fileLineCol (cpip.core.PpLexer.PpLexer attribute), 99
- fileLineCol (cpip.core.PpTokeniser.PpTokeniser attribute), 105
- fileLocator (cpip.core.PpTokeniser.PpTokeniser attribute), 105
- fileName (cpip.core.FileIncludeGraph.FileIncludeGraph attribute), 83
- fileName (cpip.core.PpLexer.PpLexer attribute), 99
- fileName (cpip.core.PpTokeniser.PpTokeniser attribute), 105
- fileNameMap (cpip.core.FileIncludeGraph.FigVisitorFileSet attribute), 81
- fileNameSet (cpip.core.FileIncludeGraph.FigVisitorFileSet attribute), 81
- fileObj (cpip.core.IncludeHandler.FilePathOrigin attribute), 88
- filePath (cpip.core.IncludeHandler.FilePathOrigin attribute), 88
- filePathIn (cpip.util.DirWalk.FileInOut attribute), 111
- FilePathOrigin (class in cpip.core.IncludeHandler), 88
- filePathOut (cpip.util.DirWalk.FileInOut attribute), 111
- fileStack (cpip.core.FileIncludeStack.FileIncludeStack attribute), 85
- fileStack (cpip.core.PpLexer.PpLexer attribute), 99
- filterHeaderNames() (cpip.core.PpTokeniser.PpTokeniser method), 105
- finalise() (cpip.core.FileIncludeGraph.FigVisitorTreeNodeBase method), 82
- finalise() (cpip.core.FileIncludeStack.FileIncludeStack method), 85
- finalise() (cpip.core.IncludeHandler.CppIncludeStd method), 86
- finalise() (cpip.core.PpLexer.PpLexer method), 99
- finalise() (cpip.IncGraphSVG.SVGTreeNodeMain method), 69
- findLogic (cpip.core.FileIncludeGraph.FileIncludeGraph attribute), 83
- findLogic (cpip.core.IncludeHandler.CppIncludeStd attribute), 87
- findLogic (cpip.IncGraphSVG.SVGTreeNodeMain attribute), 69
- flip() (cpip.core.CppCond.ConditionalState method), 74
- flipAndAdd() (cpip.core.CppCond.ConditionalState method), 74
- ## G
- gccExtensions (cpip.CPIPMain.MainJobSpec attribute), 65
- gen() (cpip.util.BufGen.BufGen method), 108
- gen() (cpip.util.MaxMunchGen.MaxMunchGen method), 115
- genBigFirst() (in module cpip.util.DirWalk), 111
- genChildNodes() (cpip.core.FileIncludeGraph.FileIncludeGraph method), 83
- genColRowEvents() (cpip.util.DictTree.DictTreeHtmlTable method), 111
- genLexPptokenAndSeqWs() (cpip.core.PpTokeniser.PpTokeniser method), 105
- genMacros() (cpip.core.MacroEnv.MacroEnv method), 89
- genMacrosInScope() (cpip.core.MacroEnv.MacroEnv method), 90
- genMacrosOutOfScope() (cpip.core.MacroEnv.MacroEnv method), 90
- genRootPos() (cpip.plot.TreePlotTransform.TreePlotTransform method), 127
- genSweepDir() (cpip.plot.TreePlotTransform.TreePlotTransform method), 127
- genTokensKeywordPpDirective() (cpip.core.ItuToTokens.ItuToTokens method), 88
- getIsReplacement() (cpip.core.PpToken.PpToken method), 101
- getPrevWs() (cpip.core.PpToken.PpToken method), 101
- getReplace() (cpip.core.PpToken.PpToken method), 101
- getUndefMacro() (cpip.core.MacroEnv.MacroEnv method), 90
- graph (cpip.core.FileIncludeGraph.FileIncludeGraphRoot attribute), 84
- ## H
- handleUnclosedComment() (cpip.core.CppDiagnostic.PreprocessDiagnosticStd method), 80
- has() (cpip.util.StrTree.StrTree method), 115
- hasBeenTrue (cpip.core.CppCond.ConditionalState attribute), 74
- hasBeenTrueAtCurrentDepth() (cpip.core.CppCond.CppCond method), 75
- hasChild() (cpip.util.Tree.DuplexAdjacencyList method), 116
- hasLeadingWhitespace() (cpip.core.PpWhitespace.PpWhitespace method), 107
- hasMacro() (cpip.core.MacroEnv.MacroEnv method), 90
- hasParent() (cpip.util.Tree.DuplexAdjacencyList method), 116
- hasSetArea (cpip.plot.PlotNode.PlotNodeBbox attribute), 123
- helpMap (cpip.CPIPMain.MainJobSpec attribute), 65
- href() (cpip.TuIndexer.TuIndexer method), 72
- ## I
- id (cpip.util.XmlWrite.XmlStream attribute), 117



identifier (cpip.core.PpDefine.PpDefine attribute), 94  
IDENTIFIER\_SEPERATOR (cpip.core.PpDefine.PpDefine attribute), 93  
implementationDefined() (cpip.core.CppDiagnostic.PreprocessDiagnosticStd method), 80  
inc() (cpip.core.PpTokenCount.PpTokenCount method), 103  
incHandler (cpip.CPIPMain.MainJobSpec attribute), 66  
INCLUDE\_ORIGIN\_CODES (cpip.core.IncludeHandler.CppIncludeStd attribute), 86  
includeDepth (cpip.core.PpLexer.PpLexer attribute), 99  
includeDOT (cpip.CPIPMain.MainJobSpec attribute), 66  
includeFinish() (cpip.core.FileIncludeStack.FileIncludeStack method), 85  
includeHeaderName() (cpip.core.IncludeHandler.CppIncludeStd method), 87  
includeNextHeaderName() (cpip.core.IncludeHandler.CppIncludeStd method), 87  
includeStart() (cpip.core.FileIncludeStack.FileIncludeStack method), 85  
incPhysicalChildPos() (cpip.plot.TreePlotTransform.TreePlotTransform method), 127  
incRefCount() (cpip.core.PpDefine.PpDefine method), 94  
indexLB() (in module cpip.util.OaS), 115  
indexMatch() (in module cpip.util.OaS), 115  
indexPath (cpip.CPIPMain.PpProcessResult attribute), 66  
indexUB() (in module cpip.util.OaS), 115  
INITIAL\_REF\_COUNT (cpip.core.PpDefine.PpDefine attribute), 93  
initialTu() (cpip.core.IncludeHandler.CppIncludeStd method), 87  
initialTu() (cpip.core.IncludeHandler.CppIncludeStdin method), 87  
initialTu() (cpip.core.IncludeHandler.CppIncludeStdOs method), 87  
initialTu() (cpip.core.IncludeHandler.CppIncludeStringIO method), 88  
initLexPhase12() (cpip.core.PpTokeniser.PpTokeniser method), 106  
isAllMacroWhitespace() (cpip.core.PpWhitespace.PpWhitespace method), 107  
isAllWhitespace() (cpip.core.PpWhitespace.PpWhitespace method), 107  
isBreakingWhitespace() (cpip.core.PpWhitespace.PpWhitespace method), 107  
isCompiled() (cpip.core.CppCond.CppCondGraphVisitorConditionalLines method), 78  
isCompiled() (cpip.core.CppCond.LineConditionalInterpretation method), 79  
isComplete (cpip.core.CppCond.CppCondGraph attribute), 75  
isCond (cpip.core.PpToken.PpToken attribute), 101  
isCurrentlyDefined (cpip.core.PpDefine.PpDefine attribute), 94  
isDebug (cpip.core.CppDiagnostic.PreprocessDiagnosticStd attribute), 80  
isDefined() (cpip.core.MacroEnv.MacroEnv method), 90  
isIdentifier() (cpip.core.PpToken.PpToken method), 101  
isLiteral (cpip.core.PragmaHandler.PragmaHandlerABC attribute), 107  
isLiteral (cpip.core.PragmaHandler.PragmaHandlerEcho attribute), 108  
isObjectTypeMacro (cpip.core.PpDefine.PpDefine attribute), 94  
isReferenced (cpip.core.PpDefine.PpDefine attribute), 94  
isReplacement (cpip.core.PpToken.PpToken attribute), 101  
isSame() (cpip.core.PpDefine.PpDefine method), 94  
isTrue() (cpip.core.CppCond.CppCond method), 75  
isUnCond (cpip.core.PpToken.PpToken attribute), 101  
isValidRefefinition() (cpip.core.PpDefine.PpDefine method), 94  
isWs() (cpip.core.PpToken.PpToken method), 101  
ituPath (cpip.CPIPMain.PpProcessResult attribute), 66  
ItuToHtml (class in cpip.ItuToHtml), 70  
ItuToTokens (class in cpip.core.ItuToTokens), 88

## K

keepGoing (cpip.CPIPMain.MainJobSpec attribute), 66  
keys() (cpip.util.DictTree.DictTree method), 109

## L

LEN\_SOURCE\_CHARACTER\_SET (in module cpip.core.PpTokeniser), 104  
LEN\_WHITESPACE\_CHARACTER\_SET (in module cpip.core.PpWhitespace), 107  
lenBuf (cpip.util.BufGen.BufGen attribute), 108  
lenCommonPrefix() (in module cpip.util.CommonPrefix), 109  
LEX\_NEWLINE (in module cpip.core.PpWhitespace), 107  
LEX\_PPTOKEN\_TYPE\_ENUM\_RANGE (in module cpip.core.PpToken), 100  
LEX\_PPTOKEN\_TYPES (in module cpip.core.PpToken), 100  
LEX\_WHITESPACE (in module cpip.core.PpWhitespace), 107  
lexPhases\_0() (cpip.core.PpTokeniser.PpTokeniser method), 106  
lexPhases\_1() (cpip.core.PpTokeniser.PpTokeniser method), 106  
lexPhases\_2() (cpip.core.PpTokeniser.PpTokeniser method), 106  
line (cpip.core.PpDefine.PpDefine attribute), 94



- LineConditionalInterpretation (class in cpip.core.CppCond), 78
- lineNum (cpip.core.FileIncludeGraph.FigVisitorTreeNodeBase attribute), 82
- lineNum (cpip.core.PpLexer.PpLexer attribute), 99
- lineNum (cpip.core.PpToken.PpToken attribute), 101
- ListAsGenerator (class in cpip.util.ListGen), 113
- listIsEmpty (cpip.util.ListGen.ListAsGenerator attribute), 114
- literal() (cpip.util.XmlWrite.XmlStream method), 117
- LPAREN (cpip.core.PpDefine.PpDefine attribute), 93
- ## M
- macro() (cpip.core.MacroEnv.MacroEnv method), 90
- MacroEnv (class in cpip.core.MacroEnv), 89
- macroEnvironment (cpip.core.PpLexer.PpLexer attribute), 99
- macroHistory() (cpip.core.MacroEnv.MacroEnv method), 90
- macroHistoryMap() (cpip.core.MacroEnv.MacroEnv method), 90
- macroNotDefinedDependencies() (cpip.core.MacroEnv.MacroEnv method), 90
- macroNotDefinedDependencyNames() (cpip.core.MacroEnv.MacroEnv method), 90
- macroNotDefinedDependencyReferences() (cpip.core.MacroEnv.MacroEnv method), 90
- macros() (cpip.core.MacroEnv.MacroEnv method), 90
- main() (in module cpip.CPIPMMain), 66
- main() (in module cpip.FileStatus), 68
- MainJobSpec (class in cpip.CPIPMMain), 65
- MatrixRep (class in cpip.util.MatrixRep), 114
- MAX\_INCLUDE\_DEPTH (cpip.core.PpLexer.PpLexer attribute), 98
- MaxMunchGen (class in cpip.util.MaxMunchGen), 114
- merge() (cpip.core.PpToken.PpToken method), 102
- mightReplace() (cpip.core.MacroEnv.MacroEnv method), 91
- ## N
- NAME\_ENUM (in module cpip.core.PpToken), 100
- nameFromString() (in module cpip.util.XmlWrite), 117
- negateLastState() (cpip.core.CppCond.ConditionalState method), 74
- newPt() (in module cpip.plot.Coord), 119
- next() (cpip.core.PpTokeniser.PpTokeniser method), 106
- next() (cpip.util.ListGen.ListAsGenerator method), 114
- nextdcL() (cpip.plot.TreePlotTransform.TreePlotTransform method), 127
- numTokens (cpip.core.FileIncludeGraph.FileIncludeGraph attribute), 83
- numTokensIncChildren (cpip.core.FileIncludeGraph.FileIncludeGraph attribute), 83
- numTokensSig (cpip.core.FileIncludeGraph.FileIncludeGraph attribute), 83
- numTokensSigIncChildren (cpip.core.FileIncludeGraph.FileIncludeGraph attribute), 83
- numTrees() (cpip.core.FileIncludeGraph.FileIncludeGraphRoot method), 84
- ## O
- oElif() (cpip.core.CppCond.CppCond method), 75
- oElif() (cpip.core.CppCond.CppCondGraph method), 76
- oElif() (cpip.core.CppCond.CppCondGraphIfSection method), 77
- oElif() (cpip.core.CppCond.CppCondGraphNode method), 77
- oElse() (cpip.core.CppCond.CppCond method), 75
- oElse() (cpip.core.CppCond.CppCondGraph method), 76
- oElse() (cpip.core.CppCond.CppCondGraphIfSection method), 77
- oElse() (cpip.core.CppCond.CppCondGraphNode method), 77
- oEndif() (cpip.core.CppCond.CppCond method), 75
- oEndif() (cpip.core.CppCond.CppCondGraph method), 76
- oEndif() (cpip.core.CppCond.CppCondGraphIfSection method), 77
- oEndif() (cpip.core.CppCond.CppCondGraphNode method), 77
- oIf() (cpip.core.CppCond.CppCond method), 75
- oIf() (cpip.core.CppCond.CppCondGraph method), 76
- oIf() (cpip.core.CppCond.CppCondGraphIfSection method), 77
- oIf() (cpip.core.CppCond.CppCondGraphNode method), 77
- oIfdef() (cpip.core.CppCond.CppCond method), 75
- oIfdef() (cpip.core.CppCond.CppCondGraph method), 76
- oIfdef() (cpip.core.CppCond.CppCondGraphIfSection method), 77
- oIfdef() (cpip.core.CppCond.CppCondGraphNode method), 77
- oIfndef() (cpip.core.CppCond.CppCond method), 75
- oIfndef() (cpip.core.CppCond.CppCondGraph method), 76
- oIfndef() (cpip.core.CppCond.CppCondGraphIfSection method), 77
- oIfndef() (cpip.core.CppCond.CppCondGraphNode method), 77
- ON\_OFF\_SWITCH\_STATES (cpip.core.PragmaHandler.PragmaHandlerSTDC attribute), 108
- origin (cpip.core.IncludeHandler.FilePathOrigin attribute), 88

## P

- Pad (class in `cpip.plot.Coord`), 119
- parameters (`cpip.core.PpDefine.PpDefine` attribute), 94
- parents() (`cpip.util.Tree.DuplexAdjacencyList` method), 116
- partialTokenStream() (`cpip.core.CppDiagnostic.PreprocessDiagnosticKeepGoing` method), 79
- partialTokenStream() (`cpip.core.CppDiagnostic.PreprocessDiagnosticRaiseOnError` method), 80
- pathSplit() (in module `cpip.util.HtmlUtils`), 112
- pcLand() (`cpip.plot.PlotNode.PlotNodeBboxBoxy` method), 124
- pcLand() (`cpip.plot.PlotNode.PlotNodeBboxRoundy` method), 124
- pcRoll() (`cpip.plot.PlotNode.PlotNodeBboxBoxy` method), 124
- pcRoll() (`cpip.plot.PlotNode.PlotNodeBboxRoundy` method), 124
- pcStop() (`cpip.plot.PlotNode.PlotNodeBboxBoxy` method), 124
- pcStop() (`cpip.plot.PlotNode.PlotNodeBboxRoundy` method), 124
- pcTo() (`cpip.plot.PlotNode.PlotNodeBboxBoxy` method), 124
- pcTo() (`cpip.plot.PlotNode.PlotNodeBboxRoundy` method), 124
- pI() (`cpip.util.XmlWrite.XmlStream` method), 117
- PLACEMARKER (`cpip.core.PpDefine.PpDefine` attribute), 93
- pLineCol (`cpip.core.PpTokeniser.PpTokeniser` attribute), 106
- plotFinalise() (`cpip.IncGraphSVG.SVGTreeNodeMain` method), 69
- plotInitialise() (`cpip.IncGraphSVG.SVGTreeNodeMain` method), 69
- PlotNodeBbox (class in `cpip.plot.PlotNode`), 122
- PlotNodeBboxBoxy (class in `cpip.plot.PlotNode`), 123
- PlotNodeBboxRoundy (class in `cpip.plot.PlotNode`), 124
- plotPointCentre() (`cpip.plot.PlotNode.PlotNodeBbox` method), 123
- plotPointSelf() (`cpip.plot.PlotNode.PlotNodeBbox` method), 123
- pop() (`cpip.core.PpTokenCount.PpTokenCountStack` method), 103
- positiveSweepDir (`cpip.plot.TreePlotTransform.TreePlotTransform` attribute), 127
- postIncChildLogicalPos() (`cpip.plot.TreePlotTransform.TreePlotTransform` method), 127
- PpDefine (class in `cpip.core.PpDefine`), 92
- PpLexer (class in `cpip.core.PpLexer`), 97
- PpProcessResult (class in `cpip.CPIPMain`), 66
- ppt (`cpip.core.FileIncludeStack.FileIncludeStack` attribute), 85
- PpToken (class in `cpip.core.PpToken`), 101
- PpTokenCount (class in `cpip.core.PpTokenCount`), 102
- PpTokenCountStack (class in `cpip.core.PpTokenCount`), 103
- PpTokeniser (class in `cpip.core.PpTokeniser`), 104
- ppTokenise() (`cpip.core.PpLexer.PpLexer` method), 99
- PpWhitespace (class in `cpip.core.PpWhitespace`), 107
- pragma() (`cpip.core.PragmaHandler.PragmaHandlerABC` method), 107
- pragma() (`cpip.core.PragmaHandler.PragmaHandlerEcho` method), 108
- pragma() (`cpip.core.PragmaHandler.PragmaHandlerNull` method), 108
- pragma() (`cpip.core.PragmaHandler.PragmaHandlerSTDC` method), 108
- pragmaHandler (`cpip.CPIPMain.MainJobSpec` attribute), 66
- PragmaHandlerABC (class in `cpip.core.PragmaHandler`), 107
- PragmaHandlerEcho (class in `cpip.core.PragmaHandler`), 107
- PragmaHandlerNull (class in `cpip.core.PragmaHandler`), 108
- PragmaHandlerSTDC (class in `cpip.core.PragmaHandler`), 108
- preceedsNewline() (`cpip.core.PpWhitespace.PpWhitespace` method), 107
- preDefMacros (`cpip.CPIPMain.MainJobSpec` attribute), 66
- preIncChildLogicalPos() (`cpip.plot.TreePlotTransform.TreePlotTransform` method), 127
- preIncFiles (`cpip.CPIPMain.MainJobSpec` attribute), 66
- PreprocessDiagnosticKeepGoing (class in `cpip.core.CppDiagnostic`), 79
- PreprocessDiagnosticRaiseOnError (class in `cpip.core.CppDiagnostic`), 79
- PreprocessDiagnosticStd (class in `cpip.core.CppDiagnostic`), 79
- preprocessDirToOutput() (in module `cpip.CPIPMain`), 66
- preProcessFilesMP() (in module `cpip.CPIPMain`), 66
- preprocessFileToOutput() (in module `cpip.CPIPMain`), 66
- preprocessFileToOutputNoExcept() (in module `cpip.CPIPMain`), 66
- PREPROCESSING\_DIRECTIVES (in module `cpip.core.PpLexer`), 97
- prevdcL() (`cpip.plot.TreePlotTransform.TreePlotTransform` method), 127
- prevWs (`cpip.core.PpToken.PpToken` attribute), 102
- processCppCondGrphToHtml() (in module `cpip.CppCondGraphToHtml`), 67
- processDir() (`cpip.FileStatus.FileInfoSet` method), 68
- processIncGraphToSvg() (in module `cpip.IncGraphSVGBase`), 70
- processIncGraphToXml() (in module `cpip.IncGraphToXml`), 70

cpip.IncGraphXML), 70  
 processMacroHistoryToHtml() (in module  
 cpip.MacroHistoryHtml), 71  
 processPath() (cpip.FileStatus.FileInfoSet method), 68  
 processTuToHtml() (in module cpip.Tu2Html), 71  
 Pt (class in cpip.plot.Coord), 119  
 pt() (cpip.plot.TreePlotTransform.TreePlotTransform  
 method), 127  
 push() (cpip.core.PpTokenCount.PpTokenCountStack  
 method), 103

## R

RAISE\_ON\_ERROR (in module cpip.util.XmlWrite),  
 116  
 reduceToksToHeaderName()  
 (cpip.core.PpTokeniser.PpTokeniser method),  
 106  
 refCount (cpip.core.PpDefine.PpDefine attribute), 94  
 referencedMacroIdentifiers()  
 (cpip.core.MacroEnv.MacroEnv method),  
 91  
 refFileLineColS (cpip.core.PpDefine.PpDefine attribute),  
 95  
 remove() (cpip.util.DictTree.DictTree method), 109  
 replace() (cpip.core.MacroEnv.MacroEnv method), 91  
 replace() (cpip.util.BufGen.BufGen method), 108  
 replaceArgumentList() (cpip.core.PpDefine.PpDefine  
 method), 95  
 replacements (cpip.core.PpDefine.PpDefine attribute), 95  
 replacementTokens (cpip.core.PpDefine.PpDefine at-  
 tribute), 95  
 replaceNewLine() (cpip.core.PpToken.PpToken method),  
 102  
 replaceObjectStyleMacro()  
 (cpip.core.PpDefine.PpDefine method), 95  
 replaceTokens (cpip.core.PragmaHandler.PragmaHandlerABG  
 attribute), 107  
 replaceTokens (cpip.core.PragmaHandler.PragmaHandlerEcho  
 attribute), 108  
 replaceTokens (cpip.core.PragmaHandler.PragmaHandlerNull  
 attribute), 108  
 replaceTokens (cpip.core.PragmaHandler.PragmaHandlerSTDC  
 attribute), 108  
 resetTokType() (cpip.core.PpTokeniser.PpTokeniser  
 method), 106  
 retArgumentListTokens() (cpip.core.PpDefine.PpDefine  
 method), 95  
 retBranches() (cpip.core.FileIncludeGraph.FileIncludeGraph  
 method), 83  
 retFileCountMap() (in module cpip.CPIPMain), 66  
 retHtmlFileLink() (in module cpip.util.HtmlUtils), 112  
 retHtmlFileName() (in module cpip.util.HtmlUtils), 112  
 retLatestBranch() (cpip.core.FileIncludeGraph.FileIncludeGraph  
 method), 83

retLatestBranchDepth() (cpip.core.FileIncludeGraph.FileIncludeGraph  
 method), 84  
 retLatestBranchPairs() (cpip.core.FileIncludeGraph.FileIncludeGraph  
 method), 84  
 retLatestLeaf() (cpip.core.FileIncludeGraph.FileIncludeGraph  
 method), 84  
 retLatestNode() (cpip.core.FileIncludeGraph.FileIncludeGraph  
 method), 84  
 retOptionMap() (in module cpip.CPIPMain), 67  
 retStrList() (cpip.core.CppCond.CppCondGraphNode  
 method), 77  
 RPAREN (cpip.core.PpDefine.PpDefine attribute), 93

## S

scale() (cpip.plot.Coord.Dim method), 118  
 scale() (cpip.plot.Coord.Pt method), 119  
 set\_\_FILE\_\_() (cpip.core.MacroEnv.MacroEnv method),  
 91  
 set\_\_LINE\_\_() (cpip.core.MacroEnv.MacroEnv method),  
 91  
 setColRowSpan() (cpip.util.DictTree.DictTreeHtmlTable  
 method), 111  
 setIsCond() (cpip.core.PpToken.PpToken method), 102  
 setIsReplacement() (cpip.core.PpToken.PpToken  
 method), 102  
 setPrevWs() (cpip.core.PpToken.PpToken method), 102  
 setReplace() (cpip.core.PpToken.PpToken method), 102  
 setTokenCounter() (cpip.core.FileIncludeGraph.FileIncludeGraph  
 method), 84  
 shrinkWs() (cpip.core.PpToken.PpToken method), 102  
 sideEffect() (cpip.util.MatrixRep.MatrixRep method),  
 114  
 SINGLE\_SPACE (cpip.core.PpToken.PpToken attribute),  
 101  
 size (cpip.FileStatus.FileInfo attribute), 68  
 slice() (cpip.util.BufGen.BufGen method), 109  
 sliceNonWhitespace() (cpip.core.PpWhitespace.PpWhitespace  
 method), 107  
 sliceWhitespace() (cpip.core.PpWhitespace.PpWhitespace  
 method), 107  
 sloc (cpip.FileStatus.FileInfo attribute), 68  
 splitLine() (in module cpip.MacroHistoryHtml), 71  
 splitLineToList() (in module cpip.MacroHistoryHtml), 71  
 stackDepth (cpip.core.CppCond.CppCond attribute), 75  
 startChildrenLogicalPos()  
 (cpip.plot.TreePlotTransform.TreePlotTransform  
 method), 128  
 startElement() (cpip.util.XmlWrite.XmlStream method),  
 117  
 state (cpip.core.CppCond.ConditionalState attribute), 74  
 StateConstExprFileLine (in module cpip.core.CppCond),  
 79  
 STDC (cpip.core.PragmaHandler.PragmaHandlerSTDC  
 attribute), 108

- strIdentPlusParam() (cpip.core.PpDefine.PpDefine method), 96
- STRINGIZE\_WHITESPACE\_CHAR (cpip.core.PpDefine.PpDefine attribute), 93
- strReplacements() (cpip.core.PpDefine.PpDefine method), 96
- StrTree (class in cpip.util.StrTree), 115
- subst() (cpip.core.PpToken.PpToken method), 102
- substAltToken() (cpip.core.PpTokeniser.PpTokeniser method), 106
- SVGCircle (class in cpip.plot.SVGWriter), 124
- SVGEllipse (class in cpip.plot.SVGWriter), 125
- SVGGroup (class in cpip.plot.SVGWriter), 125
- SVGLine (class in cpip.plot.SVGWriter), 125
- SVGPointList (class in cpip.plot.SVGWriter), 125
- SVGPolygon (class in cpip.plot.SVGWriter), 125
- SVGPolyline (class in cpip.plot.SVGWriter), 125
- SVGRect (class in cpip.plot.SVGWriter), 125
- SVGText (class in cpip.plot.SVGWriter), 125
- SVGTreeNodeMain (class in cpip.IncGraphSVG), 69
- SVGWriter (class in cpip.plot.SVGWriter), 125
- ## T
- t (cpip.core.PpToken.PpToken attribute), 102
- tdcL() (cpip.plot.TreePlotTransform.TreePlotTransform method), 128
- tdcP() (cpip.plot.TreePlotTransform.TreePlotTransform method), 128
- tokenCount() (cpip.core.PpTokenCount.PpTokenCount method), 103
- tokenCounter (cpip.core.FileIncludeGraph.FileIncludeGraph attribute), 84
- tokenCounter (cpip.core.PpDefine.PpDefine attribute), 96
- tokenCounter (cpip.IncGraphSVG.SVGTreeNodeMain attribute), 69
- tokenCounter() (cpip.core.FileIncludeStack.FileIncludeStack method), 85
- tokenCounterAdd() (cpip.core.FileIncludeStack.FileIncludeStack method), 85
- tokenCounterAdd() (cpip.core.FileIncludeStack.FileIncludeStack method), 85
- tokenCounterChildren (cpip.IncGraphSVG.SVGTreeNodeMain attribute), 69
- tokenCounterTotal (cpip.IncGraphSVG.SVGTreeNodeMain attribute), 69
- tokenCountInc() (cpip.core.FileIncludeStack.FileIncludeStack method), 85
- tokenCountInc() (cpip.core.FileIncludeStack.FileIncludeStack method), 85
- tokenCountNonWs() (cpip.core.PpTokenCount.PpTokenCount method), 103
- tokensConsumed (cpip.core.PpDefine.PpDefine attribute), 96
- tokensStr() (in module cpip.core.PpToken), 102
- tokenTypesAndCounts() (cpip.core.PpTokenCount.PpTokenCount method), 103
- tokEnumToktype (cpip.core.PpToken.PpToken attribute), 102
- tokToktype (cpip.core.PpToken.PpToken attribute), 102
- total\_bytes (cpip.CPIPMMain.PpProcessResult attribute), 66
- total\_files (cpip.CPIPMMain.PpProcessResult attribute), 66
- total\_lines (cpip.CPIPMMain.PpProcessResult attribute), 66
- totalAll (cpip.core.PpTokenCount.PpTokenCount attribute), 103
- totalAllConditional (cpip.core.PpTokenCount.PpTokenCount attribute), 103
- totalAllUnconditional (cpip.core.PpTokenCount.PpTokenCount attribute), 103
- translateTokensToString() (cpip.core.ConstantExpression.ConstantExpression method), 72
- Tree (class in cpip.util.Tree), 116
- tree() (cpip.core.FileIncludeGraph.FigVisitorTree method), 81
- treeChildParent() (cpip.util.Tree.DuplexAdjacencyList method), 116
- treeParentChild() (cpip.util.Tree.DuplexAdjacencyList method), 116
- TreePlotTransform (class in cpip.plot.TreePlotTransform), 127
- TRIGRAPH\_PREFIX (in module cpip.core.PpTokeniser), 106
- TRIGRAPH\_SIZE (in module cpip.core.PpTokeniser), 106
- TRIGRAPH\_TABLE (in module cpip.core.PpTokeniser), 106
- tt (cpip.core.PpToken.PpToken attribute), 102
- tuFileId (cpip.core.PpLexer.PpLexer attribute), 100
- TuIndexer (class in cpip.TuIndexer), 72
- tuIndexFileName (cpip.CPIPMMain.PpProcessResult attribute), 66
- ## U
- undef() (cpip.core.MacroEnv.MacroEnv method), 91
- undef() (cpip.core.PpDefine.PpDefine method), 96
- undefFileId (cpip.core.PpDefine.PpDefine attribute), 96
- undefined() (cpip.core.CppDiagnostic.PreprocessDiagnosticKeepGoing method), 79
- undefined() (cpip.core.CppDiagnostic.PreprocessDiagnosticStd method), 80
- undefLine (cpip.core.PpDefine.PpDefine attribute), 96
- units() (in module cpip.plot.Coord), 118
- UNNAMED\_FILE\_NAME (in module cpip.core.PpLexer), 100

unspecified() (cpip.core.CppDiagnostic.PreprocessDiagnosticStd method), 80

## V

validateCpStack() (cpip.core.IncludeHandler.CppIncludeStd method), 87

value() (cpip.util.DictTree.DictTree method), 109

values() (cpip.util.DictTree.DictTree method), 109

values() (cpip.util.StrTree.StrTree method), 115

VARIABLE\_ARGUMENT\_IDENTIFIER (cpip.core.PpDefine.PpDefine attribute), 93

VARIABLE\_ARGUMENT\_SUBSTITUTE (cpip.core.PpDefine.PpDefine attribute), 93

visit() (cpip.core.CppCond.CppCondGraph method), 77

visit() (cpip.core.CppCond.CppCondGraphIfSection method), 77

visit() (cpip.core.CppCond.CppCondGraphNode method), 78

visitGraph() (cpip.core.FileIncludeGraph.FigVisitorBase method), 81

visitGraph() (cpip.core.FileIncludeGraph.FigVisitorFileSet method), 81

visitGraph() (cpip.core.FileIncludeGraph.FigVisitorTree method), 81

visitGraph() (cpip.CPIPMain.FigVisitorDot method), 65

visitGraph() (cpip.CPIPMain.FigVisitorLargestCommonPrefix method), 65

visitPost() (cpip.core.CppCond.CppCondGraphVisitorBase method), 78

visitPost() (cpip.CppCondGraphToHtml.CcgVisitorToHtml method), 67

visitPre() (cpip.core.CppCond.CppCondGraphVisitorBase method), 78

visitPre() (cpip.core.CppCond.CppCondGraphVisitorConditional method), 78

visitPre() (cpip.CppCondGraphToHtml.CcgVisitorToHtml method), 67

visitCharsAndSpan() (in module cpip.util.HtmlUtils), 112

writeCSS() (cpip.util.XmlWrite.XmlStream method), 117

writeCssForFile() (in module cpip.TokenCss), 71

writeCssToDir() (in module cpip.TokenCss), 71

writeDictTreeAsTable() (in module cpip.util.HtmlUtils), 112

writeECMAScript() (cpip.util.XmlWrite.XmlStream method), 117

writeFileListAsTable() (in module cpip.util.HtmlUtils), 112

writeFileListTrippleAsTable() (in module cpip.util.HtmlUtils), 112

writeFilePathsAsTable() (in module cpip.util.HtmlUtils), 112

writeHeader() (cpip.FileStatus.FileInfo method), 68

writeHtmlFileAnchor() (in module cpip.util.HtmlUtils), 113

writeHtmlFileLink() (in module cpip.util.HtmlUtils), 113

writeIndexHtml() (in module cpip.CPIPMain), 67

writePreamble() (cpip.IncGraphSVG.SVGTreeNodeMain method), 69

writeTuIndexHtml() (in module cpip.CPIPMain), 67

## X

xmlSpacePreserve() (cpip.util.XmlWrite.XmlStream method), 117

XmlStream (class in cpip.util.XmlWrite), 116

## Y

youngestChild (cpip.util.Tree.Tree attribute), 116

## Z

zeroBaseUnitsBox() (in module cpip.plot.Coord), 119

zeroBaseUnitsDim() (in module cpip.plot.Coord), 119

zeroBaseUnitsPad() (in module cpip.plot.Coord), 119

zeroBaseUnitsPt() (in module cpip.plot.Coord), 119

## W

warning() (cpip.core.CppDiagnostic.PreprocessDiagnosticStd method), 80

width (cpip.plot.PlotNode.PlotNodeBbox attribute), 123

WORD\_REPLACE\_MAP (cpip.core.PpToken.PpToken attribute), 101

write() (cpip.FileStatus.FileInfo method), 68

write() (cpip.FileStatus.FileInfoSet method), 68

writeAltTextAndMouseOverRect() (cpip.IncGraphSVG.SVGTreeNodeMain method), 69

writeCDATA() (cpip.util.XmlWrite.XmlStream method), 117