CPIP Documentation

Release 0.9.8rc0

Paul Ross

Nov 17, 2017

Contents

Contents:

CHAPTER 1

CPIP

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.

1.1 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

1.2 Installation

Install cpip from PyPi:

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

This will give you access to the command line entry point `cpipmain`, to check this:

```
(CPIP) $ cpipmain --help
...
```

There are other installation methods including directly from source.

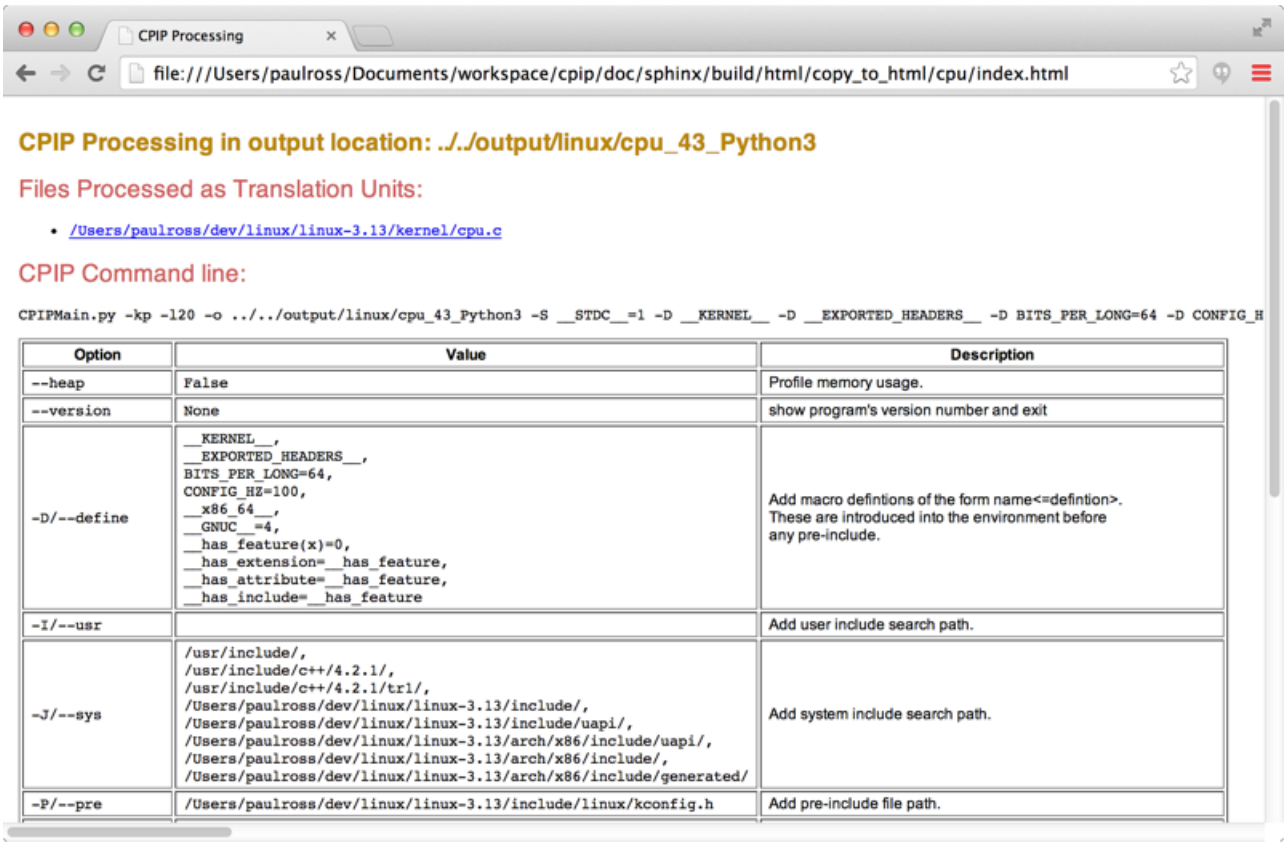
CHAPTER 2

Visualising Preprocessing

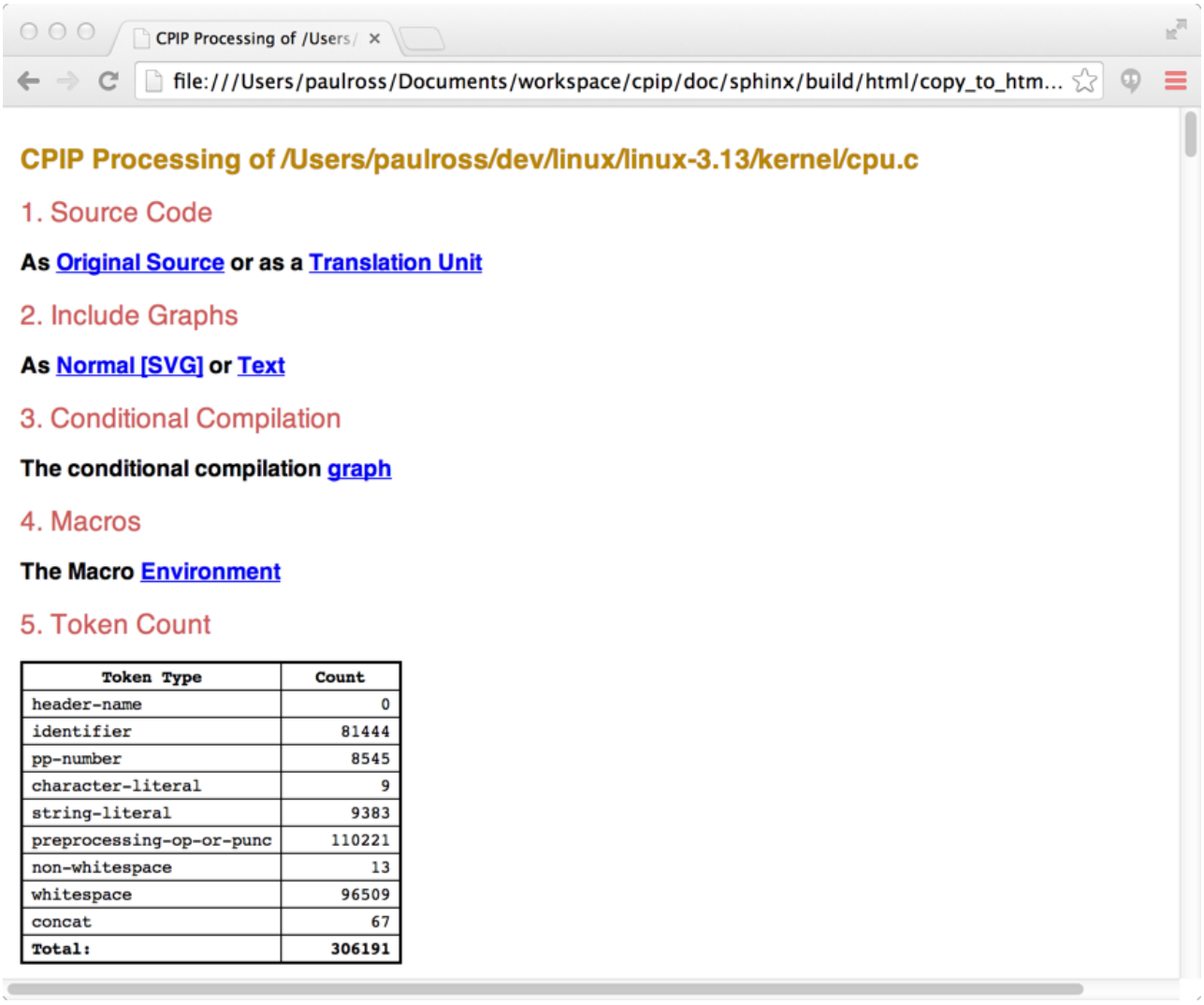
The top level entry point `cpipmain` (the script `CPIPMain.py`) acts like a preprocessor that generates HTML and SVG output for a source code file or directory. This output makes it easy to understand what the preprocessor is doing to your source.

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[1], this has a link to to preprocessing pages for that file:

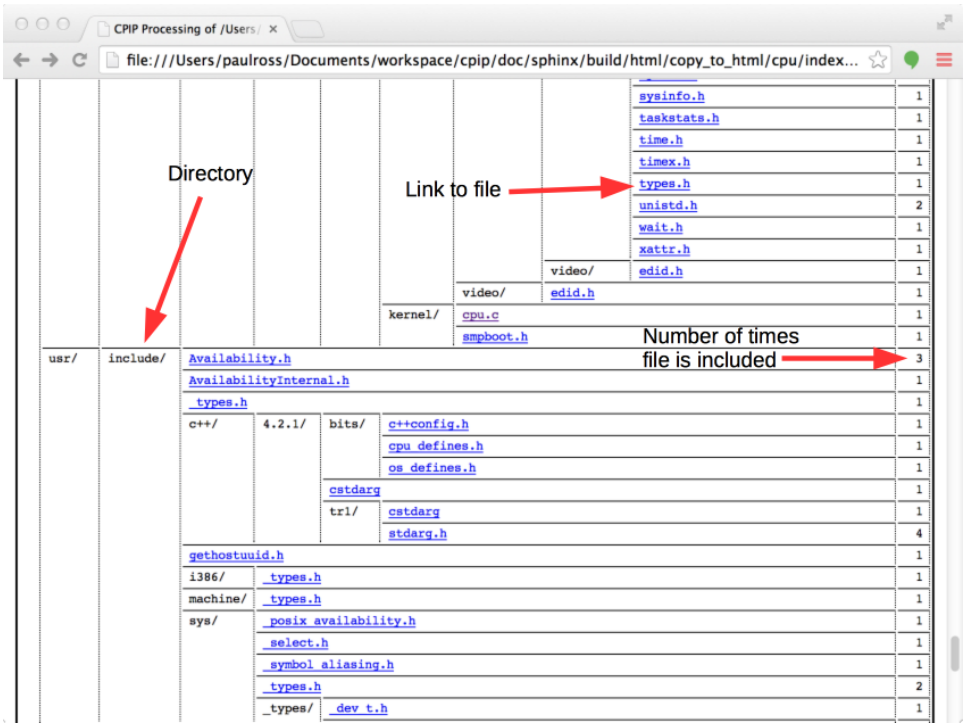
[1] This was invoked by:



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:

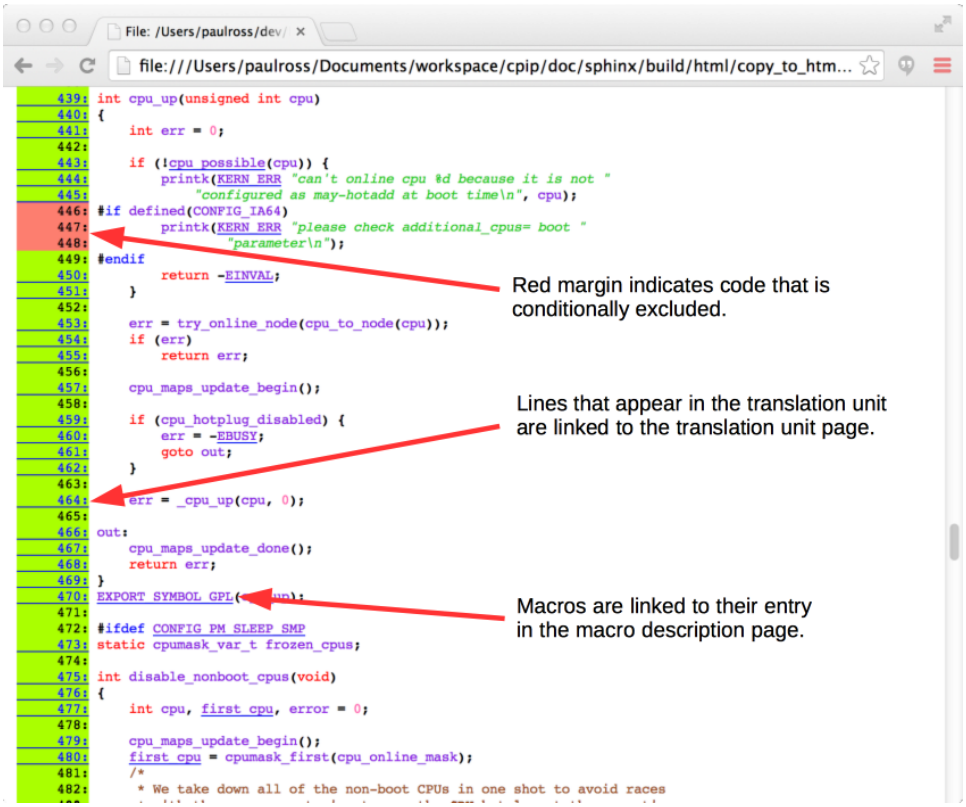


Lower down this page is a table of files that were involved in preprocessing:

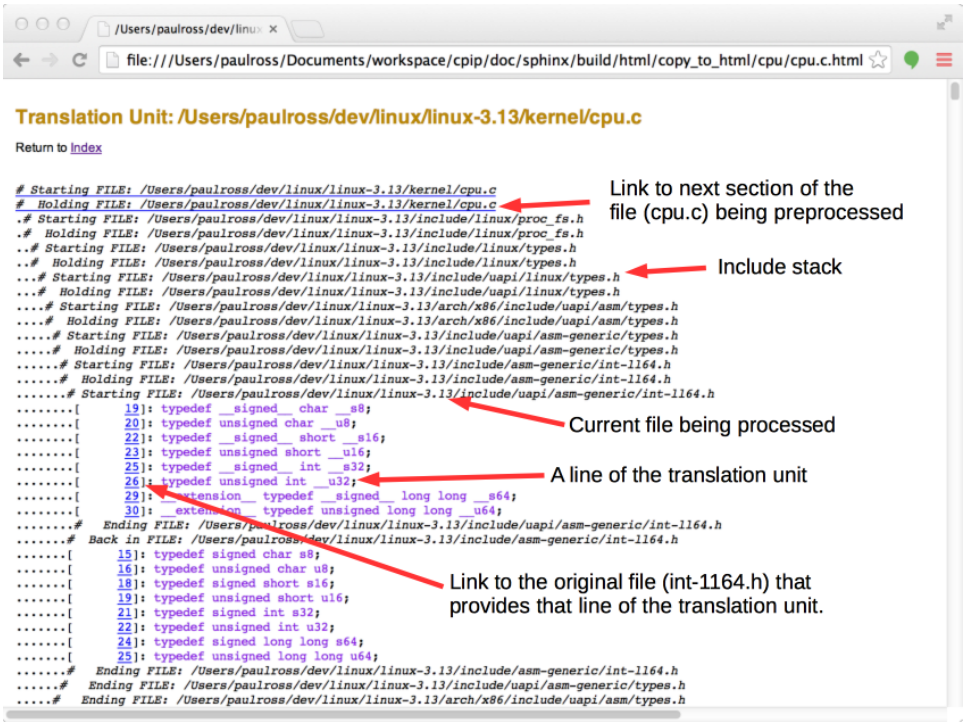


2.1 Visualising the Source Code

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



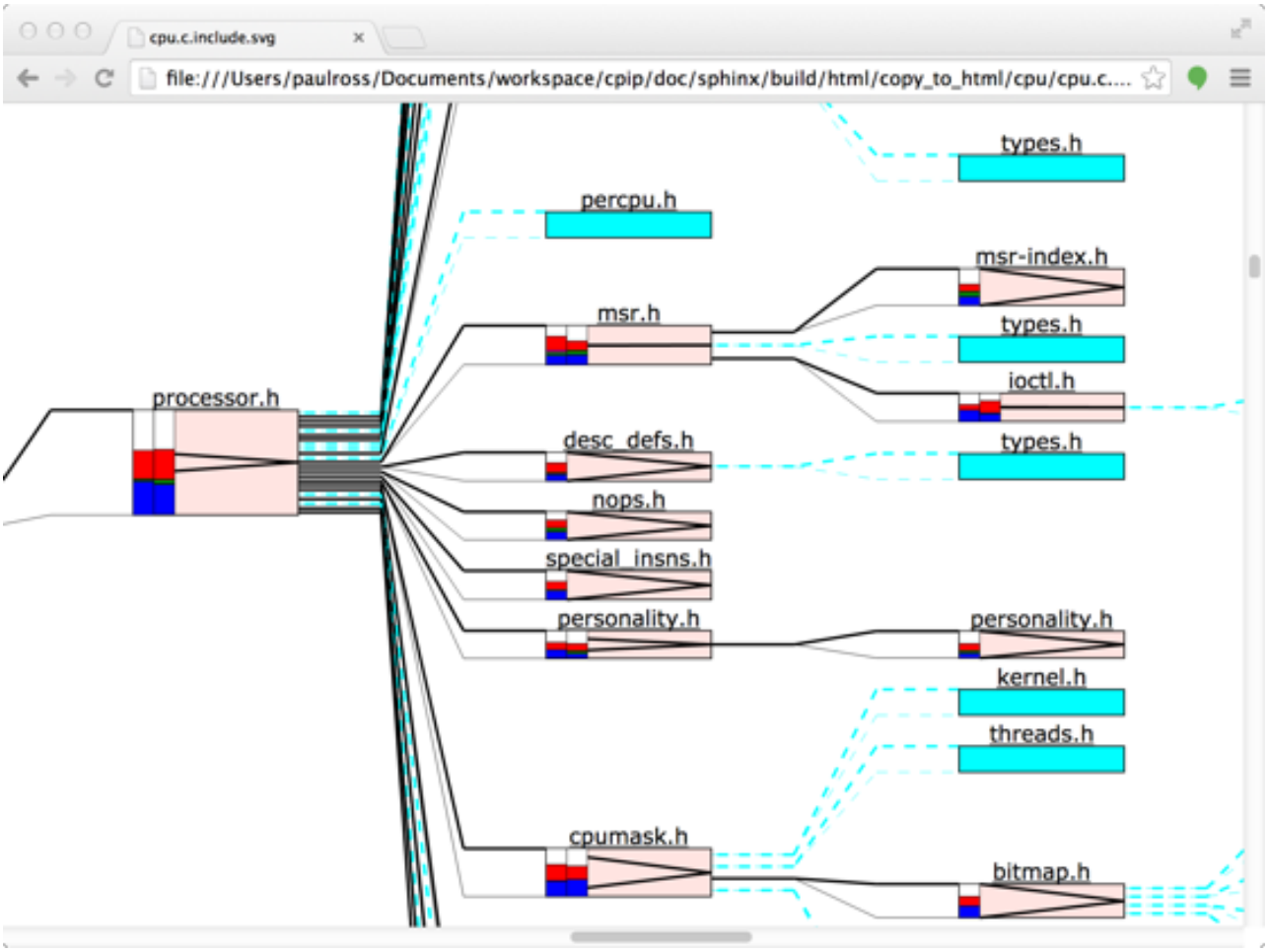
The `cpu.c` landing page link "Translation Unit" takes you to a page that shows the complete translation unit of `cpu.c` (i.e. incorporating all the `#include` files). This page is annotated so that you can understand what part of the translation unit comes from which file.



2.2 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:

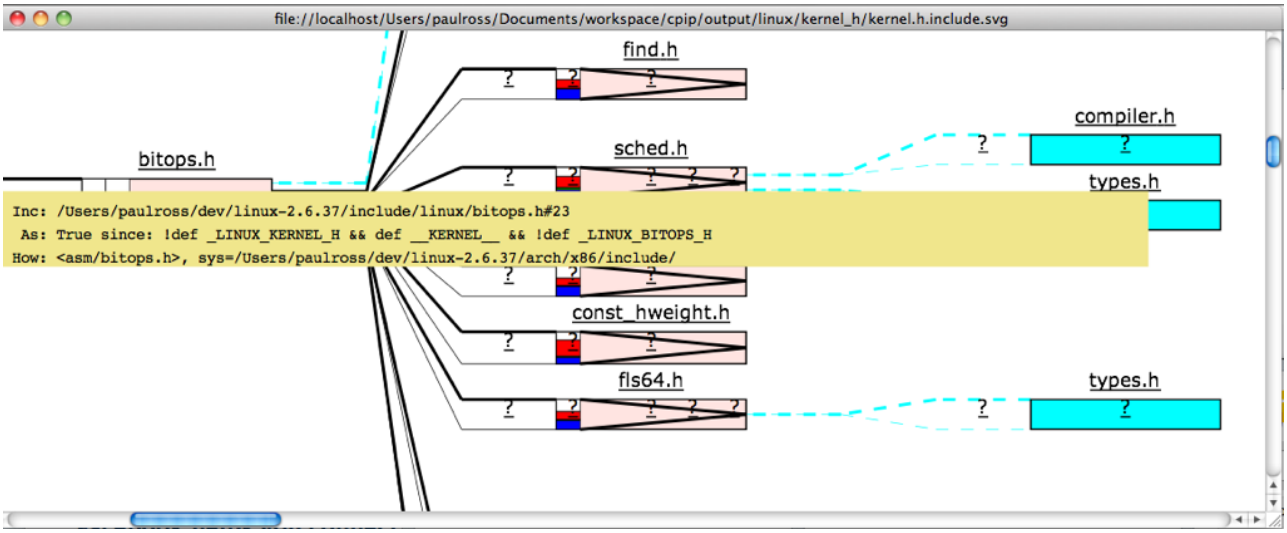


The page is dynamic and hovering over various areas provides more information:

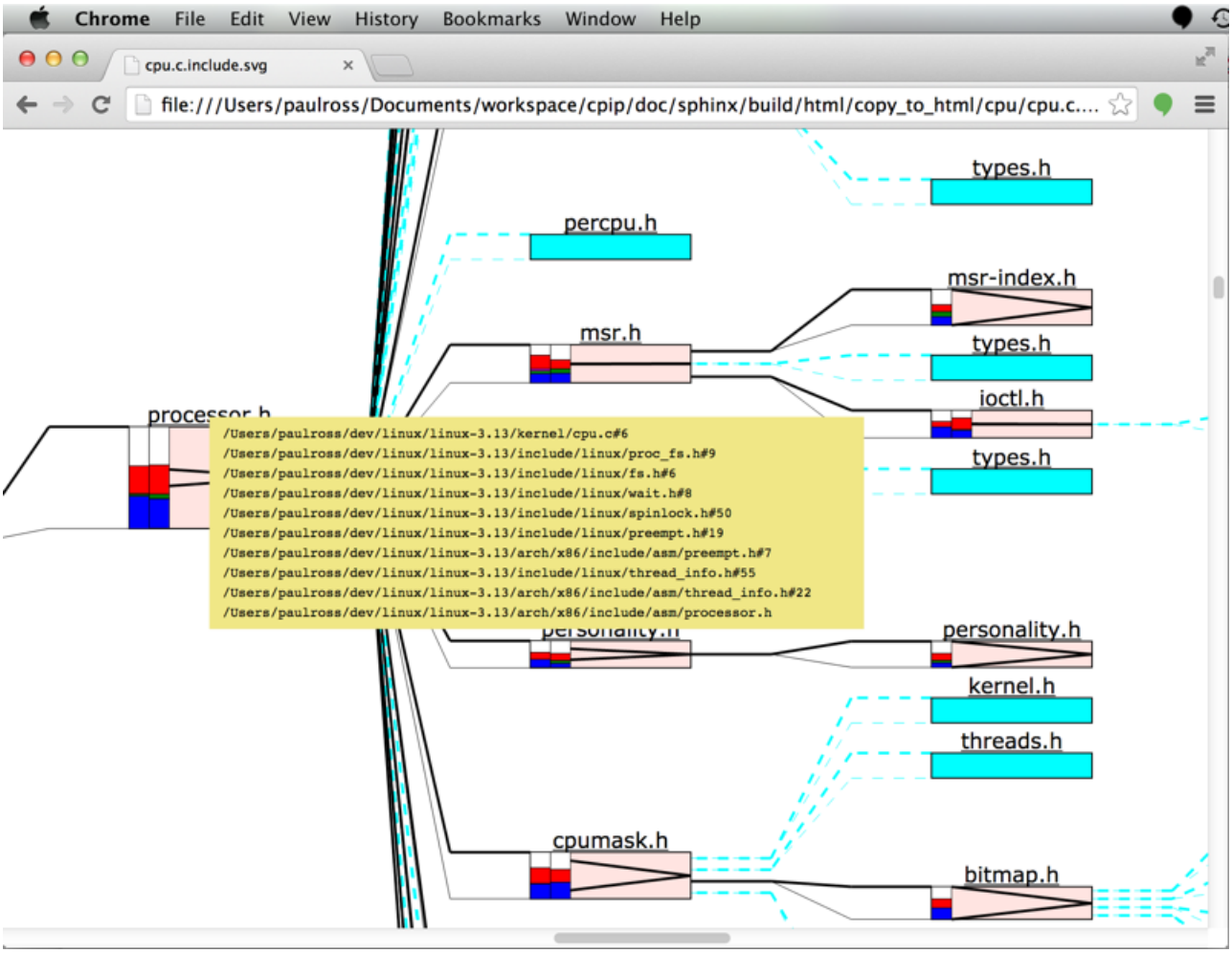
2.2.1 How and Why the File was Included

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:

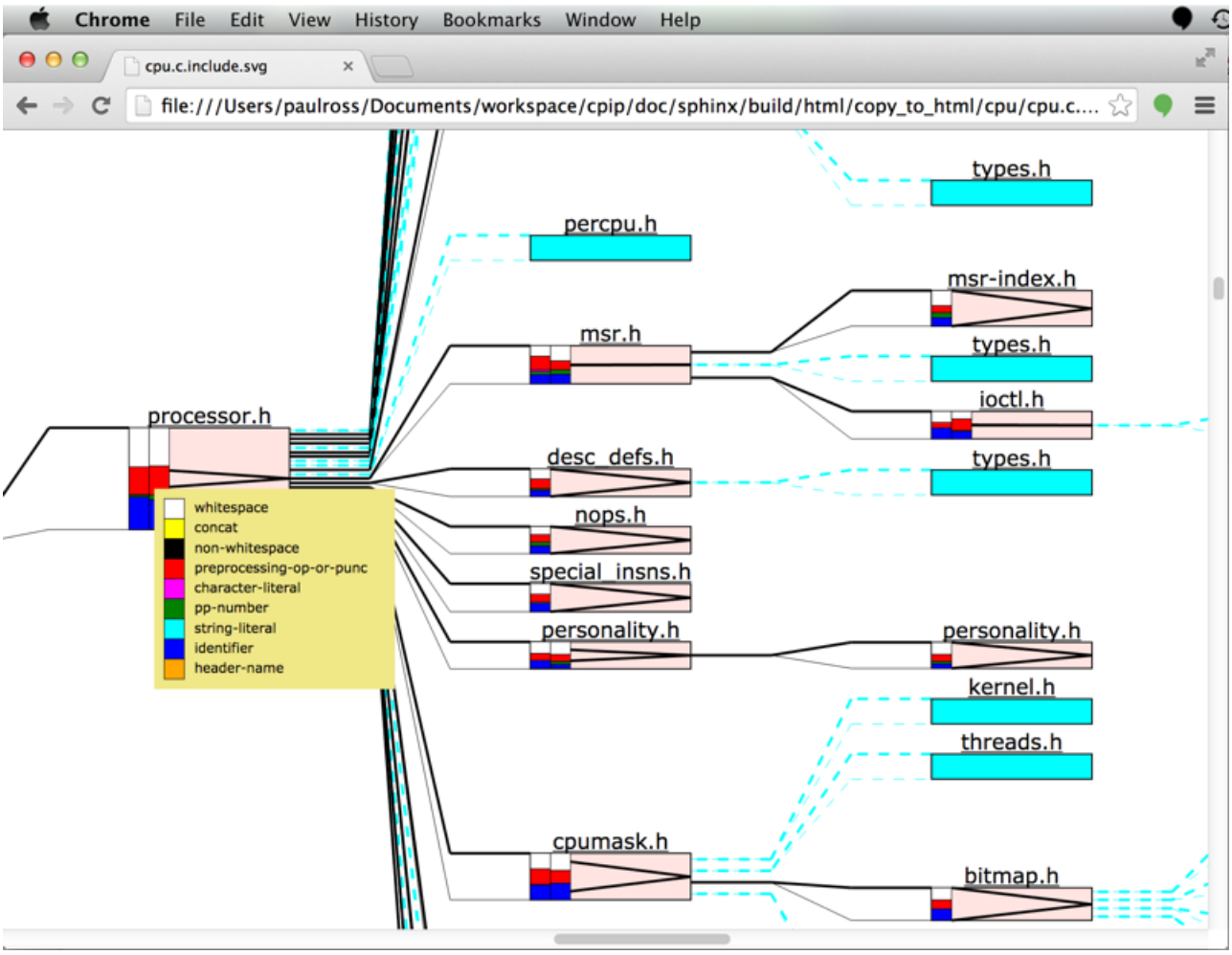
- Inc: The filename and line number of the `#include` directive.
- As: The conditional compilation state at the point of the `#include` directive.
- How: The text of the `#include` directive followed by the directory that this file was found in, this directory is prefixed by `sys=` for a system include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



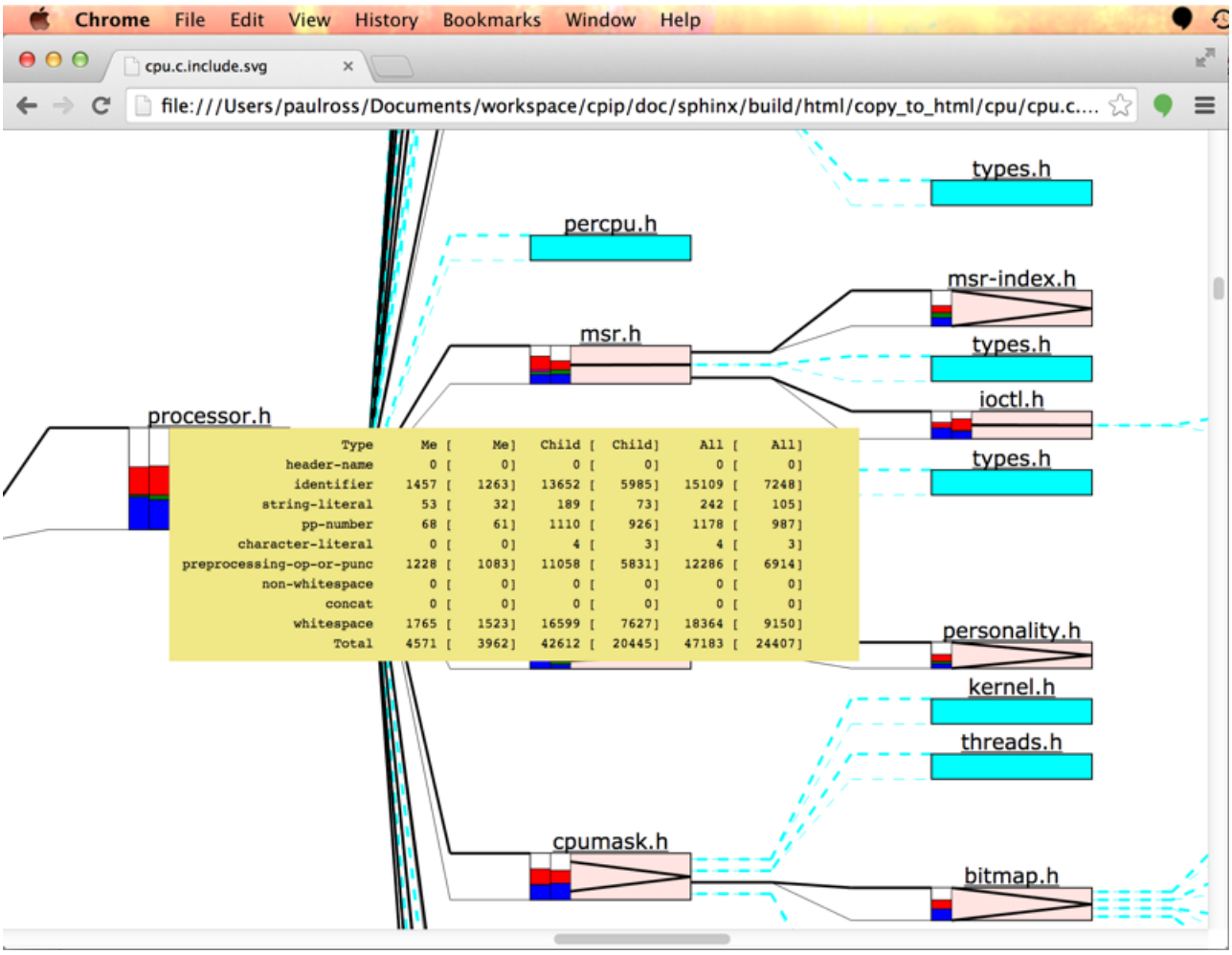
Hovering over the filename above the file box shows the file stack (children are below parents).



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:

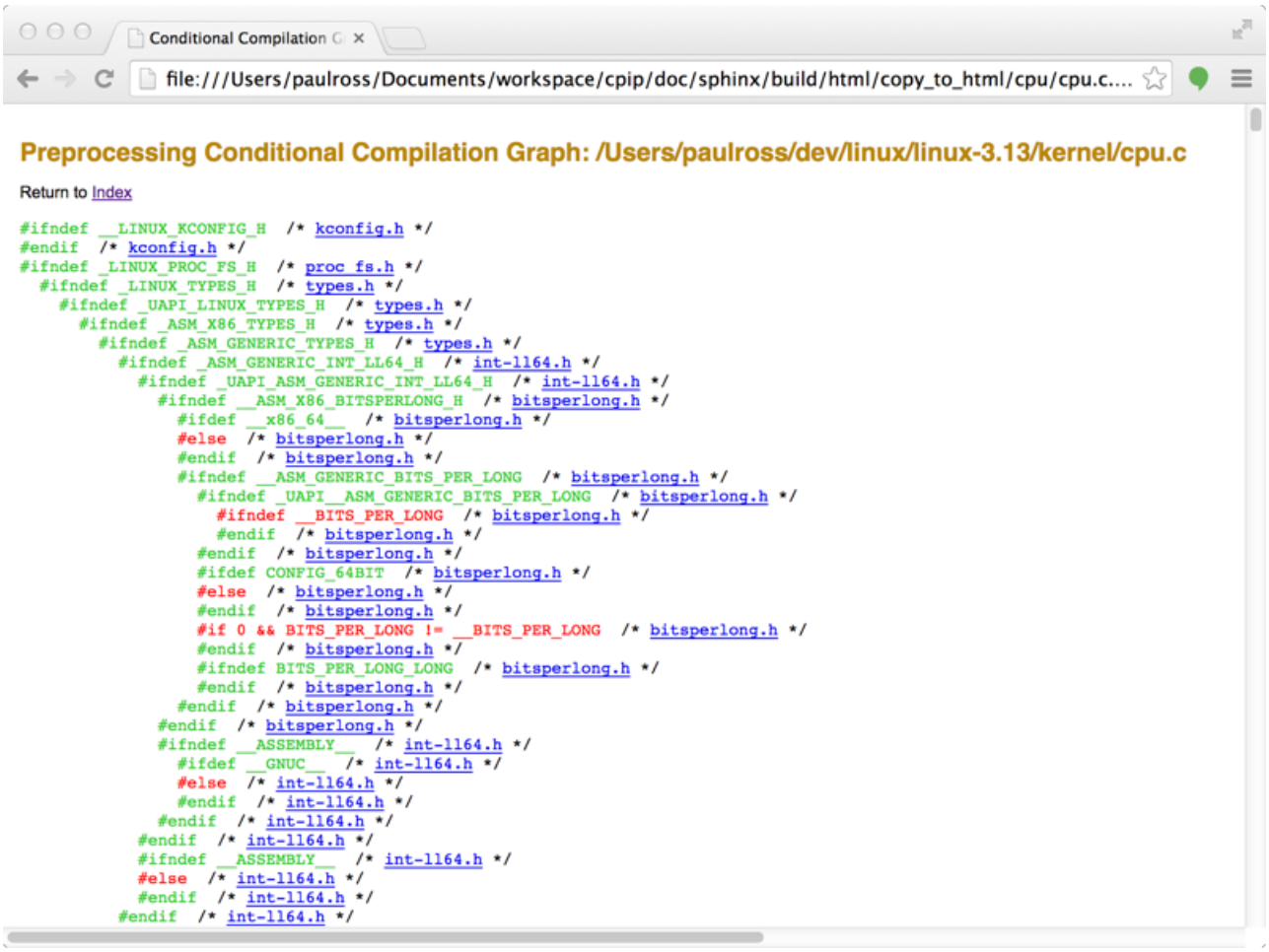


To see the actual count of preprocessing tokens hover over the file box:



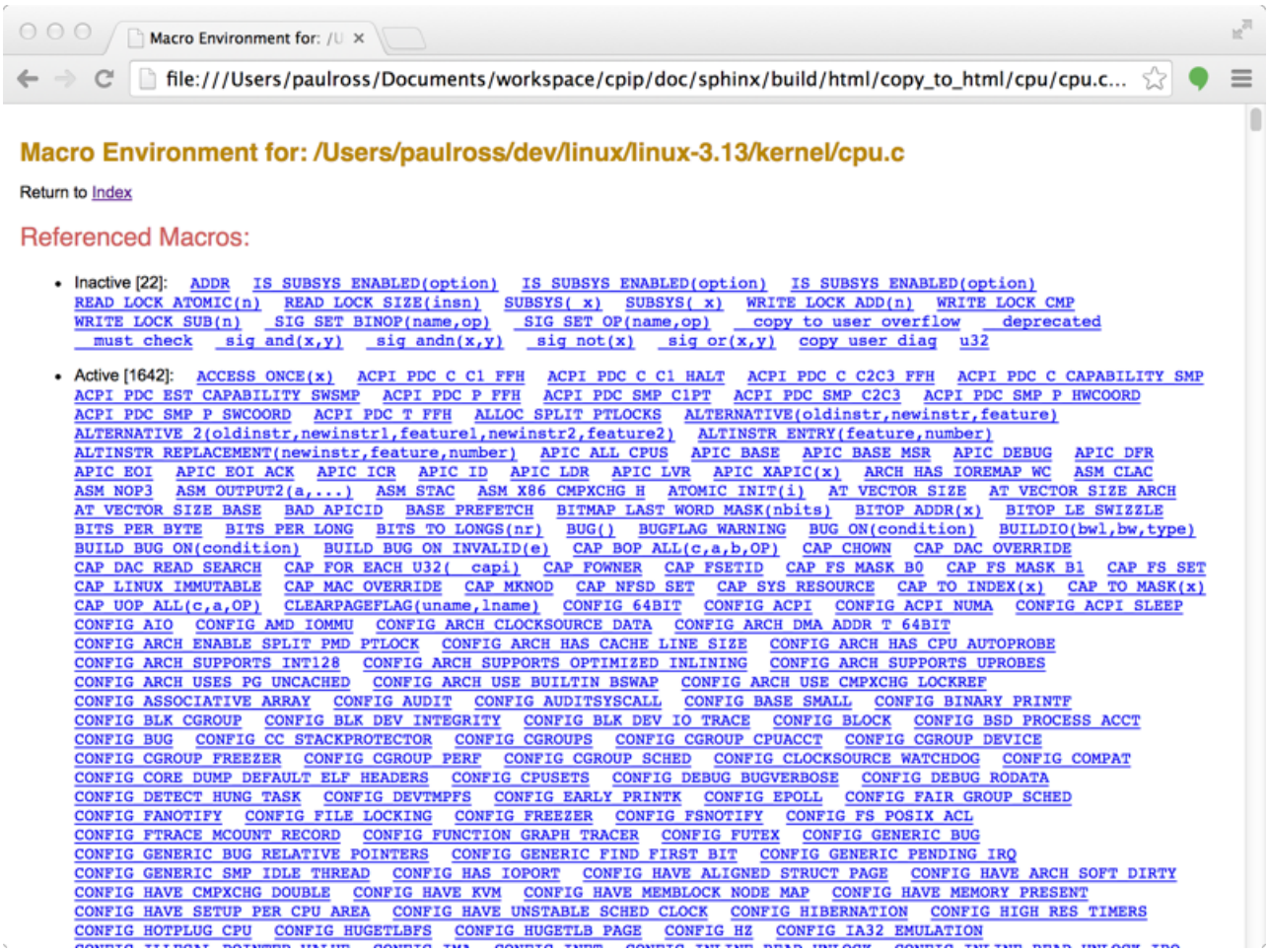
2.3 Visualising Conditional Compilation

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.



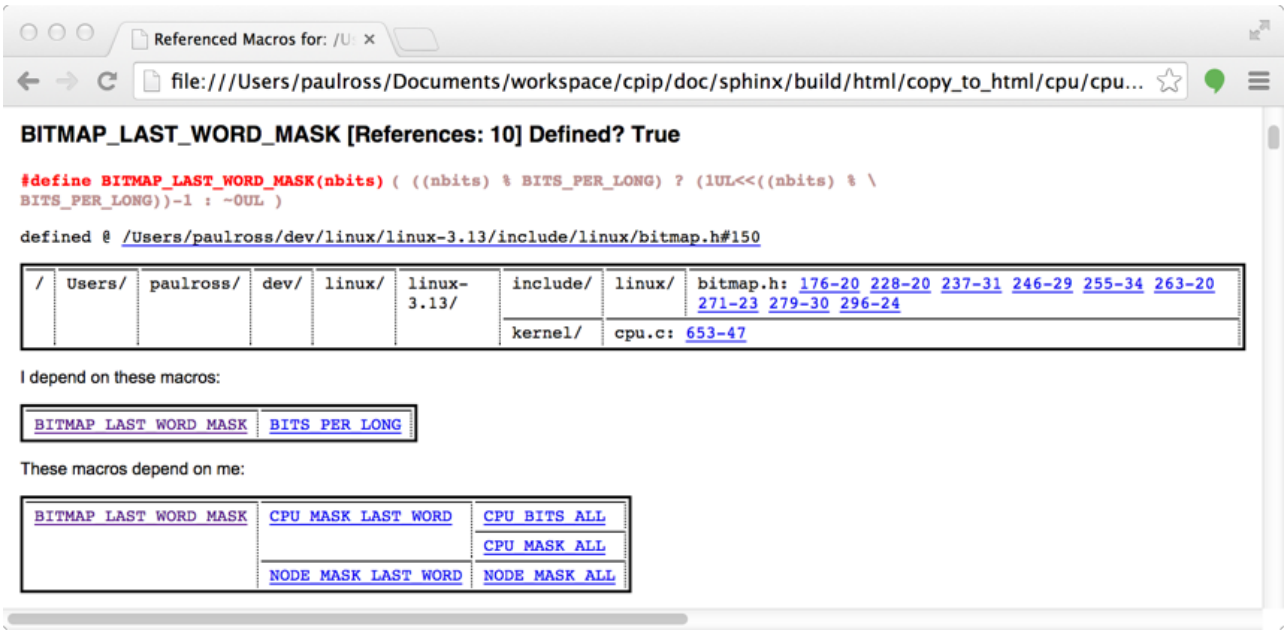
2.4 Understanding Macros

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



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.



2.5 Status

2.6 Licence

2.7 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 *<PYTHONVENVS>*, say *~/pyvenvs*:

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

3.1 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.

3.2 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
```

3.3 Developing with CPIP

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

3.3.1 Test Coverage

Install `pytest-cov`:

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

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

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

3.3.2 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*.

3.4 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.

CHAPTER 4

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 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)[1]. Other standards are considered, particularly C11, C++, C++11, C++14 and so on[2].

[1] 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).

[2] Relevant C standards [www.open-std.org] (draft, open access). Relevant C++ standards [www.open-std.org] (draft, open access).

4.1 Pre-processing C and C++

The basic task of any preprocessor is to produce a *Translation Unit* for a compiler to work with. The main job the pre-processor has to do involves three inter-related tasks[3]:

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

4.1.1 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.

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.

4.1.2 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.

4.1.3 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:

[3] Specifically the pre-processor has to do the first 6 (out of 8 or 9) *phases of translation* (for example see ISO/IEC 9899:1999 (E) ©ISO/IEC Sect. 5.1.1.2 Translation phases). Simplified they are:

1. Map physical source file multibyte characters to the source character set. Trigraph sequences are replaced by single-character representations.
2. Line continuation characters \ are deleted splicing physical source lines to form logical source lines.
3. The source file is decomposed into preprocessing tokens and white-space (including comments). Each comment is removed and replaced by one space character.
4. Preprocessing directives are executed and macro invocations are expanded. A `#include` preprocessing directive causes the named file to be processed from phase 1 to 4, recursively. All preprocessing directives are then deleted.
5. Each source character, escape sequence and string literal is converted to the execution character set.
6. Adjacent string literal tokens are concatenated.

```
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."[4]

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

4.2 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.

4.3 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:

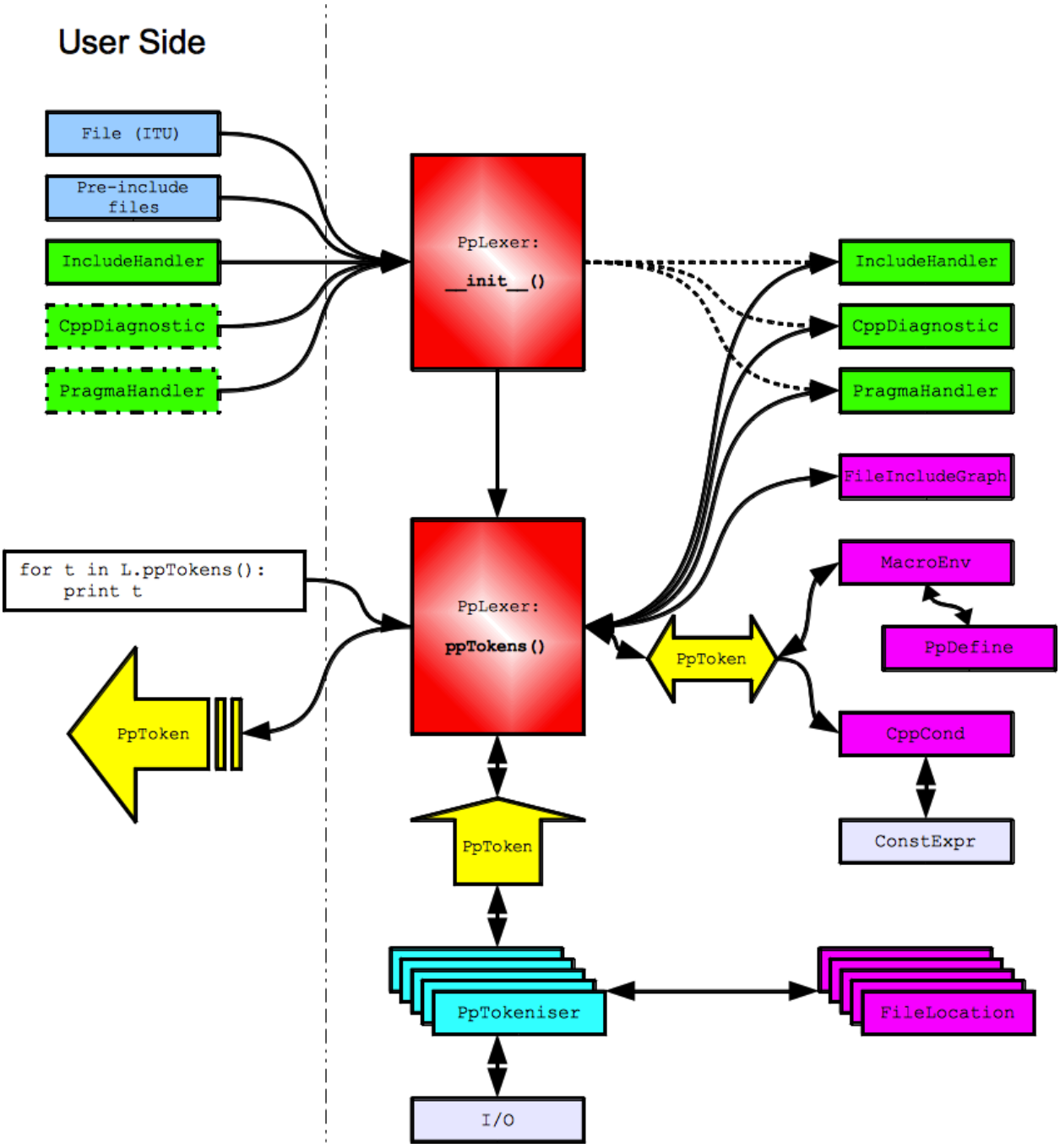
- File inclusion.

[4] Rationale for International Standard - Programming Languages - C Revision 5.10 April-2003 Sect. 6.10.3.4

- 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*.



CHAPTER 5

CPIPMain.py Examples

5.1 Screenshots

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

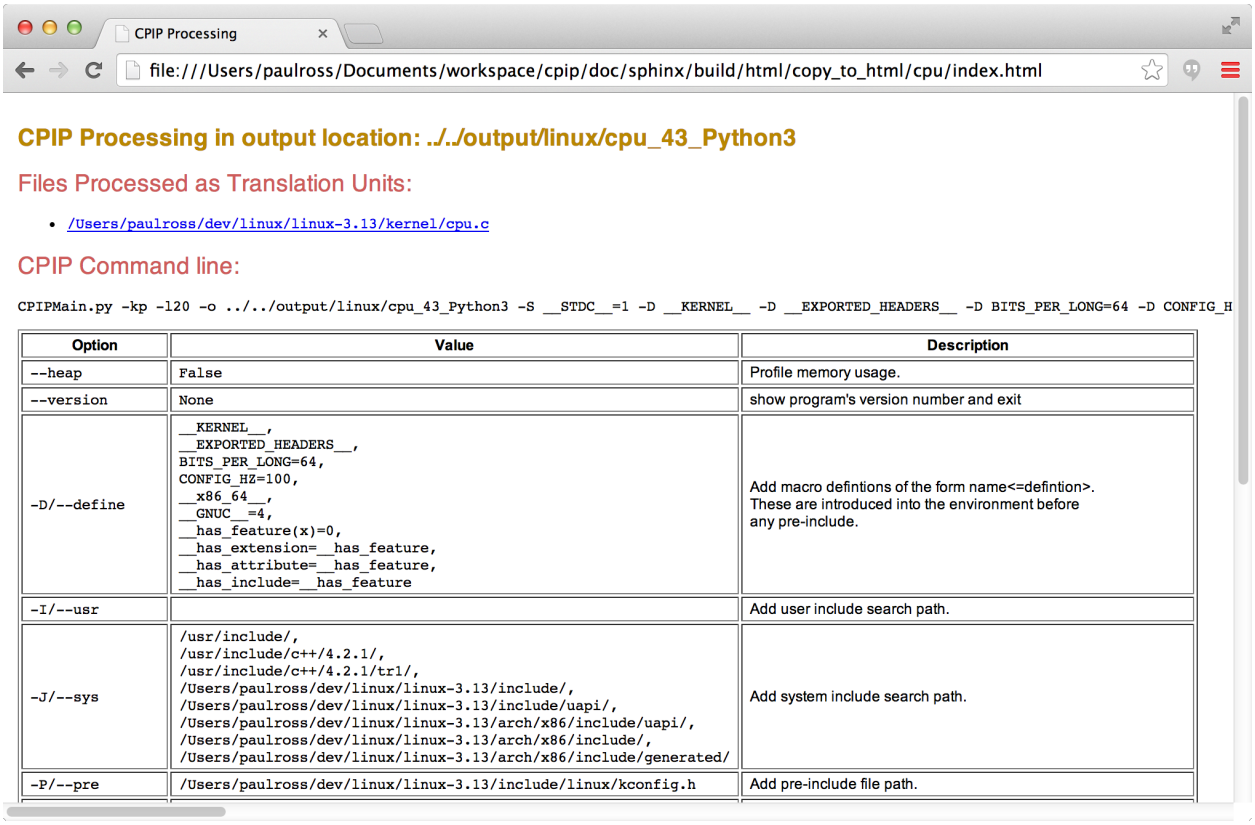
CPIPMain.py produces a set of HTML and SVG pages for each source file preprocessed. As well as the *Translation Unit* CPIPMain.py generates information about the three important tasks for a preprocessor: file inclusion, conditional compilation and macro replacement.

5.1.1 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:



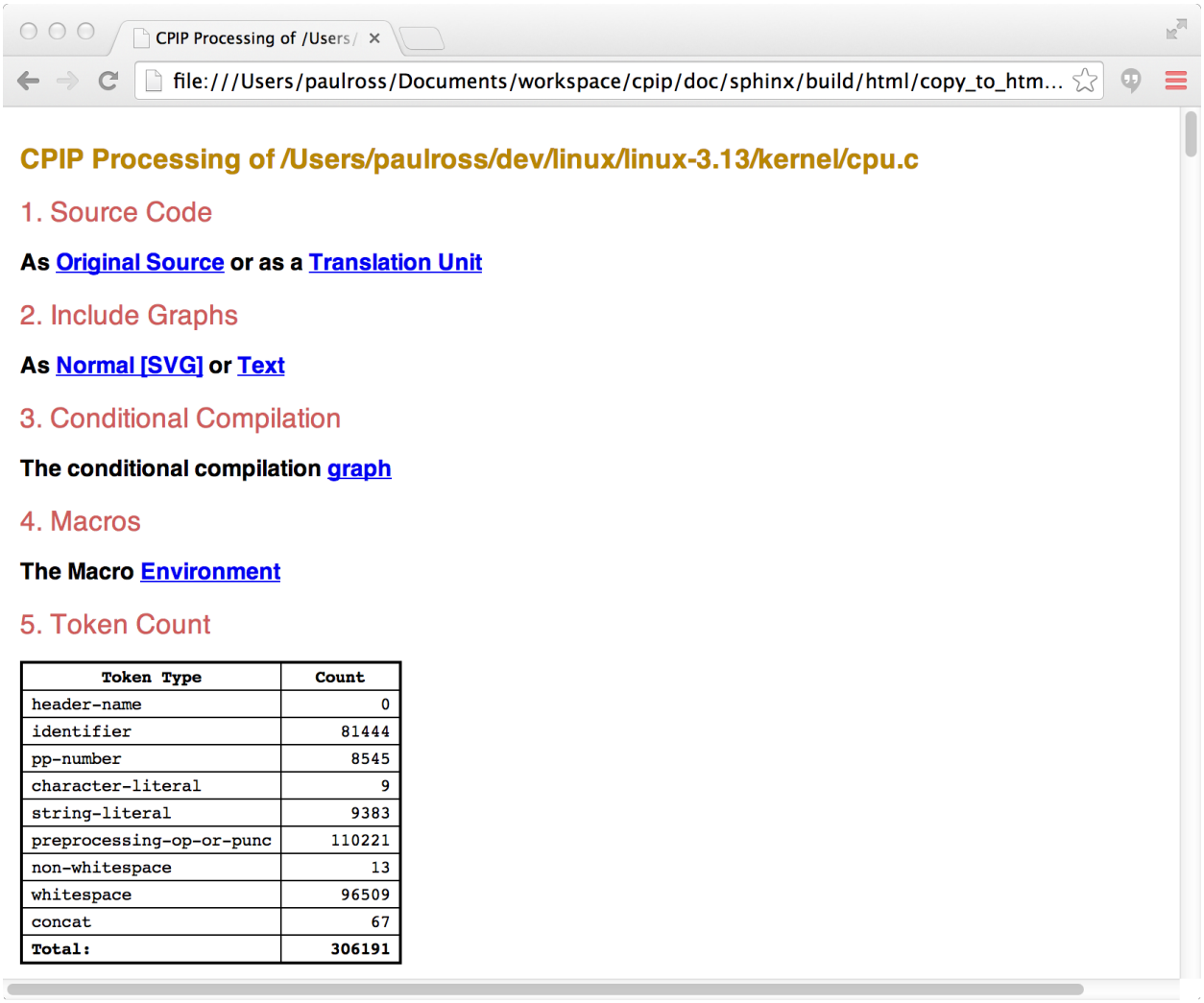
Each in the list of files preprocessed in this pass is linked to file specific page.

5.1.2 Preprocessed File Specific Page

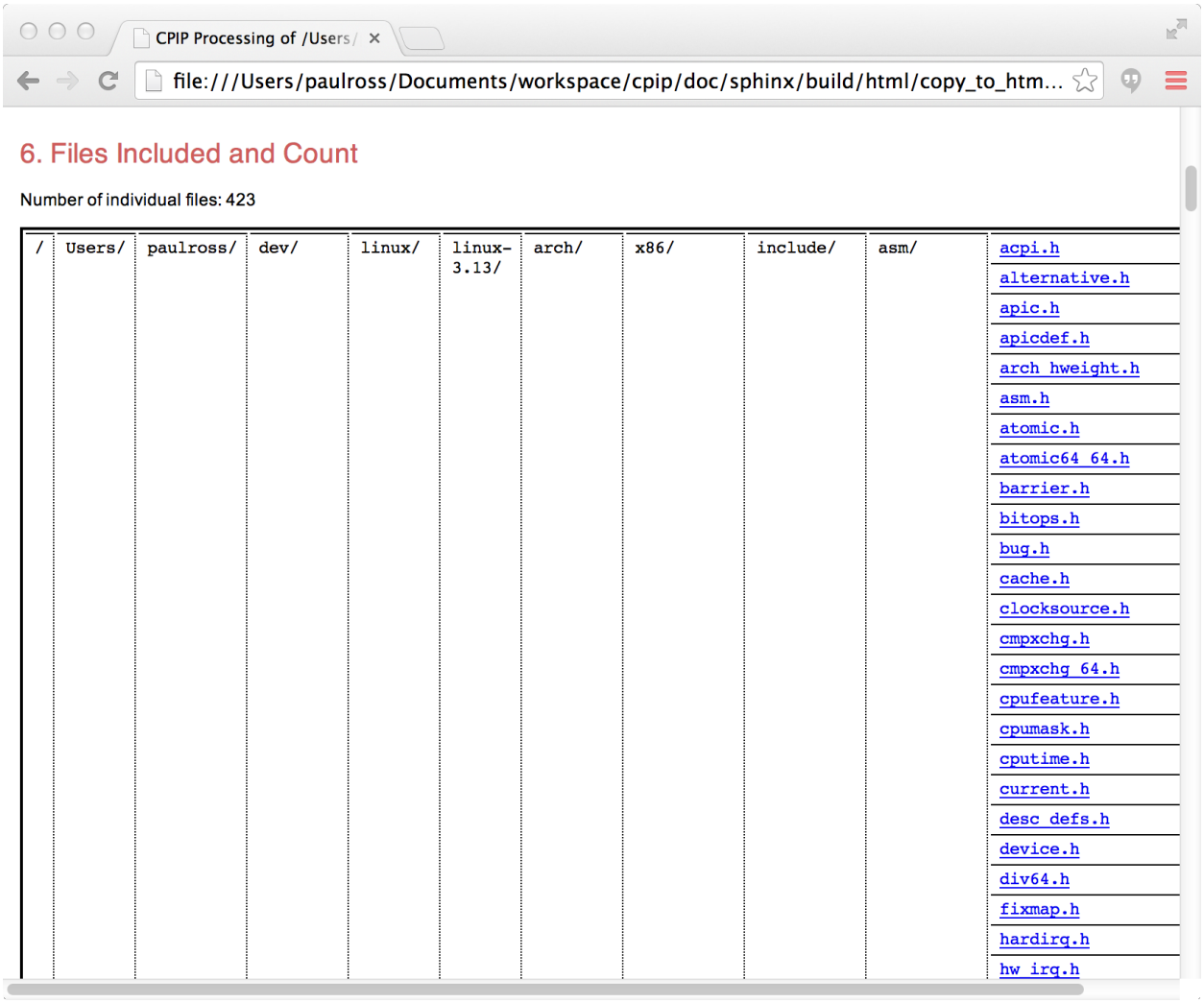
These describe the results of preprocessing a single file, it contains links to:

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.

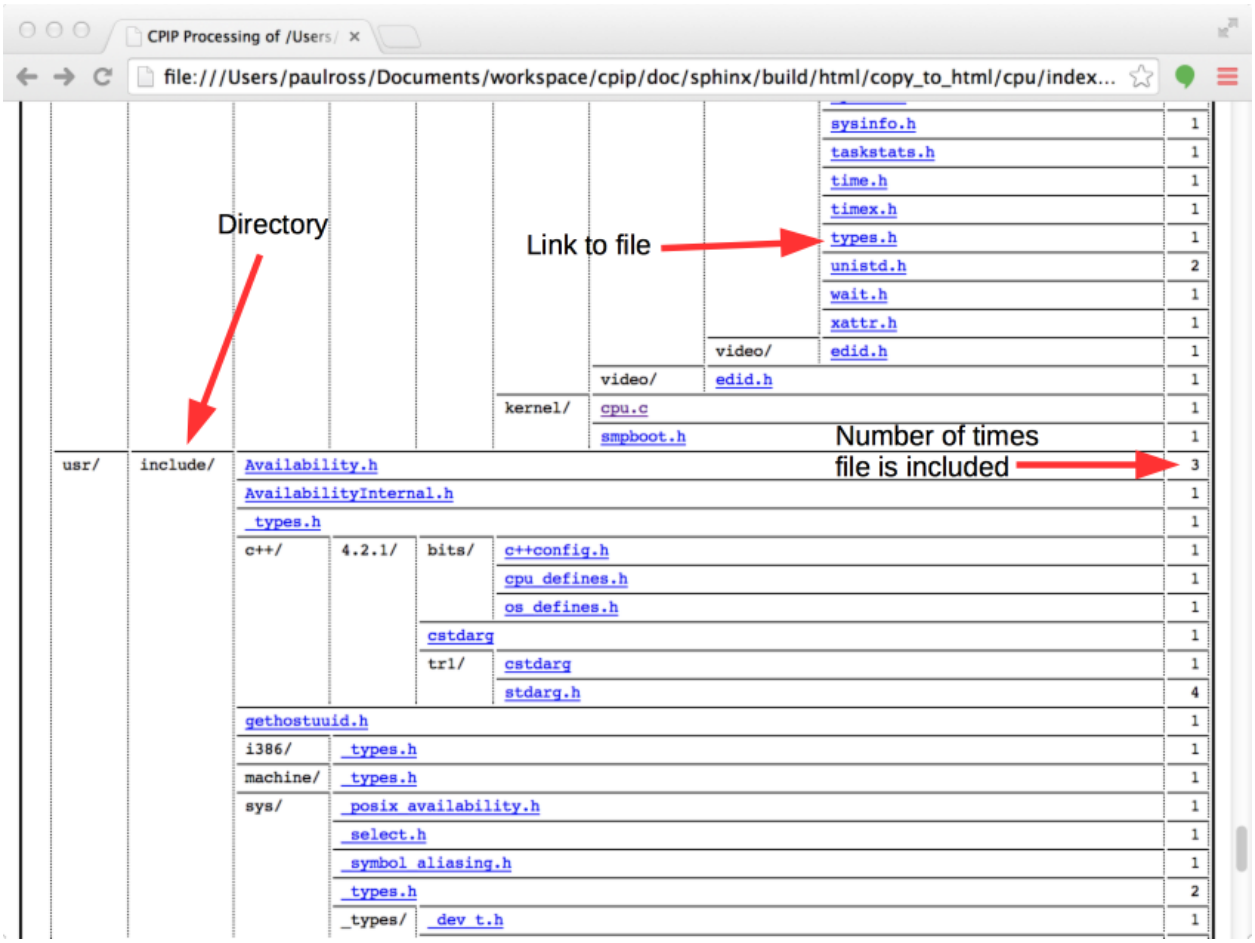
The top of the page includes links to these sections (described in detail below):



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



Here is an explanation for the table:

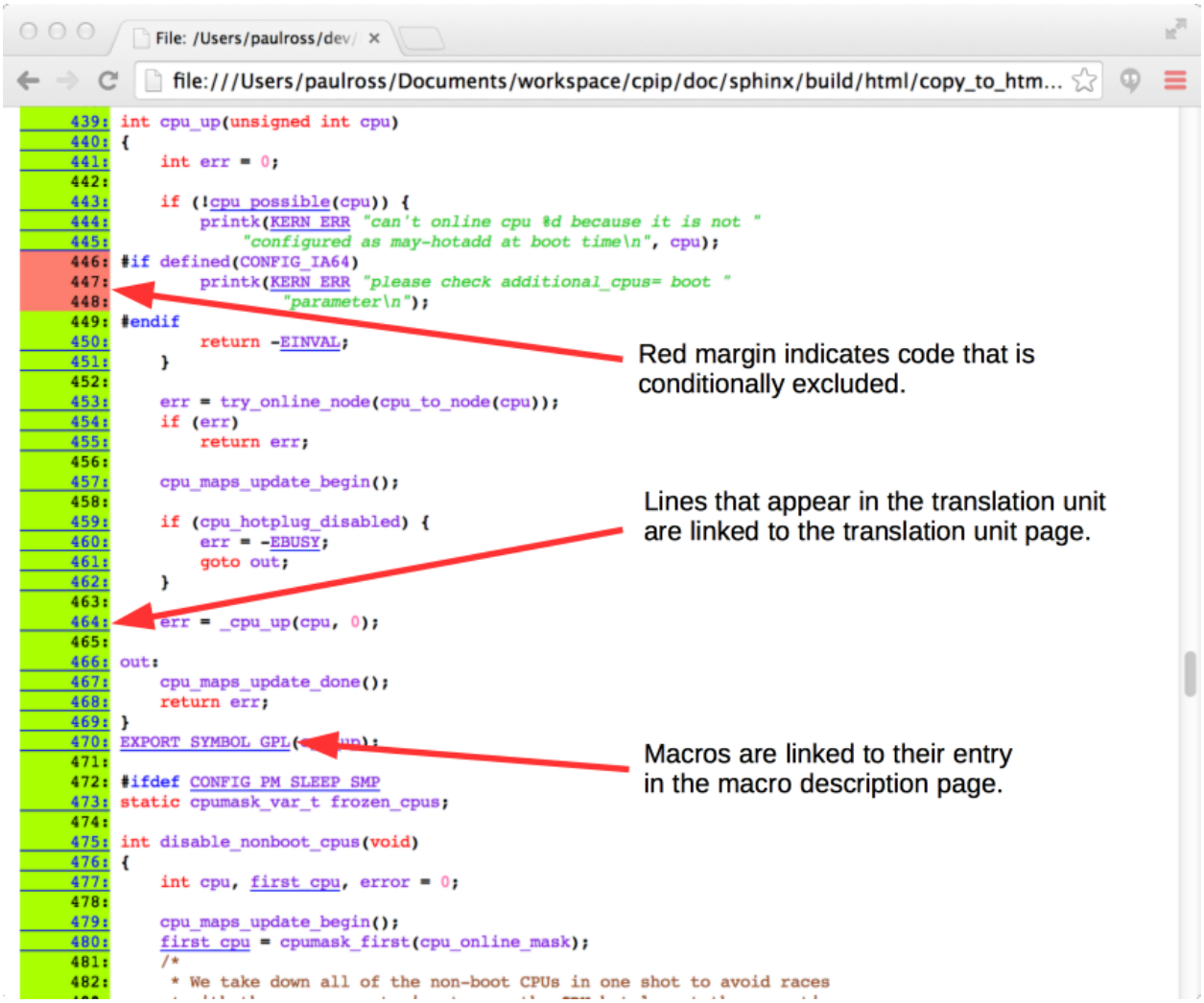


Original File and the Translation Unit

Original File

All processed source code (original file and 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.



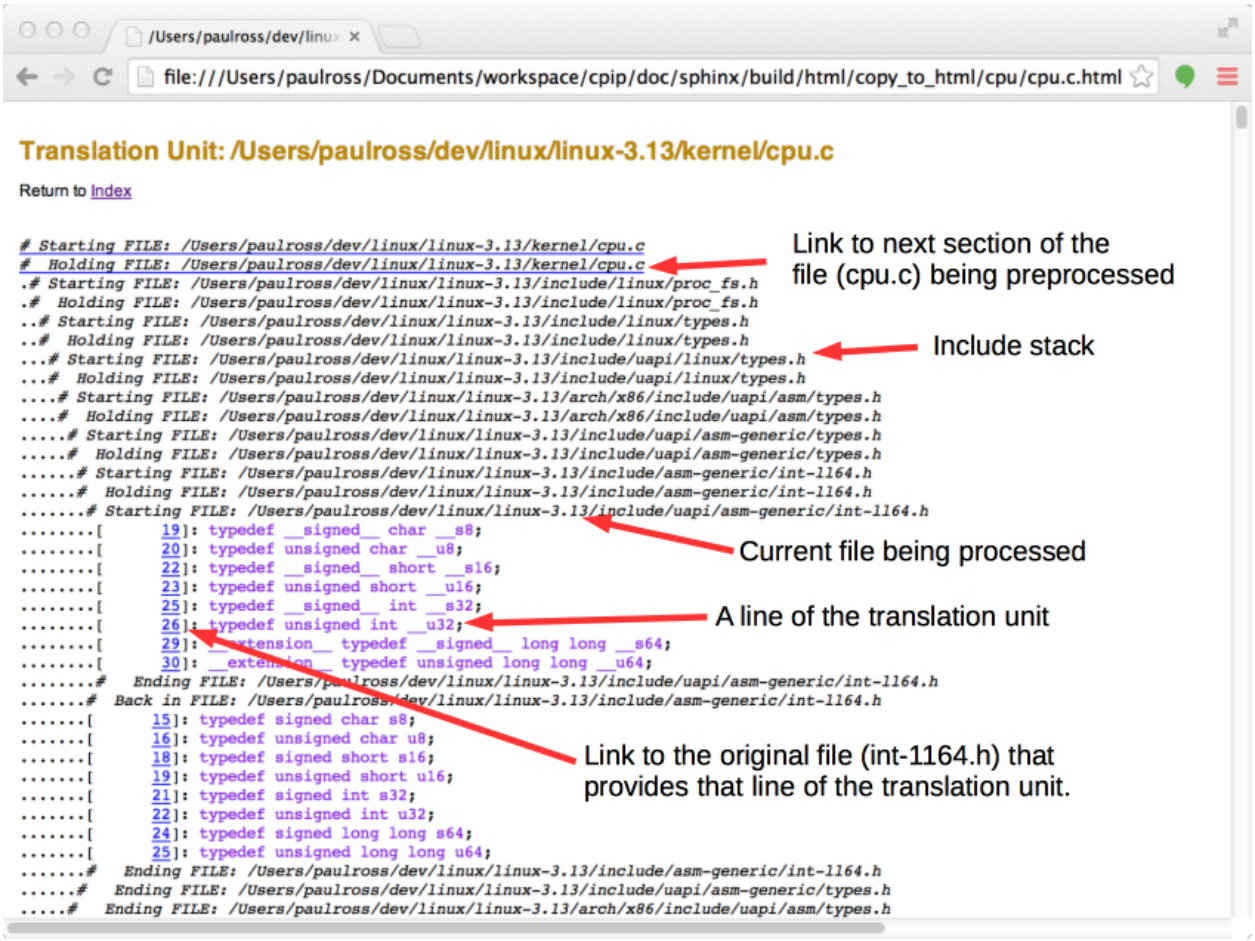
Translation Unit

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

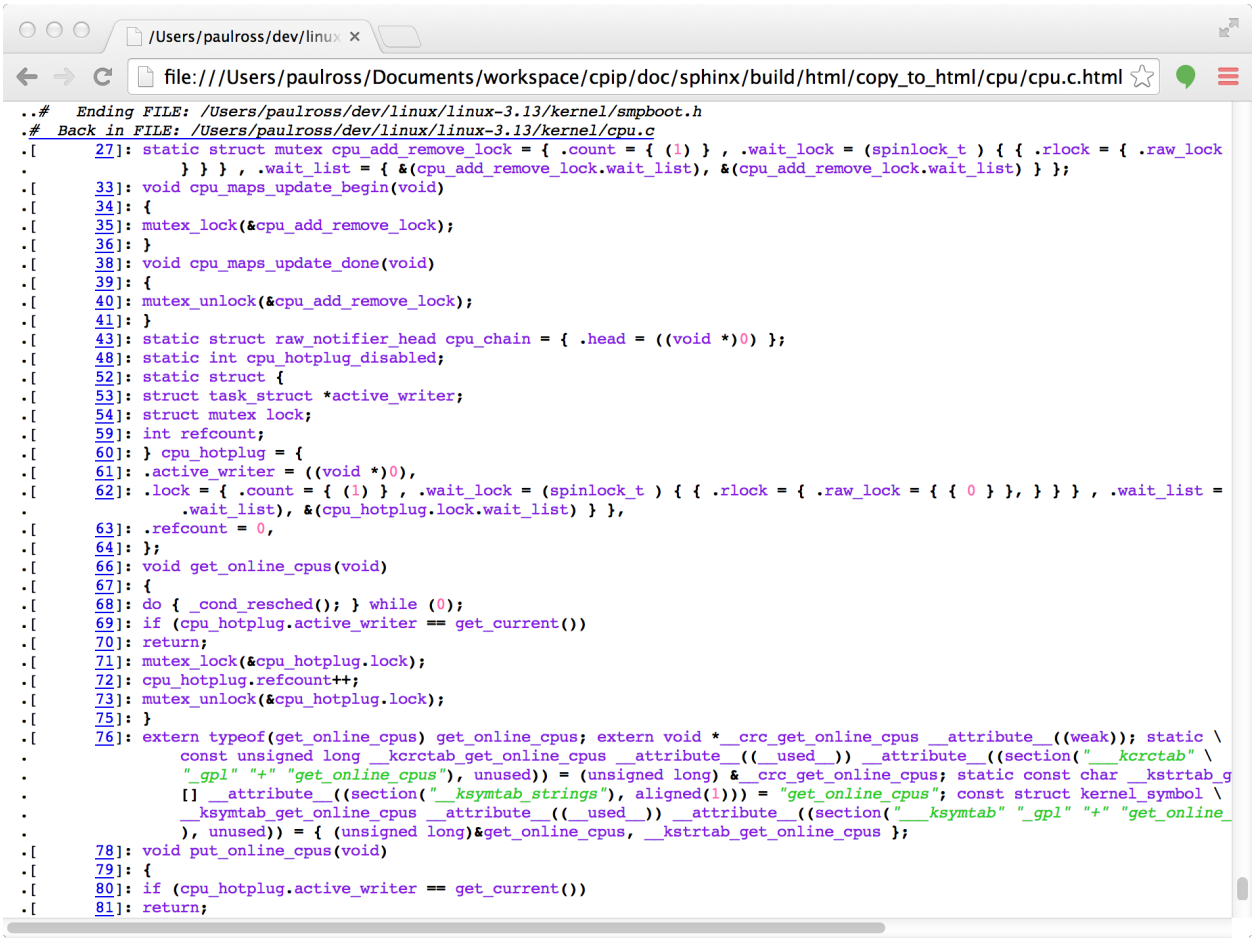
Each `#include` statement is represented in a nested fashion, any source code in the translation unit is presented syntax highlighted. The syntax is, of course, the C pre-processor language thus both `typedef` and `char` are pre-processor *identifiers* even if later on `typedef` is seen as a C keyword.

The numbered links thus `[ 19]` are to an HTML representation of the orignal 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 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.



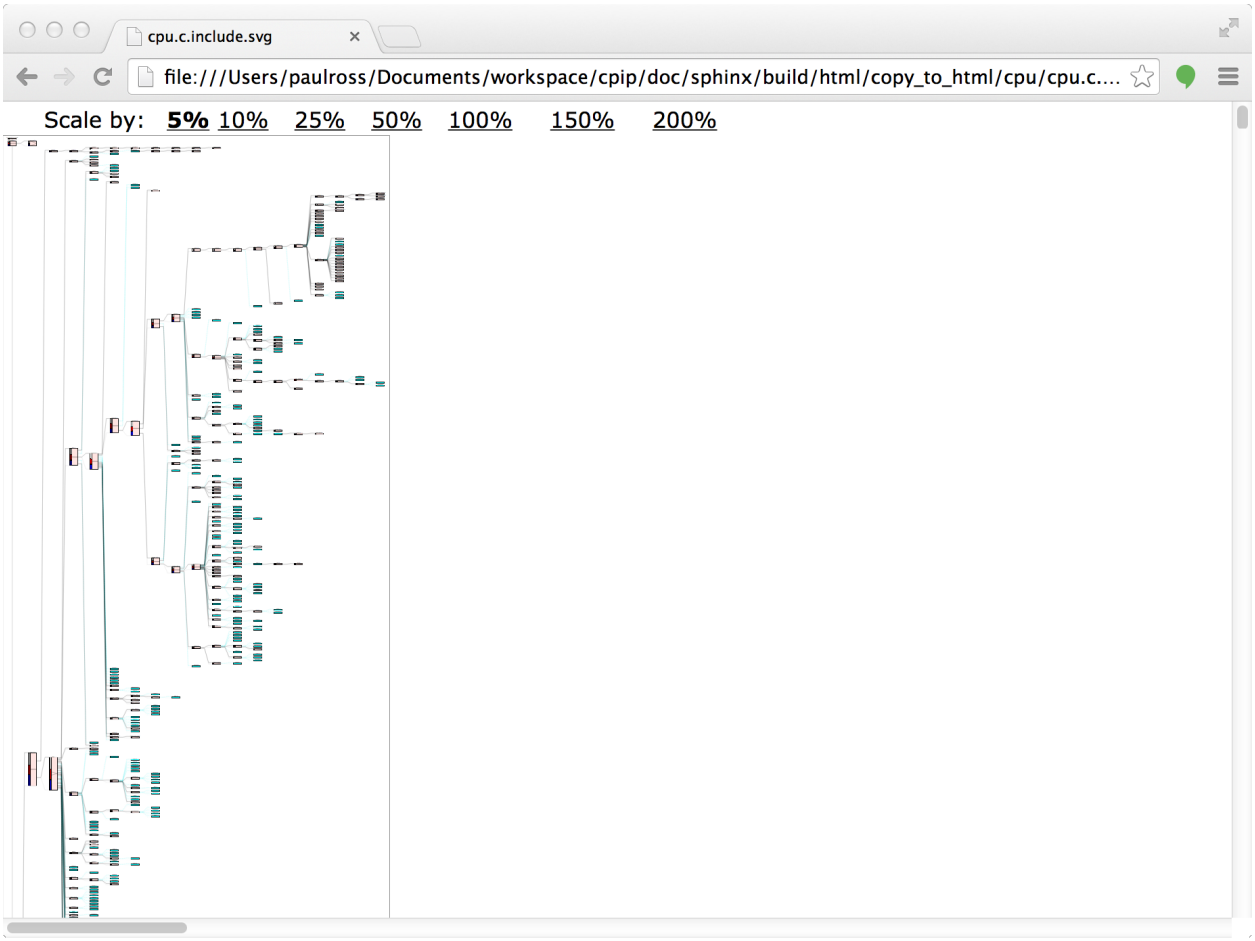
The SVG Include Graph

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

The Overall Picture

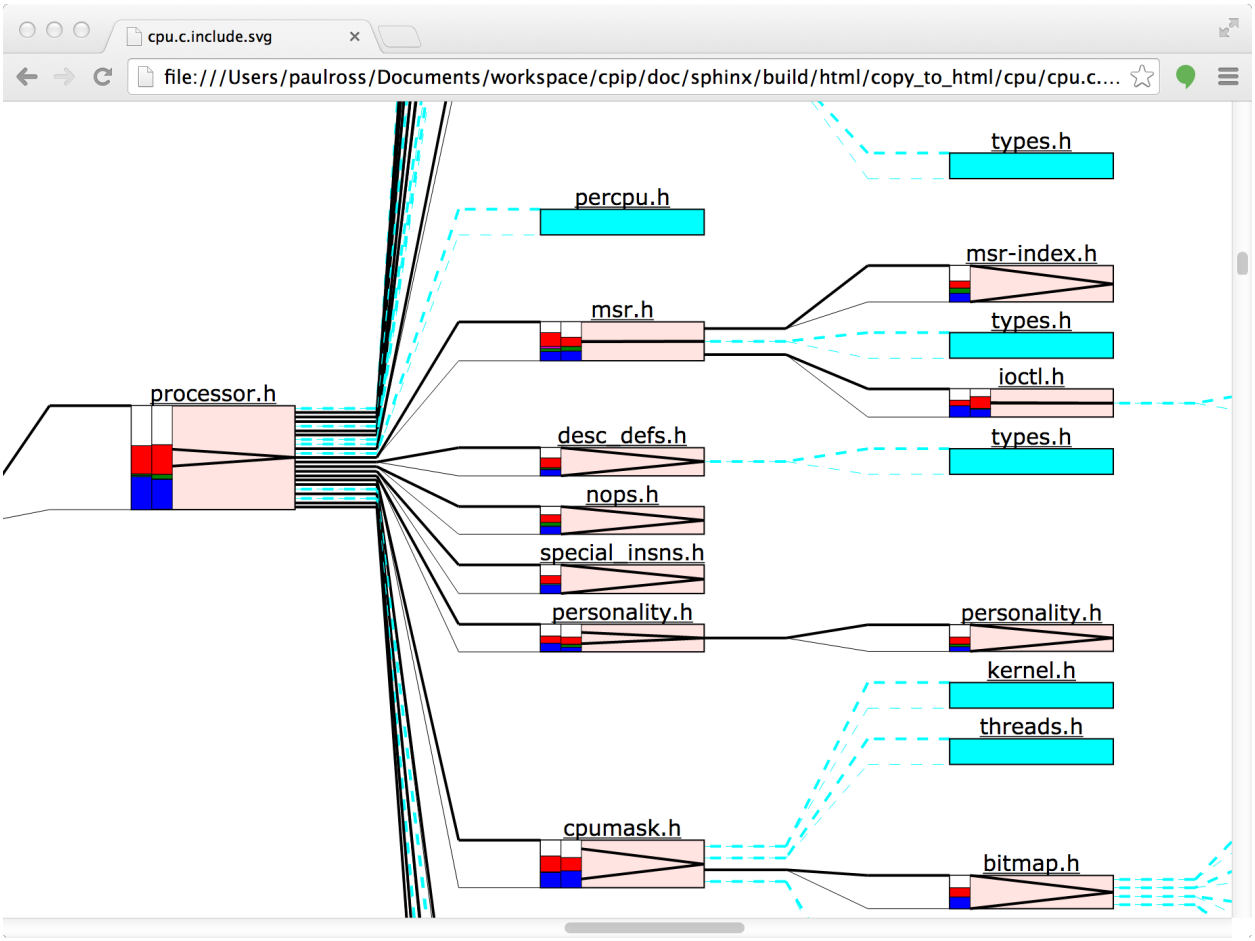
The diagram represents a tree with the root (the file being preprocessed) at center left. Each node represents a file and each edge represents an `#include` directive. Increasing include depth is left-to-right and source code order (i.e. order of the `#include` directives) is top to bottom.

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)[1]. 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).

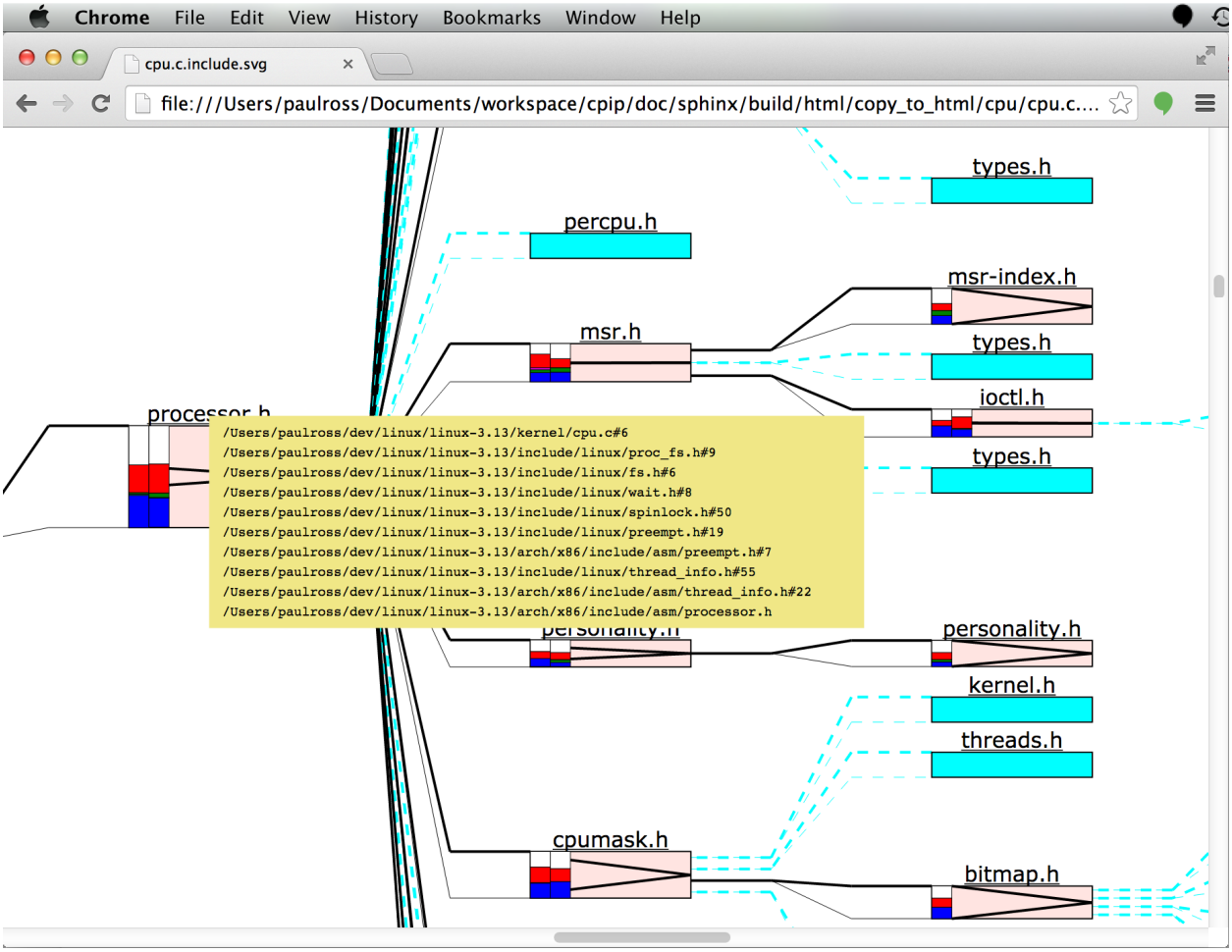
The coloured bars on the left represent the count of different token types, the left bar being the current file, the right bar being the total of the descendants. See below for which *Token Types* corespond to each colour.

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

File Path

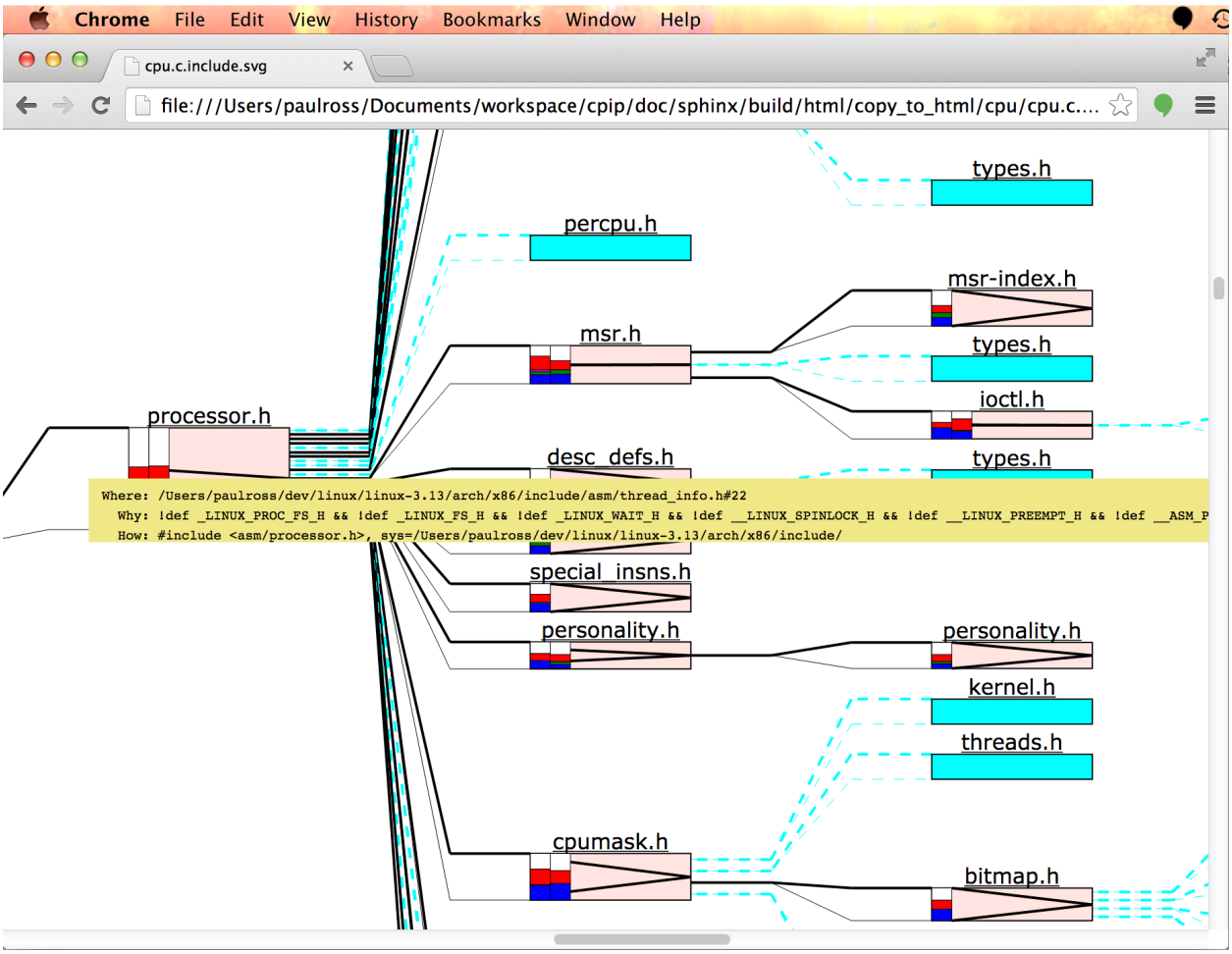
For example mousing over the file name above the box shows the the absolute path of the file stack as a pop-up yellow block. At the top of this list is the file we are preprocessing, then the stack of included files downwards to `processor.h`:

[1] A special case is that the 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:

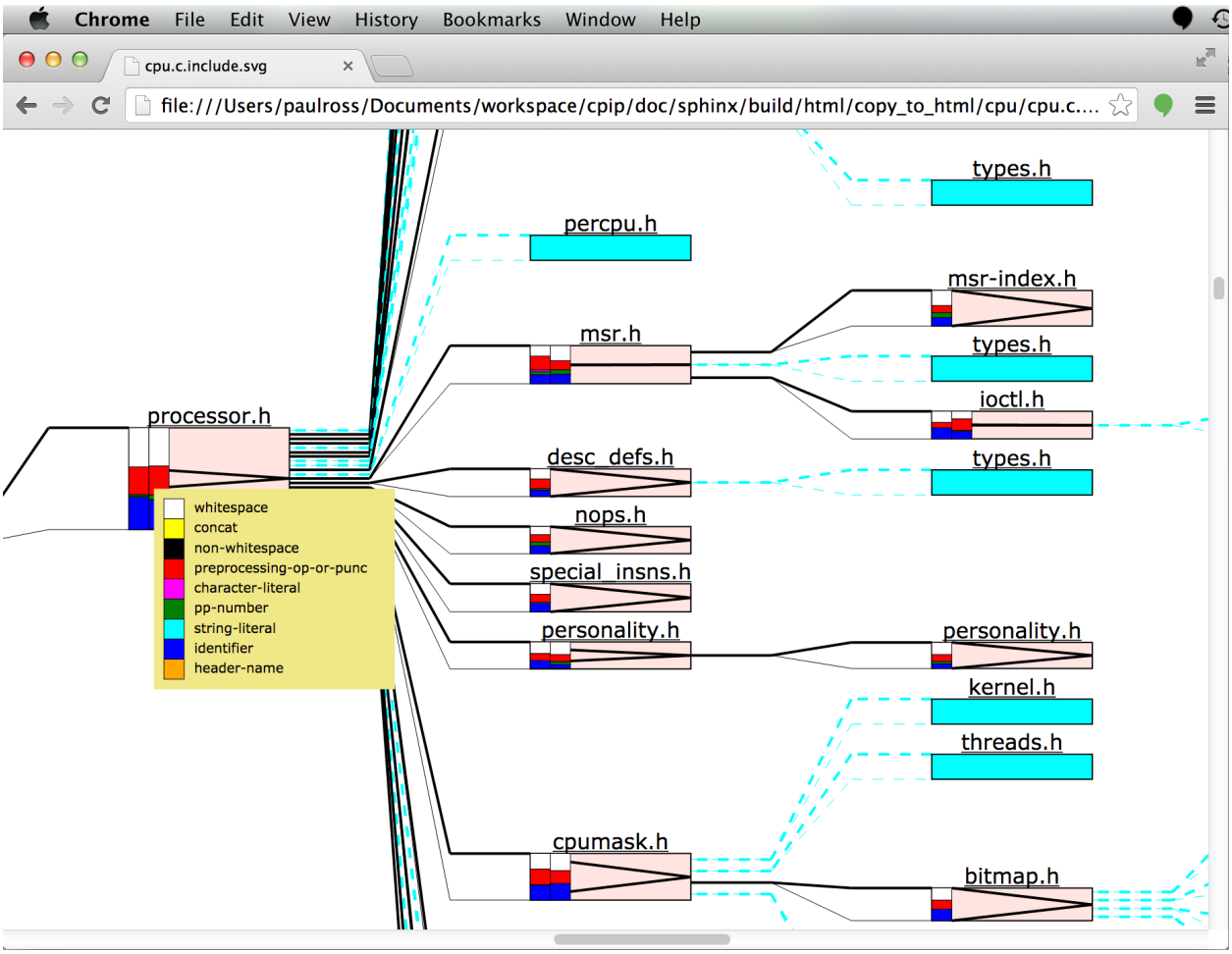


This pop-up yellow block contains the following:

- `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'.

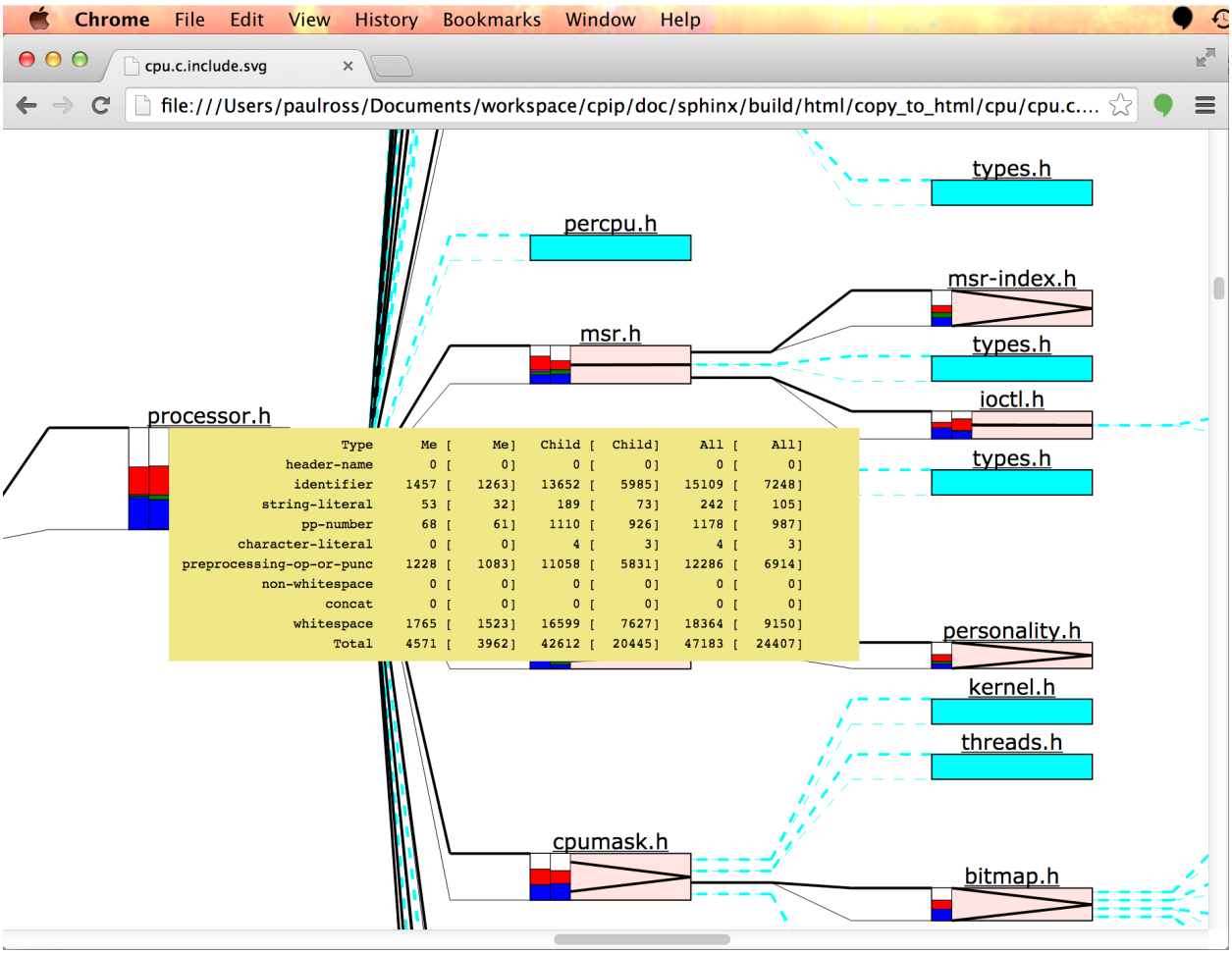
Token Types

If you are interested in what types of preprocessor tokens were encountered than 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.

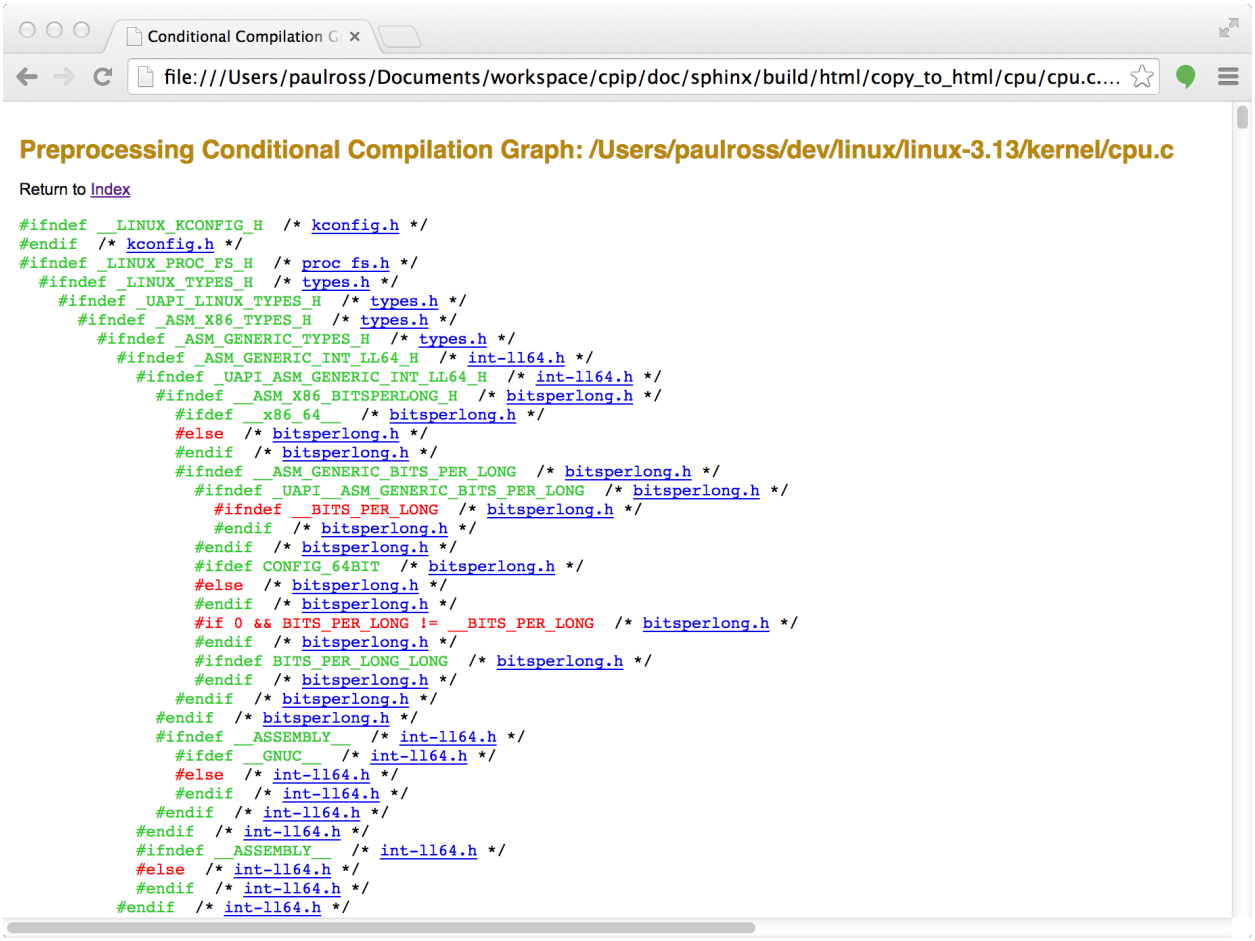


Conditional Compilation

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.

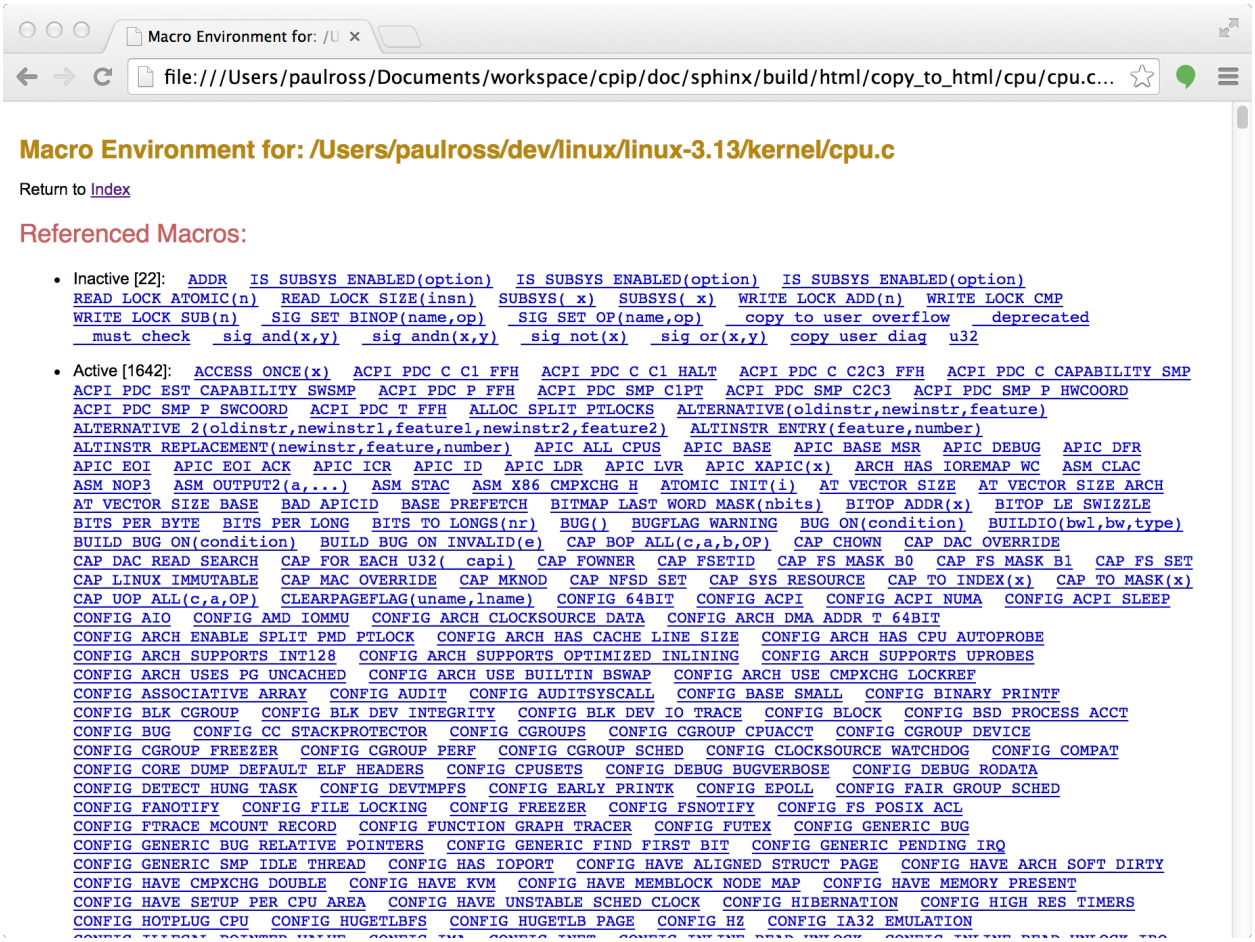
As an aid CPIP produces an HTML page that is the translation unit with *only* the conditional compilation statements, what is more they are nested according to their logical execution condition and colour coded according to the resolved state; green means code will be part of the translation unit and 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:



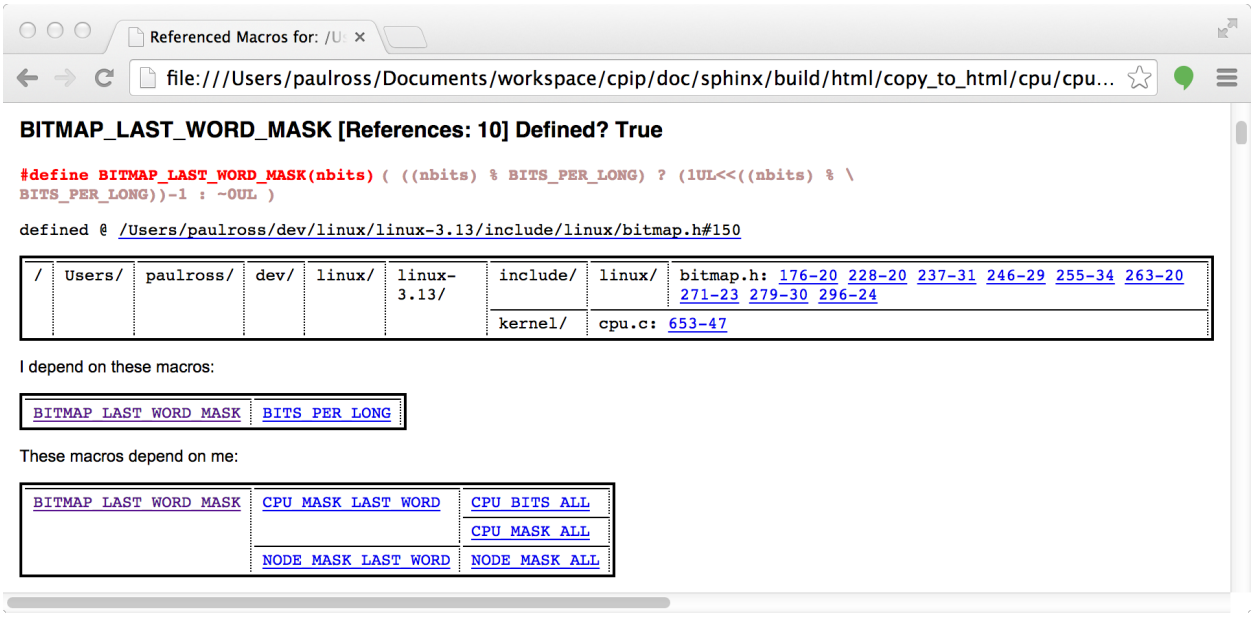
Macro Definitions

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



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`:



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 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.

5.2 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.

5.2.1 From the Linux Kernel

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

5.2.2 From the CPython Interpreter

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

CHAPTER 6

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`.

6.1 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*.

6.1.1 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
`--version`	Show program's version number and exit
`-h, --help`	Show this help message and exit.
`-c`	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]
`-d DUMP, --dump=DUMP`	Dump various outputs to stdout (see below). This option can be repeated [default: []]
`-g GLOB, --glob=GLOB`	Pattern to use when searching directories (ignored for `#includes`). [default: `*.*`]
`--heap`	Profile memory usage (requires `guppy` to be installed). [default: False]
`-j JOBS, --jobs=JOBS`	Max processes when multiprocessing. Zero uses number of native CPUs [4]. Value of 1 disables multiprocessing. [default: 0]
`-k`	Keep going as far as sensible, for some definition of "sensible". [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`	Recursively provesses directories. [default: False]
`-t, --dot`	Write an DOT include dependency file and execute DOT on it to create a SVG file. Requires GraphViz. [default: False]
`-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]
`-G`	Support GCC extensions. Currently only `#include_next`. [default: False]
`-I INCUSR, --usr=INCUSR`	Add user include search path (additive). This option can be repeated [default: []]
`-J INCSYS, --sys=INCSYS`	Add system include search path (additive). This option can be repeated [default: []]
`-S PREDEFINES, --predefine=PREDEFINES`	Add standard predefined macro defintions of the form `name<=defintion>`. These are introduced into the environment before anything else. These macros can not be redefined. `__DATE__` and `__TIME__` will be automatically defined. This option can be repeated [default: []]
`-D DEFINES, --define=DEFINES`	Add macro definitions of the form `name<=definition>`. These are introduced into the environment before any pre-include. This option can be repeated [default: []]
`-P PREINC, --pre=PREINC`	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
`-d C`	Conditional compilation graph.
`-d F`	File names encountered and their count.
`-d I`	Include graph.
`-d M`	Macro environment.
`-d T`	Token count.
`-d R`	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.

6.1.2 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/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 ----------------------------
```

6.1.3 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 agressive 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%.

CHAPTER 7

CPIP Tutorials

Various Tutorials on how to use CPIP.

Contents:

7.1 PpLexer Tutorial

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

7.1.1 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 and `PpLexer.PpLexer` object, and this takes at least two arguments; firstly the file to pre-process, the secondly an *include handler*. The latter is need because the C/C++ standards do not specify how an `#include` directive is to be processed as that is as an implementation issue. So we need to provide an 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`.

7.1.2 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 definately 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.

7.1.3 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, '')
```

7.1.4 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, '='))
```

```
    print(myLex.condCompGraph)
    print(' Conditional Compilation Graph END '.center(75, '='))

    print()
    print(' Macro Environment '.center(75, '='))
    print(myLex.macroEnvironment)
    print(' Macro Environment END '.center(75, '='))

    print()
    print(' Macro History '.center(75, '='))
    print(myLex.macroEnvironment.macroHistory(incEnv=False, onlyRef=False))
    print(' Macro History END '.center(75, '='))

if __name__ == "__main__":
    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*

7.1.5 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.

7.1.6 Advanced PpLexer Construction

The `PpLexer` constructor allows you to change the behaviour of pre-processing is 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.

7.2 FileIncludeGraph Tutorial

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

7.2.1 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:

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

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

if __name__ == "__main__":
    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>
```

7.2.2 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*

7.2.3 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.

7.2.4 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 thqat 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 suplies 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`

CHAPTER 8

CPIP Reference

8.1 Command Line Tools

8.1.1 CPIPMain - The main command line tool for preprocessing.

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

```
(CPIP36) $ python src/cpip/CPIPMain.py --help
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.
  Version: v0.9.8rc0
  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: []] i.e. every file.
  --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: []]
```

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

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

_fileName(*theFigNode*)

Treat the file name consistently.

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

.

class cpip.CPIPMain.**FigVisitorLargestCommanPrefix**

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*)

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *incHandler*, *preDefMacros*, *preIncFiles*, *diagnostic*, *pragmaHandler*, *keepGoing*, *conditionalLevel*, *dumpList*, *helpMap*, *includeDOT*, *cmdLine*, *gccExtensions*)
Create new instance of MainJobSpec(incHandler, preDefMacros, preIncFiles, diagnostic, pragmaHandler, keepGoing, conditionalLevel, dumpList, helpMap, includeDOT, cmdLine, gccExtensions)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0xa385c0>*, *len=<built-in function len>*)
Make a new MainJobSpec object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new MainJobSpec object replacing specified fields with new values

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*)

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__ (*_cls*, *ituPath*, *indexPath*, *tuIndexFileName*, *total_files*, *total_lines*, *total_bytes*)
Create new instance of PpProcessResult(ituPath, indexPath, tuIndexFileName, total_files, total_lines, total_bytes)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make (*iterable*, *new=<built-in method __new__ of type object at 0xa385c0>*, *len=<built-in function len>*)
Make a new PpProcessResult object from a sequence or iterable

_replace (*_self*, ***kwds*)
Return a new PpProcessResult object replacing specified fields with new values

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.**_removeCommonPrefixFromResults** (*titlePathTupleS*)
Given a list of: `PpProcessResult(ituPath, indexPath, tuIndexFileName(ituPath), total_files, total_lines, total_bytes)` This prunes the commmon prefix from the ituPath.

cpip.CPIPMain.**_tdCallback** (*theS*, *attrs*, *_k*, *href_nav_text_file_data*)
Callback function for the file count table.

Parameters

- **theS** (`cpip.util.XmlWrite.XhtmlStream`) – HTML stream.
- **attrs** (`dict({str : [str]})`) – Attributes.
- **_k** (`list([str])`) – <insert documentation for argument>
- **href_nav_text_file_data** (`tuple([str, str, tuple([int, int, int])])`) – <insert documentation for argument>

Returns `NoneType`

cpip.CPIPMain.**_trThCallback** (*theS*, *theDepth*)
Create the table header:

```
<tr>
  <th class="filetable" colspan="9">File Path </th>
  <th class="filetable">Include Count</th>
  <th class="filetable">Lines</th>
  <th class="filetable">Bytes</th>
  <th class="filetable">Total Lines</th>
```

```
  <th class="filetable">Total Bytes</th>
</tr>
```

Parameters

- **theS** (cpip.util.XmlWrite.XhtmlStream) – HTML stream.
- **theDepth** (int) – <insert documentation for argument>

Returns NoneType

cpip.CPIPMain.**_writeCommandLineInvocationToHTML**(*theS*, *theJobSpec*)

Writes the command line to the index page.

Parameters

- **theS** (cpip.util.XmlWrite.XhtmlStream) – HTML stream to write to.
- **theJobSpec** (__main__.MainJobSpec([cpip.core.IncludeHandler.CppIncludeStdOs, dict({str : [str]}), list([_io.StringIO]), NoneType, <class 'NoneType'>, bool, int, list([]), dict({str : [tuple([<class 'bool'>, str]), tuple([<class 'list'>, str]), tuple([int, str]), tuple([list([]), str]), tuple([list([str]), str]), tuple([str, str])]}), <class 'bool'>, str, <class 'bool'>])) – Job specification.

Returns NoneType

cpip.CPIPMain.**_writeDirectoryIndexHTML**(*theInDir*, *theOutDir*, *titlePathTupleS*, *theJobSpec*, *time_start*)

Writes a super index.html when a directory has been processed. titlePathTuples is a list of: PpProcessResult(ituPath, indexPath, tuIndexFileName, total_files, total_lines, total_bytes)

cpip.CPIPMain.**_writeIndexHtmlTrailer**(*theS*, *time_start*)

Write a trailer to the index.html page with the start/finish time and version. If time_start is None then only the current time is written.

Parameters

- **theS** (cpip.util.XmlWrite.XhtmlStream) – HTML stream.
- **time_start** (NoneType, float) – Start time.

Returns NoneType

cpip.CPIPMain.**_writeParagraphWithBreaks**(*theS*, *theParas*)

Writes the paragraphs with page breaks.

Parameters

- **theS** (cpip.util.XmlWrite.XhtmlStream) – HTML stream.
- **theParas** (list([str])) – Paragraphs.

Returns NoneType

cpip.CPIPMain.**includeGraphFileNameCcg**(*theItu*)

Returns the path to the CCG output file.

Parameters **theItu** (str) – Path to the ITU.

Returns str – path.

cpip.CPIPMain.**includeGraphFileNameDotSVG**(*theItu*)
Returns the path to the SVG DOT output file.

Parameters **theItu** (str) – Path to the ITU.

Returns str – path.

cpip.CPIPMain.**includeGraphFileNameDotTxt**(*theItu*)
Returns the path to the DOT output file.

Parameters **theItu** (str) – Path to the ITU.

Returns str – path.

cpip.CPIPMain.**includeGraphFileNameSVG**(*theItu*)
Returns the path to the SVG output file.

Parameters **theItu** (str) – Path to the ITU.

Returns str – path.

cpip.CPIPMain.**includeGraphFileNameText**(*theItu*)
Returns the path to the include graph output file.

Parameters **theItu** (str) – Path to the ITU.

Returns str – path.

cpip.CPIPMain.**main**()
Processes command line to preprocess a file or a directory.

Returns int – status, 0 is success.

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 KeyboardInterupt) 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)

Parameters

- **ituPath** (str) – Path to the initial translation unit (ITU).
- **outDir** (str) – Output directory.
- **jobSpec** (__main__.MainJobSpec([cpip.core.IncludeHandler.CppIncludeStdOs, dict({}), list([_io.StringIO]), NoneType, <class 'NoneType'>, bool, int, list([]), dict({str : [tuple([<class 'bool'>, str]), tuple([<class 'list'>, str]), tuple([int, str]), tuple([list([]), str]), tuple([list([str]), str]), tuple([str, str])]}), <class 'bool'>, str, <class 'bool'>])) – Job specification.

Returns __main__.PpProcessResult([str, str, str, int, int, int]) – Details of the result.

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 (re)reading the file.

Parameters **theLexer** (*cpip.core.PpLexer.PpLexer*) – The Lexer

Returns `dict({str : tuple([int, int, int])})` – The file count map.

cpip.CPIPMain.**retOptionMap**(*theOptParser*, *theOpts*)
Returns map of {opt_name : (value, help), ...} from the current options.

Parameters

- **theOptParser** (`argparse.ArgumentParser`) – The option parser.
- **theOpts** (`argparse.Namespace`) – Parsed options.

Returns `dict({str : [tuple([<class 'bool'>, str]), tuple([bool, str]), tuple([int, str]), tuple([list([]), str]), tuple([list([str]), str]), tuple([str, str])]})` – Option name to value and help text.

Raises `KeyError`

cpip.CPIPMain.**tuFileName**(*theTu*)
Returns the path to the translation unit output file.

Parameters **theItu** (`str`) – Path to the ITU.

Returns `str` – path.

cpip.CPIPMain.**tuIndexFileName**(*theTu*)
Returns the path to the index output file.

Parameters **theItu** (`str`) – Path to the ITU.

Returns `str` – path.

cpip.CPIPMain.**writeIncludeGraphAsText**(*theOutDir*, *theItu*, *theLexer*)
Writes out the include graph as plain text.

Parameters

- **theOutDir** (`str`) – Output directory.
- **theItu** (`str`) – Path to ITU.
- **theLexer** (*cpip.core.PpLexer.PpLexer*) – The lexer.

Returns `NoneType`

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.

Parameters

- **theItuS** (`list([str])`) – The list of translation units processed.
- **theOutDir** (`str`) – The output directory.
- **theJobSpec** (`tuple([cpip.core.IncludeHandler.CppIncludeStdOs, dict({str : [str]}), list([_io.StringIO]), NoneType, <class 'NoneType'>, bool, int, list([]), dict({str :`

[tuple([<class 'bool'>, str]), tuple([<class 'list'>, str]), tuple([int, str]), tuple([list([]), str]), tuple([list([str]), str]), tuple([str, str])]}), <class 'bool'>, str, <class 'bool'>])) – The job specification.

- **time_start** (float) – Time that the process started.
- **total_files** (int) – Total number of files processed.
- **total_lines** (int) – Total number of lines processed.
- **total_bytes** (int) – Total number of bytes processed.

Returns str – The path to the index.html file that has been written.

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

Write the index.html for a single TU.

Parameters

- **theOutDir** (str) – The output directory to write to.
- **theTuPath** (str) – The path to the original ITU.
- **theLexer** (*cpip.core.PpLexer.PpLexer*) – The pre-processing Lexer that has pre-processed the ITU/TU.
- **theFileCountMap** (dict({str : [tuple([<class 'int'>, int, int]), tuple([int, int, <class 'int'>])]})) – dict of {file_path : data, ...} where data is things like inclusion count, lines, bytes and so on.
- **theTokenCntr** (*cpip.core.PpTokenCount.PpTokenCount*) – *cpip.core.PpTokenCount.PpTokenCount* containing the token counts.
- **hasIncDot** (bool) – bool to emit graphviz .dot files.
- **macroHistoryIndexName** (str) – String of the filename of the macro history.

Returns tuple([int, int, int]) – (total_files, total_lines, total_bytes) as integers.

Raises StopIteration

8.1.2 DupeRelink - Post processing the output of CPIPMain.py to remove duplicate HTML files.

DupeRelink.py – Searches for HTML files that are the same, writes a single file into a common area and deletes all the others. Then re-links all the remaining HTML files that linked to the original files to link to the file in the common area. This is a space saving optimisation after CPIPMain.py has processed a directory of source files.

```
(CPIP36) $ python src/cpip/DupeRelink.py --help
usage: DupeRelink.py [-h] [-s SUBDIR] [-n] [-v] [-l LOGLEVEL] path

DupeRelink.py - Delete duplicate HTML files and relink them to save space. WARNING:␣
↪This deletes in-place.
  Created by Paul Ross on 2017-09-26.
  Copyright 2017. All rights reserved.
  Licensed under GPL 2.0
USAGE

positional arguments:
  path                  Path to source directory. WARNING: This will be
```

```
                        rewritten in-place.

optional arguments:
  -h, --help            show this help message and exit
  -s SUBDIR, --subdir SUBDIR
                        Sub-directory for writing the common files. [default:
                        _common_html]
  -n, --nervous         Nervous mode, don't do anything but report what would
                        be done. Use -l20 to see detailed result. [default:
                        False]
  -v, --verbose         Verbose, lists duplicate files and sizes. [default:
                        False]
  -l LOGLEVEL, --loglevel LOGLEVEL
                        Log Level (debug=10, info=20, warning=30, error=40,
                        critical=50) [default: 30]
```

cpip.DupeRelink.**_copy_delete_duplicates_fix_links**(*hash_result*, *common_dir*, *nervous_mode*, *len_root_dir*)
Copy a single file that is duplicated to the common area, rewrite the links in that copy to the original location then delete all duplicates.

cpip.DupeRelink.**_get_hash_result**(*dir_path*, *file_glob*)
Returns a dict of {hash : [file_path, ...], ...} from a root directory.

cpip.DupeRelink.**_prepare_to_process**(*root_dir*, *file_glob*)
Create a dict {hash : [file_paths, ...], ...} for duplicated files

cpip.DupeRelink.**_prune_hash_result**(*hash_result*)
Prunes a dict of {hash : [file_path, ...], ...} to just those entries that have >1 file_path.

cpip.DupeRelink.**_replace_in_file**(*fpath*, *text_find*, *text_repl*, *nervous_mode*, *len_root_dir*)
Reads the contents of the file at fpath, replaces text_from with text_repl and writes it back out to the same fpath.

cpip.DupeRelink.**_rewrite_links_where_files_deleted**(*root_dir*, *sub_dir_for_common_files*, *nervous_mode*, *hash_result*, *len_root_dir*)
In the directories where we have deleted files rewrite the links to the common directory.

cpip.DupeRelink.**main**()
Delete and relink common files.

cpip.DupeRelink.**process**(*root_dir*, *sub_dir_for_common_files='_common_html'*, *file_glob='*.html'*, *nervous_mode=False*, *verbose=False*)
Process a directory in-place by making a single copy of common files, deleting the rest and fixing the links.

8.1.3 FileStatus - Show the status of a group of source files.

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/CPIPMain.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 ␣
↪cdb81d1eaaf6a1743e5182355f2e75bb  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.

__iadd__(*other*)
Add other to me.

__weakref__
list of weak references to the object (if defined)

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.

__weakref__
list of weak references to the object (if defined)

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]
```

8.1.4 IncList - List all `#include`'d files.

Command line tool to list included file paths

`cpip.IncList.`**`main`**`()`
Main command line entry point. Help:

```
Usage: IncList.py [options] files...
Preprocess the files and lists included files.

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -k                    Keep going. [default: False]
  -l LOGLEVEL, --loglevel=LOGLEVEL
                        Log Level (debug=10, info=20, warning=30, error=40,
                        critical=50) [default: 30]
  -n                    Nervous mode (do no harm). [default: False]
  -p                    Ignore pragma statements. [default: False]
  -I INCUSR, --usr=INCUSR
                        Add user include search path. [default: []]
  -J INCSYS, --sys=INCSYS
                        Add system include search path. [default: []]
  -P PREINC, --pre=PREINC
                        Add pre-include file path. [default: []]
  -D DEFINES, --define=DEFINES
                        Add macro defintions of the form name<=defintion>.
                        These are introduced into the environment before any
                        pre-include. [default: []]
```

`cpip.IncList.`**`preProcessForIncludes`**`(`*theItu*, *incUsr*, *incSys*, *theDefineS*, *preIncS*, *keepGoing*, *ignorePragma*`)`
Pre-process a file for included files.

`cpip.IncList.`**`retIncludedFileSet`**`(`*theLexer*`)`
Returns a set of included file paths from a lexer.

8.1.5 cpp - Mimic a preprocessor

cpip.cpp – Pretends to be like cpp, Will take options and a file (or stdin) and process it.

@author: Paul Ross

@copyright: 2015-2017 Paul Ross. All rights reserved.

```
(CPIP36) $ python src/cpip/cpp.py --help
usage: cpp.py [-h] [-v] [-t] [-V] [-d MACROOPTIONS] -E [-S PREDEFINES] [-C]
              [-D DEFINES] [-P PREINC] [-I INCUSR] [-J INCSYS]
              [path]

cpip.cpp -- Pretends to be like cpp, Will take options and a file (or stdin)

  Created by Paul Ross on 2015-01-16.
  Copyright 2015. All rights reserved.

  Licensed under the GPL License 2.0

USAGE

positional arguments:
  path                  Paths to source file. If absent then stdin is
                        processed. [default: None]

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         set verbosity level [default: 0]
  -t, --tokens          Show actual preprocessing tokens.
  -V, --version         show program's version number and exit
  -d MACROOPTIONS       Pre-processor options M, D and N. [default: []]
  -E                    Pre-process, required.
  -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: []]
```

8.2 Library

8.2.1 CppCondGraphToHtml

Writes out the Cpp Conditional processing graph as HTML.

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

Writing CppCondGraph visitor object.

__init__(*theHtmlStream*)
Constructor with an output XmlWrite.XhtmlStream.

Parameters **theHtmlStream** (*cpip.util.XmlWrite.XhtmlStream*) – The HTML stream.

Returns NoneType

visitPost(*theCcgNode*, *theDepth*)
Post-traversal call with a CppCondGraphNode and the integer depth in the tree.

Parameters

- **theCcgNode** (*cpip.core.CppCond.CppCondGraphNode*) – Graph node.
- **theDepth** (int) – Node depth.

Returns NoneType

visitPre(*theCcgNode*, *theDepth*)
Pre-traversal call with a CppCondGraphNode and the integer depth in the tree.

Parameters

- **theCcgNode** (*cpip.core.CppCond.CppCondGraphNode*) – Graph node.
- **theDepth** (int) – Node depth.

Returns NoneType

cpip.CppCondGraphToHtml.**linkToIndex**(*theS*, *theIdxPath*)
Write a back link to the index page.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – The HTML stream.
- **theIdxPath** (str) – <insert documentation for argument>

Returns NoneType

cpip.CppCondGraphToHtml.**processCppCondGrphToHtml**(*theLex*, *theHtmlPath*, *theTitle*, *theIdxPath*)
Given the PpLexer write out the Cpp Cond Graph to the HTML file.

Parameters

- **theLex** (*cpip.core.PpLexer.PpLexer*) – The lexer.
- **theHtmlPath** (str) – Path to output HTML file.
- **theTitle** (str) – Title.
- **theIdxPath** (str) – Path to index page for back links.

Returns NoneType

8.2.2 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.

ATTRS_LINE_CONDITIONAL_FROM = {'stroke': 'black', 'stroke-width': '0.25', 'stroke-dasharray': '8,2,2,2'}
Attributes, conditional line from.

ATTRS_LINE_CONDITIONAL_TO = {'stroke': 'black', 'stroke-width': '0.5', 'stroke-dasharray': '8,2,2,2'}
Attributes, conditional line to.

ATTRS_LINE_MT_FROM = {'stroke': 'aqua', 'stroke-width': '0.5', 'stroke-dasharray': '8,8'}
Attributes, empty line from.

ATTRS_LINE_MT_TO = {'stroke': 'aqua', 'stroke-width': '2', 'stroke-dasharray': '8,8'}
Attributes, empty line to.

ATTRS_LINE_NORMAL_FROM = {'stroke': 'black', 'stroke-width': '0.5'}
Attributes, normal line from.

ATTRS_LINE_NORMAL_TO = {'stroke': 'black', 'stroke-width': '2'}
Attributes, normal line to.

ATTRS_LINE_ROOT_CHILDREN_JOIN = {'stroke': 'lightgrey', 'stroke-width': '8'}
Lines joining root level children

ATTRS_NODE_CONDITIONAL = {'fill': 'salmon', 'stroke': 'black', 'stroke-width': '1'}
Conditionally compiled stuff

ATTRS_NODE_MT = {'fill': 'aqua', 'stroke': 'black', 'stroke-width': '1'}
Nodes that are empty

ATTRS_NODE_NORMAL = {'fill': 'mistyrose', 'stroke': 'black', 'stroke-width': '1'}
Node attributes (e.e. for rectangles) Normal nodes

CHEVRON_COLOUR_FILL = 'palegreen'
CSS for chevron attributes

COMMON_UNITS = 'mm'
Common units.

FILE_DEPTH = Dim(value=32.0, units='mm')
File box depth.

FILE_PADDING = Pad(prev=Dim(value=4.0, units='mm'), next=Dim(value=2.0, units='mm'), parent=Dim(value=16.0, u
Padding round file box..

HIST_DEPTH = Dim(value=4.0, units='mm')
Histogram depth.

HIST_LEGEND_ID = 'HistogramLegend'
Histogram rectangle ID.

HIST_PP_TOKEN_TYPES_COLOURS = (('header-name', 'orange'), ('identifier', 'blue'), ('string-literal', 'cyan'), ('pp-nu
This controls plot order as well as colour Note: Unusually they are in sweep='-' i.e logical left-to-right order

HIST_RECT_COLOUR_STROKE = 'black'
Histogram rectangle colour.

HIST_RECT_STROKE_WIDTH = '.5'
Histogram rectangle width.

POPUP_TEXT = ' ? '
The placeholder text for JavaScript rollover

SPACE_PARENT_CHILD = Dim(value=16.0, units='mm')
Space between parent and child.

STYLE_COURIER_10 = 'text.C10'
CSS for monospaced text `font-family="Courier" font-size="10" font-weight="normal"`

STYLE_RECT_INVIS = 'rect.invis'
CSS for invisible rectangle

STYLE_TEXT_SCALE = 'text.scale'
CSS used for scale

STYLE_VERDANA_12 = 'text.V12'
CSS entries

STYLE_VERDANA_9 = 'text.V9'
CSS used for histogram

WIDTH_MINIMUM = Dim(value=5, units='mm')
Minimum width

WIDTH_PER_TOKEN = Dim(value=0.001, units='mm')
Token width

_SVGTreeNodeMain__mustPlotChildHistogram()

Returns `bool` – `True` if the child histogram should be plotted.

_SVGTreeNodeMain__mustPlotSelfHistogram()

Returns `bool` – `True` if my histogram should be plotted.

__init__(*theFig*, *theLineNum*)
Constructor.

Parameters

- **theFig** (`NoneType, cpip.core.FileIncludeGraph.FileIncludeGraph`) – File include graph.
- **theLineNum** (`int`) – Line number.

Returns `NoneType`

_altTextsForTokenCount()
Returns a list of strings that are the alternate text for token counts.

Returns `list([str])` – Alternate text.

_fileIdStackToListOfStr(*theIdStack*)
Given a list of alternating file names and line numbers such as: `['root', 3, foo.h, 7, bar.h]` this returns a list of strings thus: `['root#3, 'foo.h#7, 'bar.h']`

Parameters **theIdStack** (`list([int, str]), list([str])`) – File ID stack.

Returns `list([str])` – Merged file stack.

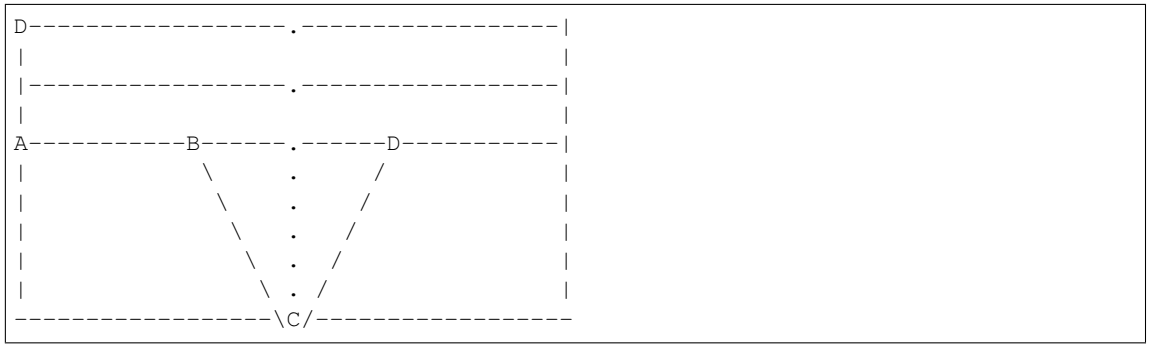
_fileNamePoint (*theDatumL*, *theTpt*)
Returns the point to plot the file name or None.

Parameters

- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Physical position.

_plotChevron (*theSvg*, *theDl*, *theTpt*)
Plots a wedge to represent the relative number of tokens in me and my children.



We plot in the order: D moveto A moveto B lineto C lineto D lineto B

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDl** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

_plotFileName (*theSvg*, *theDatumL*, *theTpt*)
Writes out the file name adjacent to the file box as static text.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

_plotFileNameStackPopup (*theSvg*, *theDatumL*, *theTpt*, *idStack*)
Writes out the file name at the top with a pop-up with the absolute path.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.
- **idStack** (list([int, str]), list([str])) – Stack of file IDs (paths).

Returns NoneType

_plotHistogram(*theSvg*, *theHistDl*, *theTpt*, *theTokCounter*)
Plots the histogram.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theHistDl** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.
- **theTokCounter** (*:py:main`cpip.core.PpTokenCount.PpTokenCount`*) – Token counter.

Returns NoneType

_plotHistogramLegend(*theSvg*, *theTpt*)
Plot a standardised legend. This is plotted as a group within a defs.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

_plotRootChildToChild(*theSvg*, *theDatumL*, *theTpt*)
Join up children of root node with vertical lines.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

_plotSelf(*theSvg*, *theDatumL*, *theTpt*, *thePassNum*, *idStack*)
Plot me to a stream at the logical datum point.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.
- **thePassNum** (int) – Pass number, if 1 then the histogram legend is written out.
- **idStack** (list([int, str]), list([str])) – Stack of file IDs (paths).

Returns NoneType

_plotSelfInternals(*theSvg*, *theDl*, *theTpt*)
Plot structures inside the box and the static text that is the file name.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDl** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

_plotSelfToChildren(*theSvg*, *theDatumL*, *theTpt*)
Plot links from me to my children to a stream at the (self) logical datum point.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

_plotTextOverlay(*theSvg*, *theDatumL*, *theTpt*, *idStack*)
Plots all the text associated with the parent and child. We write the hidden objects first then the visible objects. This is because the hidden objects are controlled onmouseover/onmouseout on the visible objects and they have to be later in the SVG file for this to work.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.
- **idStack** (list([int, str]), list([str])) – Stack of file IDs (paths).

Returns NoneType

_plotTextOverlayChildren(*theSvg*, *theDatumL*, *theTpt*)
Plot text associated with my children to a stream at the (self) logical datum point.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

_plotTextOverlayHistogram(*theSvg*, *theHistDl*, *theTpt*)
Plot the text associated with a histogram.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theHistDl** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

_plotTextOverlayTokenCountTable(*theSvg*, *theDatumL*, *theTpt*)
Plots the token count table as text+alternate text.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

_plotWhereWhyHow(*theSvg*, *iChild*, *theDatumL*, *theTpt*)
Plot description of Where/Why/How inclusion of a single child to a stream at the (self) logical datum point.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **iChild** (int) – Child ID.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

_writeScaleControls(*theSvg*)
Write the text elements that control re-scaling.

Parameters **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.

Returns NoneType

_writeTriggers(*theSvg*)
Write the rectangles that trigger pop-up text last so that they are on top.

Parameters **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.

Returns NoneType

condComp
A string of conditional tests.

Returns str – Conditional compilation string.

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

Returns NoneType

findLogic
The find logic as a string.

Returns list([str]) – The find logic.

plotFinalise(*theSvg*, *theDatumL*, *theTpt*)
Finish the plot. In this case we write the text overlays.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

plotInitialise(*theSvg*, *theDatumL*, *theTpt*)
Plot the histogram legend once only.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.

Returns NoneType

plotRoot(*theSvg*, *theDatumL*, *theTpt*, *passNum*)
Plot the root.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.

- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical position.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – Transformer of logical to physical points.
- **passNum** (int) – Pass number, if 1 then the histogram legend is written out.

Returns NoneType

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

Returns *cpip.core.PpTokenCount.PpTokenCount* – Token count.

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

Returns *cpip.core.PpTokenCount.PpTokenCount* – Token count.

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

Returns *cpip.core.PpTokenCount.PpTokenCount* – Token count.

writeAltTextAndMouseOverRect(*theSvg*, *theId*, *theAltPt*, *theAltS*, *theTrigPt*, *theTrigRect*)
Composes and writes the (pop-up) alternate text. Also writes a trigger rectangle.

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.
- **theId** (str) – The ID.
- **theAltPt** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Position of alternate text.
- **theAltS** (list([str])) – Alternate text lines.
- **theTrigPt** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Trigger position.
- **theTrigRect** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]), cpip.plot.Coord.Box([cpip.plot.Coord.Dim([int, str]), cpip.plot.Coord.Dim([int, <class 'str'>])])) – Trigger area.

Returns NoneType

writePreamble(*theS*)
Write any preamble such as CSS or JavaScript. To be implemented by child classes.

Parameters **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – SVG stream.

Returns NoneType

8.2.3 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.`**`CANVAS_PADDING`** **= Pad(prev=Dim(value=4.0, units='mm'), next=Dim(value=4.0, units='mm'),**
Canvas padding.

class `cpip.IncGraphSVGBase.`**`SVGTreeNodeBase`**(*theFig*, *theLineNum*)
Sub-class of *cpip.core.FileIncludeGraph.FigVisitorTreeNodeBase* for writing out the include graphs as an SVG file.

`ALT_FONT_FAMILY` **= 'monospace'**
Alternate text font.

`ALT_FONT_PROPERTIES` **= {'monospace': {'lenFactor': 0.5, 'heightFactor': 1.2, 'size': 10}, 'Courier': {'lenFactor': 0.5,**
Alternate text font properties.

`ALT_ID_SUFFIX` **= '.alt'**
Alternate text ID.

`ALT_RECT_FILL` **= 'khaki'**
Attributes for alternate text.

`COMMON_UNITS` **= 'mm'**
Units.

`NAMESPACE_XLINK` **= 'http://www.w3.org/1999/xlink'**
Namespace link.

`SCALE_FACTORS` **= (0.05, 0.1, 0.25, 0.5, 1.0, 1.5, 2.0)**
Used to rescale SVG rather than zooming in the browser as the latter is slow with Chrome and Safari (both WebKit) and pretty much everything else. Initial, presentational, scale is chose depending on the size of the diagram.

`SCALE_MAX_Y` **= Dim(value=1000, units='mm')**
Used to decide initial scale

`UNNAMED_UNITS` **= 'px'**
User units for viewBox and ploygon

`VIEWBOX_SCALE` **= 8.0**
Viewbox scale.

`__init__`(*theFig*, *theLineNum*)
Constructor.

Parameters

- **`theFig`** (`NoneType, cpip.core.FileIncludeGraph.FileIncludeGraph`) – The file include graph.
- **`theLineNum`** (`int`) – The line number of the parent file that included me.

Returns `NoneType`

`_enumerateChildren`(*theDatumL*, *theTpt*)
Generates a tuple of (index, logical_datum_point) for my children.

Parameters

- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The coordinate.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – The transformer.

Returns NoneType,tuple([int, cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])]) – The children

_plotRootChildToChild(*theSvg*, *theDatumL*, *theTpt*)
In the case of me being root this plots child to child.

_plotSelf(*theSvg*, *theDatumL*, *theTpt*, *idStack*)
Plot me to a stream at the logical datum point. Must be provided by child classes.

_plotSelfToChildren(*theSvg*, *theDatumL*, *theTpt*)
In the case of me being not root this plots me to my children.

_writeAlternateText(*theSvg*, *thePoint*, *theId*, *theText*, *theAltS*, *yOffs=Dim(value=0*, *units='pt')*)
Composes and writes the (pop-up) alternate text. thePoint is the physical point to locate both texts.

_writeECMAScript(*theSvg*)
Writes the ECMA script for pop-up text switching.

Parameters **theSvg**(*:py:class`cpip.plot.SVGWriter.SVGWriter`*) – The SVG stream.

Returns NoneType

_writeStringListToTspan(*theSvg*, *thePointX*, *theList*)
Converts a multi-line string to tspan elements in monospaced format.

This writes the tspan elements within an existing text element, thus:

```
<text id="original.alt" font-family="Courier" font-size="12" text-anchor=
↪"middle" x="250" y="250">
    <tspan xml:space="preserve"> One</tspan>
    <tspan x="250" dy="1em" xml:space="preserve"> Two</tspan>
    <tspan x="250" dy="1em" xml:space="preserve">Three</tspan>
</text>
```

Parameters

- **theSvg** (*cpip.plot.SVGWriter.SVGWriter*) – The SVG stream.
- **thePointX** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Coordinate.
- **theList** (list([str])) – List of strings to write.

Returns NoneType

bb
Returns a *cpip.plot.PlotNode.PlotNodeBboxBoxy* object for this node.

commentFunctionBegin(*theSvg*, ***kwargs*)
Injects a comment into the SVG with the start of the executing function name. self.TRACE must be True to enable this.

Parameters **theSvg** (*:py:class`cpip.plot.SVGWriter.SVGWriter`*) – The SVG stream.

Returns NoneType

commentFunctionEnd (*theSvg*, ***kwargs*)
Injects a comment into the SVG with the completion of the executing function name. `self.TRACE` must be `True` to enable this.

Parameters **theSvg** (*:py:class`cpip.plot.SVGWriter.SVGWriter`*) – The SVG stream.

Returns NoneType

condCompState
True/False if conditionally compiled node.

dumpToStream (*theS=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *p=''*)
Debug/trace.

finalise ()
This will be called on finalisation. For depth first finalisation the child class should call finalise on each child first.

nodeName
This is the file name or 'Root'.

numTokens
The number of significant tokens for me only (not including children).

plotCanvas
The logical size of the plot canvas as a `cpip.plot.Coord.Box`.

plotFinalise (*theSvg*, *theDatumL*, *theTpt*)
Called once immediately before the plot is closed. Can be overridden in child classes for specific use.

plotInitialise (*theSvg*, *theDatumL*, *theTpt*)
Called once immediately before the recursive plotToSVGStream(). Can be overridden in child classes for specific use.

plotRoot (*theSvg*, *theDatumL*, *theTpt*, *passNum*)
Call to plot any root node, for example our child class uses this to plot the histogram legend before starting on the text.

plotToFileObj (*theFileObj*, *theTpt*)
Root level call to plot to a file object. The SVG stream is created here.

Parameters

- **theFileObj** (`_io.TextIOWrapper`) – The output file object.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – The transformer used to transform the internal logical coordinates to physical plot positions.

Returns NoneType

plotToFilePath (*theFileName*, *theTpt*)
Root level call to plot to a SVG file, theTpt is an *cpip.plot.TreePlotTransform.TreePlotTransform* object and is used to transform the internal logical coordinates to physical plot positions.

Parameters

- **theFileName** (`str`) – File path for output.

- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – The transformer used to transform the internal logical coordinates to physical plot positions.

Returns NoneType

plotToSVGStream(*theSvg*, *theDatumL*, *theTpt*, *passNum*, *idStack*)
Plot me to a stream and my children at the logical datum point, this is a recursive call.

Parameters

- **theSvg** (*:py:class`cpip.plot.SVGWriter.SVGWriter`*) – The SVG stream.
- **theDatumL** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The Datum.
- **theTpt** (*cpip.plot.TreePlotTransform.TreePlotTransform*) – The transformer used to transform the internal logical coordinates to physical plot positions.
- **passNum** (int) – The pass number.
- **idStack** (list([]), list([int, str]), list([str])) – Stack of id's.

Returns NoneType

writePreamble(*theS*)
Write any preamble such as CSS or JavaScript. To be implemented by child classes.

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

Parameters

- **theLex** (*cpip.core.PpLexer.PpLexer*) – The lexer.
- **theFilePath** (str) – Where to write the SVG file.
- **theClass** (type) – CSS class.
- **tptPos** (str) – Transform position.
- **tptSweep** (str) – Sweep direction.

Returns NoneType

8.2.4 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.

class cpip.IncGraphXML.**IncGraphXML**(*theFig*, *theLineNum*)
Class that creates an include graph in XML.

__init__(*theFig*, *theLineNum*)
Constructor.

Parameters

- **theFig** – The file include graph.
- **theLineNum** – The line number.

_writeSelf(*theS*)
Plot me to a stream at the logical datum point. Must be provided by child classes.

finalise()
This will be called on finalisation. This just accumulates the child token counter.

tokenCounter
This is the computed PpTokenCount.PpTokenCount() me only.

tokenCounterChildren
This is the computed PpTokenCount.PpTokenCount() for all my children but not me.

writeToFileObj(*theFileObj*)
Root level call to plot to a file object. The SVG stream is created here.

writeToFilePath(*theFileName*)
Root level call to plot to a SVG file, theTpt is an TreePlotTransform object and is used to transform the internal logical coordinates to physical plot positions.

writeToSVGStream(*theS*)
Write me to a stream and my children at the logical datum point, this is a recursive call.

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

8.2.5 ItuToHtml

Converts an Initial Translation Unit (ITU) i.e. a file, to HTML.

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

__init__(*theItu*, *theHtmlDir*, *keepGoing=False*, *macroRefMap=None*, *cppCondMap=None*, *ituToTuLineSet=None*)
Takes an input source file and an output directory.

Parameters

- **theItu** (str) – The original source file path (or file like object for the input).
- **theHtmlDir** (str) – The output directory for the HTML or a file-like object for the output.
- **keepGoing** (bool) – If True keep going as far as possible.
- **macroRefMap** (dict({str : [list([tuple([<class 'str'>, <class 'int'>, str])]), list([tuple([<class 'str'>, int, str])]), list([tuple([str, int, str])])]})) – Map of {identifier : href_text, ...) to link to macro definitions.
- **cppCondMap** (*cpip.core.CppCond.CppCondGraphVisitorConditionalLines*) – Conditional compilation map.
- **ituToTuLineSet** (NoneType, set([int])) – Set of integer line numbers which are lines that can be linked to the translation unit representation.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

_convert()
Convert ITU to HTML.

Returns NoneType

Raises ExceptionItuToHTML on failure.

_handleToken(*theS*, *t*, *tt*)
Handle a token.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – The HTML stream.
- **t** (str) – The token text.
- **tt** (str) – The token type.

Returns NoneType

_incAndWriteLine(*theS*)
Write a line.

Parameters **theS** (*cpip.util.XmlWrite.XhtmlStream*) – The HTML stream.

Returns NoneType

_initReader()
Create and return a reader, initialise internals.

Returns *cpip.core.ItuToTokens.ItuToTokens* – The file tokeniser.

_writeTextWithNewlines(*theS*, *theText*, *spanClass*)
Splits text by newlines and writes it out.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – The HTML stream.
- **theText** (str) – The text to write.
- **spanClass** (NoneType, str) – CSS class.

Returns NoneType

8.2.6 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 (>=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
- *Optionally: This macro depends on the following macros:*
- *Optionally: 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 with the same name. 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.**TITLE_ANCHOR_LINKTEXT_MACROS_HISTORY = ('Macro Usage (referenced macros only)', '**
Visible links to macro history.

cpip.MacroHistoryHtml.**TITLE_ANCHOR_LINKTEXT_MACROS_IN_SCOPE = ('Macros In Scope', 'Macros_In_Scope', '**
Visible links to macros in scope.

cpip.MacroHistoryHtml.**TITLE_ANCHOR_LINKTEXT_MACROS_TABLE = ('Macro Environment (all macros, alphabetica**
Visible links to macros.

cpip.MacroHistoryHtml.**TITLE_ANCHOR_LINKTEXT_MACROS_TESTED_WHEN_NOT_DEFINED = ('Macros Tested Wh**
Visible links to macros tested when not defined.

cpip.MacroHistoryHtml.**_getMacroDependencyTrees**(*theMacroAdjList*, *theMacro*)
Returns the dependency trees (parent/child, child/parent) for the macro. Either can be None.

Parameters

- **theMacroAdjList** (*:py:class`cpip.util.Tree.DuplexAdjacencyList`*) – The adjacency list.
- **theMacro** (*cpip.core.PpDefine.PpDefine*) – The macro.

Returns `tuple([NoneType, NoneType]),tuple([NoneType, cpip.util.Tree.Tree]),tuple([cpip.util.Tree.Tree, NoneType])` – A pair of parent -> child tree and child -> parent tree.

cpip.MacroHistoryHtml.**_linkToIndex**(*theS*, *theIdx*)
Write a link to 'Return to' the index page.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – HTML stream.
- **theIdx** (str) – The index link.

Returns NoneType

cpip.MacroHistoryHtml.**_macroHistoryIndexName**(*theItu*)

Parameters **theItu** (str) – Ignored.

Returns str – The HTML file name of the macro history.

cpip.MacroHistoryHtml.**_macroHistoryNorefName**(*theItu*)

Parameters **theItu** (str) – Ignored.

Returns str – The HTML file name of the macro history with no references.

cpip.MacroHistoryHtml.**_macroHistoryRefName**(*theItu*)

Parameters **theItu** (str) – Ignored.

Returns str – The HTML file name of the macro history with references.

cpip.MacroHistoryHtml.**_macroIdTestedWhenNotDefined**(*theMacroId*)

Parameters **theMacroId** (str) – The macro ID.

Returns str – Link ID.

cpip.MacroHistoryHtml.**_retMacroId**(*theMacro*, *theIndex=None*)

Parameters

- **theMacro** (*cpip.core.PpDefine.PpDefine*) – The macro.
- **theIndex** (int) – The index.

Returns str – Encoded XML string.

cpip.MacroHistoryHtml.**_retMacroIdHrefNames**(*theEnv*, *theItu*)
Returns a dict of {identifier : [(fileId, lineNum, href_name), ...], ...} for annotating HTML.

The order in the list is the translation unit order in which macros are defined/undef'd.

Parameters

- **theEnv** (*cpip.core.MacroEnv.MacroEnv*) – The macro environment.
- **theItu** (str) – Path to the Initial Translation Unit (ITU).

Returns dict({str : list([tuple([str, int, str])])}) – Identifier dictionary.

cpip.MacroHistoryHtml.**_retSetMacros**(*theEnv*, *isReferenced*, *isActive*)
Returns a set of {(Identifier, href_name), ...} of macros identifiers and their references. Multiple identifier that have been def'd/undef'd have unique, lexagraphically sequential hrefs (with a trailing integer).

Parameters

- **theEnv** (*cpip.core.MacroEnv.MacroEnv*) – The macro environment.
- **isReferenced** (bool) – If True only macros that are referenced are included.
- **isActive** (bool) – Only currently active macros are included, undef'd ones are excluded.

Returns `set([]),set([tuple([str, str])])` – <insert documentation for return values>

Raises `StopIteration`

`cpip.MacroHistoryHtml.`**`_tdFilePathCallback`**(*theS*, *attrs*, *k*, *v*)
Callback function for the file reference table.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – HTML stream.
- **attrs** (`dict({str : [str]})`) – Element attributes.
- **k** (`list([str])`) – Keys.
- **v** (`list([tuple([str, str])])`) – Values.

Returns `NoneType`

`cpip.MacroHistoryHtml.`**`_writeMacroDefinitionAndAnchor`**(*theS*, *theDef*, *theIntOccurence*)
Writes a definition of the macro with an anchor that can be linked to. This also writes an anchor based on the first character of the macro name so that alphabetic links can reference it.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – The HTML stream.
- **theDef** (*cpip.core.PpDefine.PpDefine*) – The macro.
- **theIntOccurence** (`int`) – The occurence of the macro.

Returns `str` – Anchor name.

`cpip.MacroHistoryHtml.`**`_writeMacroDependencies`**(*theS*, *theEnv*, *theMacro*, *theMacroAdjList*, *theItu*)
Writes out the macro dependencies.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – HTML stream.
- **theEnv** (*cpip.core.MacroEnv.MacroEnv*) – The macro environment.
- **theMacro** (*cpip.core.PpDefine.PpDefine*) – The macro definition.
- **theMacroAdjList** (`cpip.util.Tree.DuplexAdjacencyList`) – Dependency adjacency list.
- **theItu** (`str`) – The Initial Translation Unit (ITU).

Returns `NoneType`

`cpip.MacroHistoryHtml.`**`_writeMacroDependenciesTable`**(*theS*, *theEnv*, *theAdjList*, *theItu*)
Writes all the macro dependencies to a rowspan/colspan HTML that references something.

table with links to the position in the HTML representation of the file

This uses a particular design pattern that uses a DictTree to sort out the rows and columns. In this case the DictTree values are lists of pairs `(href, nav_text)` where `nav_text` is the line_col of the referencing file.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – HTML stream.
- **theEnv** (*cpip.core.MacroEnv.MacroEnv*) – The macro environment.
- **theAdjList** (*cpip.util.Tree.Tree*) – Dependency adjacency list.

- **theItu** (str) – The Initial Translation Unit (ITU).

Returns NoneType

cpip.MacroHistoryHtml.**_writeMacroHistory**(*theS*, *theMacro*, *theOmitFiles*, *theIntOccurence*)
Writes out the macro history from a PpDefine object. theMacro - a PpDefine() object. theOmitFiles - a list of pseudo files not to link to e.g. ['Unnamed Pre-include',].

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – The HTML stream.
- **theMacro** (*cpip.core.PpDefine.PpDefine*) – The macro.
- **theOmitFiles** (list([str])) – A list of pseudo files not to link to e.g. ['Unnamed Pre-include',].
- **theIntOccurence** (int) – The occurence of the macro which will be added to the link ID.

Returns NoneType

cpip.MacroHistoryHtml.**_writeMacroReferencesTable**(*theS*, *theFlcS*)
Writes all the references to a file/line/col in a rowspan/colspan HTML table with links to the position in the HTML representation of the file that references something.

This uses a particular design pattern that uses a DictTree to sort out the rows and columns. In this case the DictTree values are lists of pairs (href, nav_text) where nav_text is the line-col of the referencing file.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – HTML stream.
- **theFlcS** (list([]), list([cpip.core.FileLocation.FileLineCol([str, int, int])])) – File location.

Returns NoneType

cpip.MacroHistoryHtml.**_writeMacrosTestedButNotDefined**(*theS*, *theMacroId*, *theEnv*)
Writes out the macro history for macros tested but not defined.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – The HTML stream.
- **theMacroId** (str) – Macro name.
- **theEnv** (*cpip.core.MacroEnv.MacroEnv*) – Macro environment.

Returns NoneType

cpip.MacroHistoryHtml.**_writeSectionOnMacroHistory**(*theS*, *theEnv*, *theOmitFiles*, *theHtmlPath*, *theItu*, *isReferenced*)
Write the section that says where macros were used with links to the file/line/column.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – The HTML stream.
- **theEnv** (*cpip.core.MacroEnv.MacroEnv*) – The macro environment.
- **theOmitFiles** (list([str])) – A list of pseudo files not to link to e.g. ['Unnamed Pre-include',].
- **theHtmlPath** (str) – The path to the HTML file.
- **theItu** (str) – Initial Translation Unit (ITU).

- **isReferenced** (bool) – True if macro was referenced (used).

Returns dict({str : [str]}) – Map of {macro_identifier : file_id_link, ...}`

Raises StopIteration

cpip.MacroHistoryHtml.**_writeTableOfMacros**(*theS*, *theEnv*, *theHtmlPath*)
Writes the table of macros, where they are defined, their ref count etc.

cpip.MacroHistoryHtml.**_writeTd**(*theStream*, *theStr*)
Write a <td> element and contents.

cpip.MacroHistoryHtml.**_writeTdMacro**(*theS*, *theDef*, *startLetter*, *theIntOccurence*)
Write the macro cell in the general table. theDef is a PpDefine object.

cpip.MacroHistoryHtml.**_writeTh**(*theStream*, *theStr*)
Write a <th> element and contents.

cpip.MacroHistoryHtml.**_writeTocMacros**(*theS*, *theEnv*, *isReferenced*, *filePrefix*)
Write out the table of contents from the environment. isReferenced controls whether these are referenced macros (interesting) or non referenced macros (a larger, less interesting set). filePrefix - If not None this is the HTML file to link to.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – HTML stream.
- **theEnv** (*cpip.core.MacroEnv.MacroEnv*) – The macro environment.
- **isReferenced** (bool) – Write out only referenced macros.
- **filePrefix** (NoneType, str) – File prefix for the href's.

Returns NoneType

cpip.MacroHistoryHtml.**_writeTrMacro**(*theS*, *theHtmlPath*, *theMacro*, *theIntOccurence*, *theStartLetter*, *retMap*)
Write the macro as a row in the general table. theMacro is a PpDefine object. theStartLetter is the current letter we are writing ['A', 'B', ...] which writes an anchor at the beginning of each letter section.

cpip.MacroHistoryHtml.**processMacroHistoryToHtml**(*theLex*, *theHtmlPath*, *theItu*, *theIndexPath*)
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.

Parameters

- **theLex** (*cpip.core.PpLexer.PpLexer*) – The lexer.
- **theHtmlPath** (str) – File path to write to.
- **theItu** (str) – Path to the initial translation unit (ITU).
- **theIndexPath** (str) – Path to the index.

Returns tuple([dict({str : [list([tuple([<class 'str'>, <class 'int'>, str])]), list([tuple([<class 'str'>, int, str])]), list([tuple([str, int, str])])]}), str]) – Map that links macro names ot file positions.

cpip.MacroHistoryHtml.**splitLine**(*theStr*, *splitLen=60*, *splitLenHard=80*)
Splits a long string into string that is a set of lines with continuation characters.

Parameters

- **theStr** (str) – Long line.
- **splitLen** (int) – Soft split length.
- **splitLenHard** (int) – Hard split length.

Returns str – Line with continuation character and newlines.

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.

Parameters

- **sIn** (str) – Line to split.
- **splitLen** (int) – Soft split length.
- **splitLenHard** (int) – Hard split length.

Returns list([str]) – List of lines.

8.2.7 TokenCss

CSS Support for ITU+TU files in HTML.

cpip.TokenCss.**ENUM_TT_MAP = {'j': 'trigraph', 'm': 'keyword', 'f': 'preprocessing-op-or-punc', 'b': 'identifier', 'o': 'Unk**
Reverse map of {enum_int : token_type, ...}

cpip.TokenCss.**TT_ENUM_MAP = {'character-literal': 'd', 'whitespace': 'h', 'identifier': 'b', 'C++ comment': 'l', 'string-lite**
Map of {token_type : enum_int, ...}

cpip.TokenCss.**retClass**(*theTt*)

Parameters **theTt** (str) – Token type

Returns str – CSS class.

Raises ExceptionTokenCss For unknown token type.

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.

Parameters **theDir** (str) – Directory.

Returns NoneType

8.2.8 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.**FILE_SHIFT_ACTIONS = ('Starting', 'Holding', 'Back in', 'Ending')**
State changes

cpip.Tu2Html.**_adjustFileStack**(*theS*, *lexStack*, *theFileStack*, *theIntId*)
Adjust the file stacks and write to the stream.

<insert documentation for function>

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – HTML stream.
- **lexStack** (list([str])) – ???
- **theFileStack** (list([]), list([str])) – ???
- **theIntId** (int) – ???

Returns int – theIntId

cpip.Tu2Html.**_writeFileName**(*theS*, *lenIndent*, *theFileStack*, *theAction*, *theFile*, *theIntId*)

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – HTML stream.
- **lenIndent** (int) – Size of indent.
- **theFileStack** (list([str])) – File stack.
- **theAction** (str) – One of ('Starting', 'Holding', 'Back in', 'Ending')
- **theFile** (str) – File ID such as a path.
- **theIntId** (int) – ???

Returns int – theIntId

cpip.Tu2Html.**linkToIndex**(*theS*, *theIdxPath*)
Write a link to the index.

Parameters

- **theS** (*cpip.util.XmlWrite.XhtmlStream*) – HTML stream.
- **theIdxPath** (str) – Path to the index.

Returns NoneType

cpip.Tu2Html.**processTuToHtml**(*theLex*, *theHtmlPath*, *theTitle*, *theCondLevel*, *theIdxPath*, *incItuAnchors=True*)
Processes the PpLexer and writes the tokens to the HTML file.

Parameters

- **theLex** (*cpip.core.PpLexer.PpLexer*) – The lexer.
- **theHtmlPath** (str) – The path to the HTML file to write.
- **theTitle** (str) – A string to go into the <title> element.
- **theCondLevel** (int) – The Conditional level to pass to theLex.ppTokens()
- **theIdxPath** (str) – Path to link back to the index page.
- **incItuAnchors** (bool) – If True will write anchors for lines in the ITU that are in this TU. If True then setItuLineNumbers returned is likely to be non-empty.

Returns tuple([cpip.core.PpTokenCount.PpTokenCount, set([int])]) – 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" />`.

Raises `StopIteration`

8.2.9 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.

8.2.10 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 evaluate it as a constant expression.

`__init__`(*theTokTypeS*)
Constructor takes a list pf PpToken.

Parameters **`theTokTypeS`** (`list([cpip.core.PpToken.PpToken])`) – List of tokens.

Returns `NoneType`

`__weakref__`
list of weak references to the object (if defined)

`_evaluateConditionalExpression`(*theMatch*)
Evaluates a conditional expression e.g. expr ? t : f Which we convert with a regular expression to:

```
if exp:
    t
else:
    f
```

`_evaluateExpression`(*theStr*)
Evaluates a conditional expression e.g. 1 < 2

Parameters **`theStr`** (`str`) – The string to evaluate in the Pythoin environment.

Returns `int` – Result of `eval()`, 0 is success.

Raises `ExceptionEvaluateExpression` on failure.

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

Returns `int` – Result of `eval()`, 0 is success.

Raises `ExceptionEvaluateExpression` on failure.

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

Returns `str` – Result.

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
---------   --------                -----------------------
#if         constant-expression     s.push(bool)
#ifdef      identifier              s.push(bool)
#ifndef     identifier              s.push(!bool)
#elif       constant-expression     s.pop(), s.push(bool)
```

```
#else       N/A                     Either s.push(!s.pop()) or s.flip()
#endif      N/A                     s.pop()
```

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
---------   --------                ------------------
#if         constant-expression     m.push(['%s' % tokens,])
#ifdef      identifier              m.push(['(defined %s)' % identifier)])
#ifndef     identifier              m.push(['!(defined %s)' % identifier)])
#elif       constant-expression     m[-1].push('!%s' % m[-1].pop()),
                                    m[-1].push(['%s' % tokens,])
                                    Note: Here we flip the existing state via
                                    a push(!pop())) then push the additional
                                    condition so that we have multiple
                                    contitions 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"`

cpip.core.CppCond.**CPP_COND_ALT_DIRECTIVES = ('else', 'elif')**
Conditional alternative directives.

cpip.core.CppCond.**CPP_COND_DIRECTIVES = ('if', 'ifdef', 'ifndef', 'elif', 'else', 'endif')**
Conditional directives.

cpip.core.CppCond.**CPP_COND_END_DIRECTIVE = 'endif'**
Conditional end directive.

cpip.core.CppCond.**CPP_COND_IF_DIRECTIVES = ('if', 'ifdef', 'ifndef')**
Conditional 'if' directives.

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

__init__ (*theState*, *theIdOrCondExpr*)
theState is a boolean and theIdOrCondExpr is a string representing a constant-expression or identifier.

The boolean state of this has restrictions appropriate to `#if/#elif/#else` processing in that the can not transition `True->False->True` i.e. can only have one True state.

Of course `False->True->False` is permitted.

Parameters

- **theState** (`bool, int`) – State.
- **theIdOrCondExpr** (`str`) – Constant expression.

Returns `NoneType`

__weakref__
list of weak references to the object (if defined)

_add (*theConstExpr*)
Add a string to the list of constant expressions. Newline is replaced with a single space.

Parameters **theConstExpr** (`str`) – Constant expression.

Returns `NoneType`

constExprStr (*invert=False*)
Returns self as a string which is the concatenation of constant-expressions.

Parameters **invert** (`bool`) – Negate the test.

Returns `str` – Constant expression.

flip()
Inverts the boolean such as for #else directive.

Returns `NoneType`

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.

Parameters

- **theBool** (`bool`) – Negate the state.
- **theConstExpr** (`str`) – Constant expression.

Returns `NoneType`

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

Returns `int` – State.

negateLastState()
Inverts the state of the last item on the stack.

state
Returns boolean state of self.

Returns `bool,int` – State.

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 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`.

__bool__()
Syntactic sugar for truth testing, wraps isTrue().

__init__()
Constructor, this just initialise the internal state.

__str__()
Returns a string representation of the stack.

Note: This returns a string that says 'if my state were True then this is why. This string does not explain actual state, for that consult *isTrue()*.

__weakref__
list of weak references to the object (if defined)

_flip()
Changes the state of the top *ConditionalState* object on the stack.

Returns `NoneType`

_pop()
Removes a *ConditionalState* object from the stack. The removed object is returned.

Returns `cpip.core.CppCond.ConditionalState` – Pop'd value.

_push(*theBool*, *theIce*)
Pushes a new *ConditionalState* object onto the stack.

Parameters

- **theBool** (`bool, int`) – State.
- **theIce** (`str`) – ???

Returns `NoneType`

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

Returns `NoneType`

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.

Returns `int` – State.

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

Returns `bool` – State of the stack.

oElif(*theBool*, *theConstExpr*)
Deal with the result of a `#elif`.

Parameters

- **theBool** (`bool`) – Is a boolean that is the result of the callers evaluation of a constant-expression.
- **theConstExpr** (`str`) – 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`.

Returns `NoneType`

oElse()
Deal with the result of a `#else`.

Returns `NoneType`

oEndif()
Deal with the result of a `#endif`.

Returns `NoneType`

oIf(*theBool*, *theConstExpr*)
Deal with the result of a `#if`.

Parameters

- **theBool** (`bool`) – Is a boolean that is the result of the callers evaluation of a constant-expression.
- **theConstExpr** (`str`) – 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`.

Returns `NoneType`

oIfdef(*theBool*, *theConstExpr*)
Deal with the result of a `#ifdef`.

Parameters

- **theBool** (`bool`) – Is a boolean that is the result of the callers evaluation of a constant-expression.
- **theConstExpr** (`str`) – 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`.

Returns `NoneType`

oIfndef(*theBool*, *theConstExpr*)
Deal with the result of a `#ifndef`.

Parameters

- **theBool** (`bool`) – Is a boolean that is the result of the callers evaluation of a constant-expression.
- **theConstExpr** (`str`) – 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`.

Returns `NoneType`

stackDepth
Returns the depth of the conditional stack as an integer.

class cpip.core.CppCond.**CppCondGraph**
Represents a graph of conditional preprocessing directives.

__weakref__
list of weak references to the object (if defined)

_oIfIfDefIfndef(*theCppD*, *theFlc*, *theTuIdx*, *theBool*, *theCe*)
Generic preprocessor directive handler.

Parameters

- **theCppD** (str) – The preprocessor directive.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – A cpip.core.FileLocation.FileLineColumn object that identifies the position in the file.
- **theTuIdx** (int) – An integer that represents the position in the translation unit.
- **theBool** (bool) – The current state of the conditional stack.
- **theCe** (str) – The constant expression as a string (not evaluated).

Returns NoneType

_raiseIfComplete(*theCppD*)
Raise an exception if I can not accept this directive, does not apply to #if statements so should not be called for them.

Parameters **theCppD** (str) – The preprocessor directive.

Returns NoneType

Raises ExceptionCppCondGraph

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

Returns bool – True if complete.

oElif(*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #elif.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – A cpip.core.FileLocation.FileLineColumn object that identifies the position in the file.
- **theTuIdx** (int) – An integer that represents the position in the translation unit.
- **theBool** (bool) – The current state of the conditional stack.
- **theCe** (str) – The constant expression as a string (not evaluated).

Returns NoneType

oElse(*theFlc*, *theTuIdx*, *theBool*)
Deal with the result of a #else.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – A cpip.core.FileLocation.FileLineColumn object that identifies the position in the file.
- **theTuIdx** (int) – An integer that represents the position in the translation unit.
- **theBool** (bool) – The current state of the conditional stack.

Returns NoneType

oEndif (*theFlc*, *theTuIdx*, *theBool*)
Deal with the result of a #endif.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – A cpip.core.FileLocation.FileLineColumn object that identifies the position in the file.
- **theTuIdx** (int) – An integer that represents the position in the translation unit.
- **theBool** (bool) – The current state of the conditional stack.

Returns NoneType

oIf (*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #if.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – A cpip.core.FileLocation.FileLineColumn object that identifies the position in the file.
- **theTuIdx** (int) – An integer that represents the position in the translation unit.
- **theBool** (bool) – The current state of the conditional stack.
- **theCe** (str) – The constant expression as a string (not evaluated).

Returns NoneType

oIfdef (*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #ifdef.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – A cpip.core.FileLocation.FileLineColumn object that identifies the position in the file.
- **theTuIdx** (int) – An integer that represents the position in the translation unit.
- **theBool** (bool) – The current state of the conditional stack.
- **theCe** (str) – The constant expression as a string (not evaluated).

Returns NoneType

oIfndef (*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #ifndef.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – A cpip.core.FileLocation.FileLineColumn object that identifies the position in the file.

- **theTuIdx** (int) – An integer that represents the position in the translation unit.
- **theBool** (bool) – The current state of the conditional stack.
- **theCe** (str) – The constant expression as a string (not evaluated).

Returns NoneType

visit(*theVisitor*)
Take a visitor object and pass it around giving it each *CppCondGraphNode* object.

Parameters **theVisitor** (cpip.CppCondGraphToHtml.CcgVisitorToHtml, cpip.core.CppCond.CppCondGraphVisitorConditionalLines) – The visitor.

Returns NoneType

class cpip.core.CppCond.**CppCondGraphIfSection**(*theIfCppD*, *theFlc*, *theTuIdx*, *theBool*, *theCe*)
Class that represents a conditionally compiled section starting with #if... and ending with #endif.

__init__(*theIfCppD*, *theFlc*, *theTuIdx*, *theBool*, *theCe*)
Constructor.

Parameters

- **theIfCppD** (str) – A string, one of '#if', '#ifdef', '#ifndef'.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – A cpip.core.FileLocation.FileLineColumn object that identifies the position in the file.
- **theTuIdx** (int) – An integer that represents the position in the translation unit.
- **theBool** (bool) – The current state of the conditional stack.
- **theCe** (str) – The constant expression as a string (not evaluated).

Returns NoneType

__weakref__
list of weak references to the object (if defined)

_oIfIfDefIfndef(*theCppD*, *theFlc*, *theTuIdx*, *theBool*, *theCe*)
Generic if function.

Parameters

- **theCppD** (str) – Preprocessor directive.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.
- **theCe** (str) – The preprocessor directive.

Returns NoneType

_raiseIfSectionComplete(*theCppD*)

Parameters **theCppD** (str) – Preprocessor directive.

Returns NoneType

Raises ExceptionCppCondGraphIfSection If the section is complete.

isSectionComplete

Returns bool – Section complete.

oElif(*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #elif.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.
- **theCe** (str) – The preprocessor directive.

Returns NoneType

oElse(*theFlc*, *theTuIdx*, *theBool*)
Deal with the result of a #else.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.

Returns NoneType

oEndif(*theFlc*, *theTuIdx*, *theBool*)
Deal with the result of a #endif.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.

Returns NoneType

oIf(*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #if.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.
- **theCe** (str) – The preprocessor directive.

Returns NoneType

oIfdef(*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #ifdef.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.
- **theCe** (str) – The preprocessor directive.

Returns NoneType

oIfndef(*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #ifndef.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.
- **theCe** (str) – The preprocessor directive.

Returns NoneType

visit(*theVisitor*, *theDepth*)
Take a visitor object make the pre/post calls.

Parameters

- **theVisitor** (cpip.CppCondGraphToHtml.CcgVisitorToHtml, cpip.core.CppCond.CppCondGraphVisitorConditionalLines) – Visitor.
- **theDepth** (int) – Graph depth.

Returns NoneType

class cpip.core.CppCond.**CppCondGraphNode**(*theCppDirective*, *theFileLineCol*, *theTuIdx*, *theBool*, *theConstExpr=None*)
Base class for all nodes in the *CppCondGraph*.

__init__(*theCppDirective*, *theFileLineCol*, *theTuIdx*, *theBool*, *theConstExpr=None*)
Constructor.

Parameters

- **theCppDirective** (str) – Preprocessor directive.
- **theFileLineCol** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – ???
- **theConstExpr** (NoneType, str) – The constant expression.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

_oIfIfDefIfndef(*theCppD*, *theFlc*, *theTuIdx*, *theBool*, *theCe*)
Generic if function.

Parameters

- **theCppD** (str) – Preprocessor directive.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.
- **theCe** (str) – The preprocessor directive.

Returns NoneType

_raiseIfCanNotAccept (*theCppD*)
Raise an exception if I can not accept this directive.

Parameters **theCppD** (str) – Preprocessor directive.

Returns NoneType

Raises ExceptionCppCondGraphNode If the section is complete.

canAccept (*theCppD*)
True if I can accept a Preprocessing Directive; theCppD.

Parameters **theCppD** (str) – Preprocessor directive.

Returns bool – I can accept it.

oElif (*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #elif.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.
- **theCe** (str) – The preprocessor directive.

Returns NoneType

oElse (*theFlc*, *theTuIdx*, *theBool*)
Deal with the result of a #else.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.

Returns NoneType

oEndif (*theFlc*, *theTuIdx*, *theBool*)
Deal with the result of a #endif.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.

- **theBool** (bool) – Conditional compilation state.

Returns NoneType

oIf (*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #if.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.
- **theCe** (str) – The preprocessor directive.

Returns NoneType

oIfdef (*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #ifdef.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.
- **theCe** (str) – The preprocessor directive.

Returns NoneType

oIfndef (*theFlc*, *theTuIdx*, *theBool*, *theCe*)
Deal with the result of a #ifndef.

Parameters

- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File location.
- **theTuIdx** (int) – Translation unit index.
- **theBool** (bool) – Conditional compilation state.
- **theCe** (str) – The preprocessor directive.

Returns NoneType

retStrList (*theDepth*)
Returns a list of string representation.

visit (*theVisitor*, *theDepth*)
Take a visitor object make the pre/post calls.

Parameters

- **theVisitor** (cpip.CppCondGraphToHtml.CcgVisitorToHtml, cpip.core.CppCond.CppCondGraphVisitorConditionalLines) – The visitor.
- **theDepth** (int) – Tree depth.

Returns NoneType

class cpip.core.CppCond.**CppCondGraphVisitorBase**
Base class for a CppCondGraph visitor object.

__weakref__
list of weak references to the object (if defined)

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 perfom 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.

_addFileLineState(*file*, *line*, *state*)
Adds the state of the file at the line number

Parameters

- **fileId** – File ID such as its path.
- **line** (`int`) – Line number.
- **state** (`bool`) – Conditional compilation state.

Returns `NoneType`

_lineCondition(*theFile*)
An ordered list of (line_num, boolean).

fileIdS
An unordered list of file IDs.

fileLineCondition
The condition of the file.

Returns `dict({str : [cpip.core.CppCond.LineConditionalInterpretation]})` – File/line condition.

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.

Parameters

- **fileId** (`str`) – File ID such as its path.
- **lineNum** (`int`) – Line number.

Returns `int` – 1 if compiled, 0 otherwise.

visitPost(*theCcgNode*, *theDepth*)
Post visit.

Parameters

- **theCcgNode** (*cpip.core.CppCond.CppCondGraphNode*) – The graph node.
- **theDepth** (int) – The graph depth.

Returns NoneType

visitPre(*theCcgNode*, *theDepth*)
Capture the fileID, line number and state.

Parameters

- **theCcgNode** (*cpip.core.CppCond.CppCondGraphNode*) – The node.
- **theDepth** (int) – Graph depth.

Returns NoneType

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.

__init__(*theList*)
Constructor.

Parameters **theList** (`list([tuple([int, bool])])`) – List of line numbers and compilation state.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

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.

Parameters **lineNum** (int) – Line number.

Returns int – 1 if this line is compiled, 0 if not or -1 if it is ambiguous.

Raises `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`

`cpip.core.CppCond.`**`TOKEN_AND = '&&'`**
AND

`cpip.core.CppCond.`**`TOKEN_JOIN_AND = ' && '`**
AND with seperators.

`cpip.core.CppCond.`**`TOKEN_JOIN_OR = ' || '`**
OR with seperators.

`cpip.core.CppCond.`**`TOKEN_NEGATION = '!'`**
Invert test.

`cpip.core.CppCond.`**`TOKEN_OR = '||'`**
OR

`cpip.core.CppCond.`**`TOKEN_PAD = ' '`**
Pad character

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.

__init__()
Constructor.

__weakref__
list of weak references to the object (if defined)

_prepareMsg(*event*, *msg*, *theLoc*)
Prepares a message.

Parameters

- **event** (`NoneType`) – The event e.g. 'error', if None it is not accumulated.
- **msg** (`str`) – The main message, a string.
- **theLoc** (`cpip.core.FileLocation.FileLineCol([str, int, <class 'int'>]), cpip.core.FileLocation.FileLineCol([str, int, int])`) – The file locator e.g. `FileLocation.FileLineCol`. If present it must have: `(fileId, lineNum colNum)` attributes.

Returns `str` – <insert documentation for return values>

debug(*msg*, *theLoc=None*)
Reports a debug message.

Parameters

- **msg** (`str`) – The main message, a string.
- **theLoc** (`cpip.core.FileLocation.FileLineCol([str, int, <class 'int'>]), cpip.core.FileLocation.FileLineCol([str, int, int])`) – The file locator e.g. `FileLocation.FileLineCol`. If present it must have: `(fileId, lineNum colNum)` attributes.

Returns `NoneType`

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.

__weakref__
list of weak references to the object (if defined)

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, ...}`.

Returns `dict({str : [int]})` – File count map.

fileNameSet
The set of file names seen.

visitGraph(*theFigNode*, *theDepth*, *theLine*)
Hierarchical visitor pattern.

Parameters

- **theFigNode** (*cpip.core.FileIncludeGraph.FileIncludeGraph*) – A *FileIncludeGraph* as a graph node.
- **theDepth** (`int`) – The current depth in the graph as an integer.
- **theLine** (`int`) – The line that is a non-monotonic sibling node ordinal.

Returns `NoneType`

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.

__init__(*theNodeClass*)
Constructor.

Parameters **theNodeClass** (`type`) – The class of the visitor.

Returns `NoneType`

__weakref__
list of weak references to the object (if defined)

_addAncestor()
Add an ancestor node.

Returns `NoneType`

_addChild(*theNode*, *theLine*)
Add a child node.

Parameters

- **theNode** (`cpip.core.FileIncludeGraph.FileIncludeGraph`) – The node.
- **theLine** (`int`) – Line number.

Returns `NoneType`

_addSibling(*theNode*, *theLine*)
Add a sibling.

Parameters

- **theNode** (*cpip.core.FileIncludeGraph.FileIncludeGraph*) – The sibling node.
- **theLine** (int) – The line number.

Returns NoneType

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

Returns int – The depth.

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

Returns *cpip.IncGraphSVG.SVGTreeNodeMain* – Top level node.

visitGraph(*theFigNode*, *depth*, *line*)
Visit the given node.

Parameters

- **theFigNode** (*cpip.core.FileIncludeGraph.FileIncludeGraph*) – The node.
- **depth** (int) – Node depth.
- **line** (int) – Line number.

Returns NoneType

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.

__init__(*theLineNum*)
Constructor.

Parameters **theLineNum** (int) – The line number.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

addChild(*theObj*)
Add the object as a child node.

Parameters **theObj** (*cpip.IncGraphSVG.SVGTreeNodeMain*) – The node.

Returns NoneType

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.

Returns int – Line number.

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.

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

__init__(*theFile*, *theState*, *theCondition*, *theLogic*)

Constructor.

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

Parameters

- **theFile** (`str`) – A file ID (e.g. a path)
- **theState** (`bool`) – A boolean conditional compilation state.
- **theCondition** (`str`) – A conditional compilation condition string e.g. `"a >= b+2"`.
- **theLogic** (`list([str]), str`) – 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.

Returns `NoneType`

__weakref__

list of weak references to the object (if defined)

_getBranches(*theBrancheS*, *aBranch*)

Recursive call to the tree, adds each unique branch in full.

_retFileLine(*theLine*)

Returns '#d' appended to the filename.

_retLatestBranch(*theList*)

Recursive call that returns the branch to the last inserted leaf. theList is modified in-place.

_retLatestBranchDepth(*i*)

Recursive call that returns an integer that is the depth of the latest branch.

_retString (*theIndent*)
Returns an indented string recursively.

Parameters **theIndent** (int) – Size of indent.

Returns str – Indented string.

_splitFileLine (*theStr*)
Splits a file name and line number, the latter is returned as an integer or None if no match.

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.

Parameters

- **visitor** (cpip.core.FileIncludeGraph.FigVisitorFileSet, cpip.core.FileIncludeGraph.FigVisitorTree) – The visitor.
- **depth** (int) – Visitor depth.
- **line** (int) – File line.

Returns NoneType

addBranch (*theFileS*, *theLine*, *theIncFile*, *theState*, *theCondition*, *theLogic*)
Adds a branch to the graph.

Parameters

- **theFileS** (list([str])) – A list of files that form the branch.
- **theLine** (int) – An integer value of the line number of the #include statement of the last named file in theFileS.
- **theIncFile** (str) – The file that is included.
- **theState** (bool) – A boolean that describes the conditional compilation state.
- **theCondition** (str) – The conditional compilation test e.g. '1>0'
- **theLogic** (list([str])) – A string representing how the branch was obtained.

Returns NoneType

Raises *ExceptionFileIncludeGraph* if:

- The branch is zero length.
- **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).
- theLine is a duplicate of an existing line.
- The branch has missing nodes.

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

Returns str – The evaluated conditional compilation string.

condCompState
Returns the recorded conditional compilation state as a boolean.

Returns bool – The state.

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.

Returns `str` – The file name.

findLogic
Returns the findLogic string passed in in the constructor.

Returns `list([str]),str` – The logic string.

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.

Returns `int` – Count of tokens.

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).

Returns `int` – Count of tokens.

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.

Returns `int` – Count of tokens.

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).

Returns `int` – Count of tokens.

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.

Parameters **theBranch** (list([str])) – Branch.

Returns *cpip.core.FileIncludeGraph.FileIncludeGraph* – The include graph.

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 #include's are a separate token counter.

Parameters **theTokCounter** (*cpip.core.PpTokenCount.PpTokenCount*) – Token counter.

Returns NoneType

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

Returns *cpip.core.PpTokenCount.PpTokenCount* – The counter 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.

__init__()
Constructor.

__weakref__
list of weak references to the object (if defined)

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.

Parameters **visitor** (cpip.core.FileIncludeGraph.FigVisitorFileSet, cpip.core.FileIncludeGraph.FigVisitorTree) – The visitor

Returns NoneType

addGraph(*theGraph*)
Add a *FileIncludeGraph* object.

Parameters **theGraph** (*cpip.core.FileIncludeGraph.FileIncludeGraph*) – The graph.

Returns NoneType

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.

Returns *cpip.core.FileIncludeGraph.FileIncludeGraph* – The graph.

numTrees()
Returns the number of *FileIncludeGraph* objects.

Returns int – Number of trees.

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.

__init__(*theFpo*, *theDiag*)
Constructor.

Parameters

- **theFpo** (cpip.core.IncludeHandler.FilePathOrigin([_io.StringIO, str, NoneType, str]), cpip.core.IncludeHandler.FilePathOrigin([_io.TextIOWrapper, str, str, str])) – A FilePathOrigin object that identifies the file.
- **theDiag** (cpip.core.CppDiagnostic.PreprocessDiagnosticStd) – A CppDiagnostic object to give to the PpTokeniser.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

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

Parameters

- **tok** (*cpip.core.PpToken.PpToken*) – Token.
- **isUnCond** (bool) – Unconditional flag.
- **num** (int) – Increment, default 1.

Returns NoneType

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

Parameters **theC** (*cpip.core.PpTokenCount.PpTokenCount*) – Token counter.

Returns NoneType

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.

__init__(*theDiagnostic*)
Constructor, takes a CppDiagnostic object to give to the PpTokeniser.

Parameters **theDiagnostic** (*cpip.core.CppDiagnostic.PreprocessDiagnosticStd*) – The diagnostic for emitting messages.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

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

Returns str – File ID.

depth
Returns the current include depth as an integer.

Returns int – Depth.

fileIncludeGraphRoot
The *FileIncludeGraph.FileIncludeGraphRoot* object.

Returns cpip.core.FileIncludeGraph.FileIncludeGraphRoot – The include graph root.

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

Returns *cpip.core.FileLocation.FileLineCol* – File location.

fileStack
Returns a copy of the stack of file IDs.

Returns list([str]) – File IDs.

finalise()
Finalisation, may raise an ExceptionFileIncludeStack.

Returns NoneType

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

Returns str – File ID such as a path.

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

Parameters

- **theFpo** (cpip.core.IncludeHandler.FilePathOrigin([_io.StringIO, str, NoneType, str]), cpip.core.IncludeHandler.FilePathOrigin([_io.TextIOWrapper, str, str, str])) – A FileLocation.FilePathOrigin object that identifies the file.
- **theLineNum** (NoneType, int) – The integer line number of the file that includes (None if Root).
- **isUncond** (bool) – A boolean that is the conditional compilation state.
- **condStr** (str) – A string of the conditional compilation stack.

- **incLogic** (list([]), list([str])) – A string that describes the find include logic.

Returns NoneType

ppt
Returns the PpTokeniser from the top of the stack.

Returns cpip.core.PpTokeniser.PpTokeniser – The tokeniser.

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

Parameters

- **tok** (*cpip.core.PpToken.PpToken*) – The token.
- **isUnCond** (bool) – Is unconditionally compiled.
- **num** (int) – Increment, default 1.

Returns NoneType

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).

Parameters **theC** (cpip.core.PpTokenCount.PpTokenCount) – The token counter.

Returns NoneType

FileLocation

Various classes for use by the preprocessor for keeping track of the location in a set of files.

This consists of three related classes:

LogicalPhysicalLineMap:

This low-level class maintains and internal theoretical relationship between the logical position in a string and the physical position. These differ when physical characters are replaced by logical ones.

This is not usually used directly by other modules.

FileLocation:

This consists of a stack of one or more LogicalPhysicalLineMap objects each of which represents a phase of translation. This understands what it means to replace a trigraph or encounter a line continuation phrase.

Typically this is used by a cpip.core.PpTokensiser.PpTokeniser.

CppFileLocation:

This consists of a stack of one or more FileLocation objects each of which represents an ITU or included file. Conceptually this is a table of columns (each a FileLocation object) and cells (each a LogicalPhysicalLineMap). The public API gives access to the 'current' LogicalPhysicalLineMap i.e. the top right one in the table. The public API allows pushing (adding a column when a file is #include'd) and popping (removing the last column at the end of #include processing).

Typically this is used by a *cpip.core.PpLexer.PpLexer*.

Using line continuation in *LogicalPhysicalLineMap*:

class FileLocation needs to poke the underlying LogicalPhysicalLineMap in the right way...

This is accomplished by calling from class FileLocation the underlying *LogicalPhysicalLineMap._addToIr()*

This makes calls occur in N pairs.

N = The number of 'n' phrases.

L(n) is the length of the physical line n (0 <= n < N) not including the 'n'

Make N calls to _addIr(...)

NOTE: The use of 1+ and -1* here `_addToIr(theLogicalLine=a, theLogicalCol=1+b, dLine=c, dColumn=-1*d)`

Where:

```
a(n) = The current logical line number (starting at 1), constant for the group
b(n) = Sigma[L(n) for 0...n)]
c(n) = 1
d(n) = L(n)
```

Examples:

```
myPstrS = ['abc\n', '\n', 'd\n', 'ef\n',]
N = 3
L(n) -> (3, 0, 1)
a = 1, c = 1
(b, d)
(3, 3)
(3, 0)
(4, 1)

myPstrS = ['abc\n', 'd\n', '\n', 'ef\n',]
N = 3
L(n) -> (3, 1, 0)
a = 1, c = 1
(b, d)
(3, 3)
(4, 1)
(4, 0)

myPstrS = ['ab\n', 'c\n', 'd\n', 'ef\n',]
N = 3
L(n) -> (2, 1, 1)
a = 1, c = 1
(b, d)
(2, 2)
(3, 1)
(4, 1)
```

The second call of the pair is as follows, this needs to know N so has to be done after all first-of-pair calls:

```
_addToIr(theLogicalLine=d, theLogicalCol=1, dLine=e, dColumn=f)
```

Where:

```
d = n+2 where n is the number of the '\n' in the group starting at 0
e = N-n-1 where N is the total number of '\n' in the group.
f = Length of the last physical line spliced, not including the '\n'
```

Programatically:

```
for n in range(N):
    myLplm._addToIr(theLogicalLine=n+2, theLogicalCol=1, dLine=N-n-1, dColumn=f)
```

In all the examples above f = 2

exception cpip.core.FileLocation.**ExceptionFileLocation**
Simple specialisation of an exception class for the FileLocation classes.

class cpip.core.FileLocation.**FileLine**(*fileId*, *lineNum*)
Simple PODs for file location for anyone that wants to use them

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *fileId*, *lineNum*)
Create new instance of FileLine(fileId, lineNum)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0xa385c0>*, *len=<built-in function len>*)
Make a new FileLine object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new FileLine object replacing specified fields with new values

fileId
Alias for field number 0

lineNum
Alias for field number 1

class cpip.core.FileLocation.**FileLineCol**(*fileId*, *lineNum*, *colNum*)

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *fileId*, *lineNum*, *colNum*)
Create new instance of FileLineCol(fileId, lineNum, colNum)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0xa385c0>*, *len=<built-in function len>*)
Make a new FileLineCol object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new FileLineCol object replacing specified fields with new values

colNum
Alias for field number 2

fileId
Alias for field number 0

lineNum
Alias for field number 1

class cpip.core.FileLocation.**FileLocation**(*theFileName*)
Class that persists the line/column location in a source file. This also handles various passes of the same file for the PpTokeniser.

__init__(*theFileName*)
Initialise with a file name (actually an ID)

NOTE: We do not check for it's existence as we are not allied to the file system (we could get the files from a database instead.

Parameters **theFileName** (NoneType, str) – File ID for example the file path.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

colNum

Returns int – Column number.

fileLineCol()
Return an instance of FileLineCol from the current settings.

Returns cpip.core.FileLocation.FileLineCol([str, int, int]) – File location.

fileName

Returns str – <insert documentation for return values>

incCol(*num=1*)
Increment the column by num. There is no range check on num.

Parameters **num** (int) – Amount to increment, default 1.

Returns NoneType

incLine(*num=1*)
Increment the line by num. There is no range check on num.

Parameters **num** (int) – Amount to increment, default 1.

Returns NoneType

lineCol
Returns the current line and column number as a pair.

lineNum

Returns int – Line number.

lineSpliceCount
The number of line splices in the current splice group.

logicalPhysicalLineMap
Return the current top level LogicalPhysicalLineMap Read Only instance.

logicalToPhysical(*theLline*, *theLcol*)
Returns the physical line and column number for a logical line and column.

Parameters

- **theLline** (int) – Logical line number.
- **theLcol** (int) – Logical column number.

Returns tuple([int, int]) – Physical line and column.

pLineCol
Returns the current physical line and column number.

Returns tuple([int, int]) – Location.

pformatLogicalToPhysical(*theLfile*, *thePfile*)
Given a logical and a physical representation this goes through character by both character, pretty formats the comparison and returns the formatted string.

retColNum()

Returns int – Column number.

retLineNum()

Returns int – Line number.

retPredefinedMacro(*theName*)
Returns the value of __FILE__ or __LINE__. Applies ISO/IEC 14882:1998(E) 16 Predefined macro names [cpp.predefined] note 2 May raise ExceptionFileLocation if theName is something else.

setTrigraph()
Records that a trigraph has be substituted at the current place.

spliceLine(*thePhysicalLine*)
Update the line/column mapping to record a line splice.

startNewPhase()
Starts a new processing phase e.g. a translation phase. This adds a new LogicalPhysicalLineMap() to the stack.

Returns NoneType

substString(*lenPhysical*, *lenLogical*)
Records a string substitution at the current logical location. This does NOT update the current line or column, use update(...) to do that.

update(*theString*)
Increment line and column counters from a string.

Parameters **theString** (str) – Source code.

Returns NoneType

class cpip.core.FileLocation.**LogicalPhysicalLineMap**
Class that can map logical positions (i.e. after text substitution) back to the original physical line columns. The effect of various substitutions is as follows:

Phase 1: Trigraph replacement - logical is same or smaller than physical. Phase 1: Mapping non lex.charset to universal-character-name - logical is larger than physical. Phase 2: Line splicing - if done logical is smaller that physical. Phase 3: Digraph replacement - logical is same or smaller that physical.

__weakref__
list of weak references to the object (if defined)

_addToIr(*theLogicalLine*, *theLogicalCol*, *dLine*, *dColumn*)
Adds, or updates a record to the internal representation.

offsetAbsolute(*theLineCol*)
Given a pair of integers that represent line/column starting at zero this returns a tuple pair of the absolute line/column.

offsetRelative(*theLineCol*)
Given a pair of integers that represent line/column starting at START_LINE, START_COLUMN this returns a tuple pair of the relative line/column i.e. starting at (0, 0).

pLineCol(*lLine*, *lCol*)
Returns the (physical line number, physical column number) from a logical line and logical column.

Parameters

- **theLline** (int) – Logical line number.
- **theLcol** (int) – Logical column number.

Returns tuple([int, int]) – Physical line and column.

substString(*theLogicalLine*, *theLogicalCol*, *lenPhysical*, *lenLogical*)
Records a string substitution.

cpip.core.FileLocation.**START_LINE = 1**
See ISO/IEC 14882:1998(E) 16.4 Line control [cpp.line] note 2 Means that lines start at 1. We also take the first column to be number 1.

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 is the usr includes.
J is the sys includes.
Size of I Size of J   #include <...>      #include "..."
0         0           None                CP

0         >0          SYSTEMINCLUDEdirs   CP, SYSTEMINCLUDEdirs

>0        0           USERINCLUDEdirs     CP, USERINCLUDEdirs

>0        >0          SYSTEMINCLUDEdirs,  CP, USERINCLUDEdirs,
                      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 = {None: 'Not found', 'comp': 'Compiler specific directories', 'CP': 'Current Place', 'usr': '
Codes for the results of a search for an include

__init__(*theUsrDirs*, *theSysDirs*)
Constructor.

Parameters

- **theUsrDirs** (list([str])) – List of search directories for user includes.

- **theSysDirs** (list([str])) – List of search directories for system includes.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

_currentPlaceFromFile(*theFilePath*)
Helper method that returns the enclosing directory of the file as the current place.

Parameters **theFilePath** (str) – File path.

Returns str – Directory path.

_fixDirsep(*theCharSeq*)
Returns a character sequence with the allowable directory seperator replaced by that the OS will recognise.

Parameters **theCharSeq** (str) – The character sequence.

Returns str – OS compatible character sequence.

_includeHcharseq(*theHstr*, *include_next=False*)
Return the file location of a #include <...> as a FilePathOrigin object or None on failure.

If not None this also records the CP for the file.

Parameters

- **theHstr** (str) – The include string.
- **include_next** (bool) – Use GGC extension #include-next.

Returns cpip.core.IncludeHandler.FilePathOrigin([_io.TextIOWrapper, str, str, str]) – File path of the included file.

_includeQcharseq(*theQstr*, *include_next=False*)
Return the file location of a #include "..." as a FilePathOrigin object or None on failure.

If not None return value this also records the 'current place' (CP) for the file.

Parameters

- **theQstr** (str) – The include string.
- **include_next** (bool) – Use GGC extension #include-next.

Returns cpip.core.IncludeHandler.FilePathOrigin([_io.TextIOWrapper, str, str, str]) – File path of the included file.

_searchFile(*theCharSeq*, *theSearchPath*)
Given an HcharSeq/Qcharseq and a searchpath this should return a class FilePathOrigin or None

canInclude()
Returns True if the last include succeeded.

Returns bool – 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.

Returns NoneType

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.

Parameters **theFpo** (cpip.core.IncludeHandler.FilePathOrigin([_io.StringIO, str, NoneType, str]), cpip.core.IncludeHandler.FilePathOrigin([_io.TextIOWrapper, str, str, str])) – The FilePathOrigin to push onto the include stack.

Returns NoneType

cpStackSize
Returns the size of the current stack of current places.

Returns int – Stack size.

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

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

Returns NoneType

finalise()
Finalise at the end of the translation unit. Might raise a ExceptionCppInclude.

Returns NoneType

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']
```

Item [0] is the invocation. 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 [-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.

Returns list([]),list([str]) – How the file was found.

includeHeaderName (*theStr*)
Return the file location of a `#include header-name` where the `header-name` is a `pp-token` (a *cpip.core.PpToken.PpToken*) with the contents either a `<h-char-sequence>` or a `"q-char-sequence"` (including delimiters).

If not None return value this also records the 'current place' (CP) for the file.

Parameters **theStr** (`str`) – Header name with delimiters.

Returns `cpip.core.IncludeHandler.FilePathOrigin([_io.TextIOWrapper, str, str, str])` – File path of the included 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.

Returns `bool` – False on failure.

class `cpip.core.IncludeHandler.`**CppIncludeStdOs** (*theUsrDirs*, *theSysDirs*)
This implements _searchFile() based on an OS file system call.

_searchFile (*theCharSeq*, *theSearchPath*)
Given an HcharSeq/Qcharseq and a searchpath this tries the file system for the file and returns a FilePathOrigin object or None on failure.

Parameters

- **theCharSeq** (`str`) – Character sequence.
- **theSearchPath** (`str`) – Search path.

Returns `NoneType,cpip.core.IncludeHandler.FilePathOrigin([_io.TextIOWrapper, str, str, NoneType])` – File found or `None`.

Raises `FileNotFoundError`

initialTu (*theTuPath*)
Given an path as a string this returns the class FilePathOrigin or None for the initial translation unit.

Parameters **theTuPath** (`str`) – File path.

Returns `cpip.core.IncludeHandler.FilePathOrigin([_io.TextIOWrapper, str, str, str])` – The file path origin.

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.

__init__(*theUsrDirs*, *theSysDirs*, *theInitialTuContent*, *theFilePathToContent*)

Acts like a IncludeHandler but looks up in theFilePathToContent map that is a {path_string : content_string, ...}. This will be used to simulate resolving a #include statement.

_searchFile(*theCharSeq*, *theSearchPath*)

Given an HcharSeq/Qcharseq and a searchpath this tries the file system for the file.

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

__getnewargs__()

Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *fileObj*, *filePath*, *currentPlace*, *origin*)

Create new instance of FilePathOrigin(fileObj, filePath, currentPlace, origin)

__repr__()

Return a nicely formatted representation string

_asdict()

Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0xa385c0>*, *len=<built-in function len>*)

Make a new FilePathOrigin object from a sequence or iterable

_replace(*_self*, ***kwds*)

Return a new FilePathOrigin object replacing specified fields with new values

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*)

Tokensises a file like object.

__init__(*theFileObj=None*, *theFileId=None*, *theDiagnostic=None*)

Constructor.

Parameters

- **theFileObj** (_io.TextIOWrapper) – File object.
- **theFileId** (str) – File ID such as the path.
- **theDiagnostic** (NoneType) – A diagnostic for processing messages.

Returns NoneType

_translatePhase_1()

Performs translation phase one. Note: We do not (yet) support universal-character-name conversion so this only does trigraphs.

Returns NoneType

Raises IndexError

_translatePhase_2()

Performs translation phase two. This does line continuation markers Note: We do not (yet) test for accidental UCN creation.

Returns NoneType

Raises IndexError

_translatePhase_3()

Performs translation phase three. Replaces comments and decomposes stream into preprocessing tokens.

Returns NoneType

Raises IndexError

genTokensKeywordPpDirective()

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

Returns tuple([str, str]) – Token value, token type.

Raises StopIteration

multiPassString

Returns *cpip.util.MultiPassString.MultiPassString* – The multi-pass string object.

translatePhases123()

Translate phase 1, 2, 3.

Returns NoneType

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.**ExceptionMacroReplacementPredefinedRedefintion**
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_\n_terminated, ...}` For example:

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

Each identifier must be in `STD_PREDEFINED_NAMES`

_MacroEnv__define(*ppD*)
Takes a *cpip.core.PpDefine.PpDefine* object and adds it to the map of objects. Does **not** check if it is a redefinition of a predefined macro. Does check if it is a valid redefinition.

On success it returns the identifier of the macro.

The insertion is stable i.e. a valid re-definition does not replace the existing definition so that the definition file, line and reference count are preserved.

Parameters **ppD** (*cpip.core.PpDefine.PpDefine*) – Macro definition.

Returns `str` – Macro identifier.

_MacroEnv__setString(*theStr*)
Takes a string `'identifier replacement\n'` and sets the macro map. This uses `__define()` so only a redefinition exception is raised.

Parameters **theStr** (`str`) – Replacement string.

Returns `NoneType`

__init__(*enableTrace=False*, *stdPredefMacros=None*)
Constructor.

A 'reference' is defined as: replacement or if defined. So:

- Count set to zero on *define()*
- **Increment on:**
  - *isDefined()*
  - *defined()*
  - *_expand()*

Parameters

- **enableTrace** (bool) – If True allows calls to *_debugTokenStream()* that may or may not produce log output (depending on logging level).
- **stdPredefMacros** (dict({str : [str]})) – If present should be a dictionary of: {identifier : replacement_string_\n_terminated, ...}

  These identifiers are not permitted to be redefined.

  This also increments the count is the number of times that the identifier has been referenced in the lifetime of me.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

_assertDefineMapIntegrity()
Returns True if dynamic tests on self._defineMap and self._expandedSet pass. i.e. every entry in self._expandedSet must be in self._defineMap.keys().

Returns bool – False on failure.

_debugTokenStream(*thePrefix*, *theArg=''*)
Writes to logging.debug() an interpretation of the token stream provided by theList. It will be preceded by the debugMarker value (if set) and that will always be cleared.

_expand(*theTtt*, *theGen*, *theFileLineCol*)
Recursive call to expand macro symbols.

theFileLineCol Is a FileLocation.FileLineCol object.

_hasExpanded(*theTtt*)
Returns True if theTok represents a macro name that has already been expanded.

_reset()
Initialises the dynamic values.

Returns NoneType

_staticMacroDependencies(*theIdentifier*)
Returns the immediate dependencies as a list of strings for a macro identified by the string.

Parameters **theIdentifier** (str) – Macro name.

Returns list([]),list([str]) – List of macro 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*

Returns *cpip.util.Tree.DuplexAdjacencyList* – The dependencies.

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.

Parameters

- **theGen** (`generator`) – Token generator.
- **theFile** (`str`) – File identifier such as the path.
- **theLine** (`int`) – Line number.

Returns `str` – Macro name.

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.*

Parameters

- **theTtt** (*cpip.core.PpToken.PpToken*) – The token.
- **flagInvert** (`bool`) – Invert the test.
- **theFileLineCol** (`cpip.core.FileLocation.FileLineCol([str, int, int])`) – File location.

Returns *cpip.core.PpToken.PpToken* – A token that is either 0 or 1.

Raises `KeyError`

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.

Parameters **theIdent** (`NoneType`) – Optionally the macro name.

Returns *cpip.core.PpDefine.PpDefine* – Yields macro definitions.

genMacrosOutOfScope(*theIdent=None*)
Generates PpDefine objects encountered during my existence but then undefined in the order of undefinition.

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

Parameters **theIdent** (NoneType) – Optionally the macro name.

Returns *cpip.core.PpDefine.PpDefine* – Yields macro definitions.

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.

Parameters **theIdentifier** (str) – Macro name.

Returns bool – True if the macro is currently defined.

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.

Parameters **theIdentifier** (str) – Macro name.

Returns *cpip.core.PpDefine.PpDefine* – The macro.

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 ofself.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.

Returns list([str]) – List of macro names.

macroNotDefinedDependencyReferences(*theIdentifier*)
Returns an ordered list of `cpip.core.FileLocation.FileLineCol`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.

Parameters **theIdentifier** (`str`) – Macro name.

Returns `list([cpip.core.FileLocation.FileLineCol([str, int, int])])` – List of locations.

macros()
Returns and unsorted list of strings of current macro identifiers.

Returns `list([str])` – Macro names.

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`.

Parameters **theTtt** (*cpip.core.PpToken.PpToken*) – The token.

Returns `bool` – True if this might be replaceable.

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, redefintion, replacement and rescaning 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 identifers 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/IECISO/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 placemarker 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

_PpDefine__addTokenAndTypeToReplacementList(*theTtt*)
Adds a token and a token type to the replacement list. Runs of whitespace tokens are concatenated.

Parameters **theTtt** (*cpip.core.PpToken.PpToken*) – Token.

Returns NoneType

_PpDefine__logWarningHashHashHash()
Emit a warning to the log that # and ## are dangerous together.

__init__(*theTokGen*, *theFileId*, *theLine*)
Takes a preprocess token generator and creates a macro. The generator (e.g. a instance of PpTokeniser.next()) can generate pp-tokens that appear after the start of the #define directive from the first non-whitespace token onwards i.e. this __init__ will, itself, consume leading whitespace.

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 replacement-list is stored as a list of preprocessor tokens. The identifier-list is stored as a list of names. Leading and trailing whitespace in the replacement list is removed to facilitate redefinition comparison.

Parameters

- **theTokGen** (generator) – Token generator.
- **theFileId** (str) – File ID such as the path.
- **theLine** (int) – theLine is a positive integer that represents the line in theFile that the #define statement occurred. This must be >= 1

Returns NoneType

__weakref__
list of weak references to the object (if defined)

_appendArgIdentifier(*theTok*, *theGenTok*)
Appends the token text to the argument identifier list.

_appendToReplacementList(*theGenTok*)
Takes a token sequence up to a newline and assign it to the replacement-list. Leading and trailing whitespace is ignored.

TODO: Set setPrevWs flag where necessary.

Parameters **theGenTok** (generator) – Token generator.

Returns NoneType

_consumeAndRaise(*theGen*, *theException*)
Consumes all tokens up to and including the next newline then raises an exception. This is commonly used to get rid of bad token streams but allow the caller to catch the exception, report the error and continue.

_consumeNewline(*theGen*)
Consumes all tokens up to and including the next newline.

_cppStringize(*theArgTokens*)
Applies the '#' operator to function style macros ISO/IEC ISO/IEC 14882:1998(E) 16.3.2 The # operator [cpp.stringize]

_ctorFunctionMacro(*theGenTok*)
Construct function type macros. [[identifier-list,] ,')', replacement-list, new-line, ...]

The identifier-list is not specified in the specification but there seems to be some disparity between the standards and cpp.exe. The relevant bits of the standards [C: ISO/IEC 9899:1999(E) 6.10.3-10 and -11 and C++: ISO/IEC 14882:1998(E) 16.3-9 (C++)] appear, to me, to suggest that left and right parenthesis are allowed in the identifier-list and that (,) is ignored. But cpp.exe will not accept that.

Playing with cpp -E it seems that it is a comma separated list where whitespace is ignored, nothing else is allowed. See unit tests testInitFunction_70(), 71 and 72. cpp.exe also is not so strict when it comes the the above sections. For example in this:

..code_block: c

    #define FOO(a,b,c) a+b+c FOO (1,(2),3)

The whitespace between FOO and LPAREN is ignored and the replacement occurs.

_functionLikeReplacement(*theArgMap*)

Returns the replacement list where if a token is encountered that is a key in the map then the value in the map is inserted into the replacement list.

`theArgMap` is of the form returned by _retReplacementMap(). This also handles the `'#'` token i.e. [cpp.stringize] and `'##'` token i.e. [cpp.concat].

Returns a list of pairs i.e. [(token, token_type), ...]

TODO: Accidental token pasting #define f(x) =x= f(=) We want '= = =' not '==='.

_isPlacemarker(*theTok*)

Returns True if the Token represents a PLACEMARKER token. This is the correct comparison operator can be used if self.PLACEMARKER is defined as None.

_nextNonWsOrNewline(*theGen*)

Returns the next non-whitespace token or whitespace that contains a newline.

Parameters **theGen** (`generator`) – Token generator.

Returns *cpip.core.PpToken.PpToken* – The next non-whitespace token or whitespace that contains a newline.

_objectLikeReplacement()

Returns the replacement list for an object like macro. This handles the `##` token i.e. [cpp.concat].

Returns a list of pairs i.e. `[(token, token_type), ...]`

_retReplacementMap(*theArgs*)

Given a list of lists of (token, type) this returns a map of: `{identifier : [replacement_token and token types, ...], ...}`

For example for:

```
#define FOO(c,b,a) a+b+c
FOO(1+7,2,3)
```

i.e theArgs is (types are shown as text for clarity, in practice they would be enumerated):

```
[
    [
        PpToken.PpToken('1', 'pp-number'),
        PpToken.PpToken('+', 'preprocessing-op-or-punc'),
        PpToken.PpToken('7', 'pp-number')
    ],
    [
        PpToken.PpToken('2', 'pp-number'),
    ],
    [
        PpToken.PpToken('3', 'pp-number'),
    ],
]
```

Map would be:

```
{
    'a' : [
            PpToken.PpToken('3', 'pp-number'),
        ],
    'b' : [
            PpToken.PpToken('2', 'pp-number'),
        ],
    'c' : [
            PpToken.PpToken('1', 'pp-number'),
            PpToken.PpToken('+', 'preprocessing-op-or-punc'),
            PpToken.PpToken('7', 'pp-number')
        ],
}
```

Note that values that are placemarker tokens are PpDefine.PLACEMARKER.

For example:

```
#define FOO(a,b,c) a+b+c
FOO(,2,)
```

Generates:

```
{
    'a' : PpDefine.PLACEMARKER,
    'b' : [
            ('2', 'pp-number'),
        ]
    'c' : PpDefine.PLACEMARKER,
}
```

PERF: See TODO below.

TODO: Return a map of identifiers to indexes in the supplied argument as this will save making a copy of the argument tokens?

So:

```
#define FOO(c,b,a) a+b+c
FOO(1+7,2,3)
```

Would return a map of:

```
{
    'a' : 2,
    'b' : 1,
    'c' : 0,
}
```

And use index -1 for a placemarker token???:

```
#define FOO(a,b,c) a+b+c
FOO(,2,)
```

Generates:

```
{
    'a' : -1,
    'b' : 1
    'c' : -1,
}
```

_retToken (*theGen*)
Returns the next token object and increments the IR.

Parameters **theGen** (generator) – Token generator.

Returns *cpip.core.PpToken.PpToken* – The next token.

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.

Returns NoneType

Raises AssertionError

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.

Returns str – File ID, path for example.

identifier
The macro identifier i.e. the name as a string.

Returns `str` – Macro name.

incRefCount(*theFileLineCol=None*)

Increment the reference count. Typically callers do this when replacement is certain of in the event of definition testing

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.

Parameters **theFileLineCol** (`cpip.core.FileLocation.FileLineCol([str, int, int])`) – File location.

Returns `NoneType`

isCurrentlyDefined

Returns True if the current instance is a valid definition i.e. it has not been `#undef`'d.

Returns `bool` – Has valid definition.

isObjectTypeMacro

Returns `bool` – 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 equivelent.

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.

Returns `int` – Line number.

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.

Returns `int` – Reference count.

refFileLineColS
Returns the list of FileLineCol objects where this macro was referenced.

Returns `list([]),list([cpip.core.FileLocation.FileLineCol([str, int, int])])` – Places the 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*

Returns `list([]),list([cpip.core.PpToken.PpToken])` – Tokens.

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 argunments:

```
[
    [
        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'),
    ],
]
```

Placemarker 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.

Returns `str` – Macro declaration..

strReplacements()
Returns the replacements tokens with minimised whitespace as a string.

Returns `str` – The replacements tokens with minimised whitespace as a string.

tokenCounter
The PpTokenCount object that counts tokens that have been consumed from the input.

Returns *cpip.core.PpTokenCount.PpTokenCount* – Token count.

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.**ExceptionPpLexerNestedInclueLimit**
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', 'unde**
Allowable preprocessing directives

class cpip.core.PpLexer.**PpLexer**(*tuFileId*, *includeHandler*, *preIncFiles=None*, *diagnostic=None*, *pragmaHandler=None*, *stdPredefMacros=None*, *autoDefineDateTime=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*.

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

Conditianlity 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

__init__(*tuFileId*, *includeHandler*, *preIncFiles=None*, *diagnostic=None*, *pragmaHandler=None*, *stdPredefMacros=None*, *autoDefineDateTime=True*, *gccExtensions=False*, *annotateLineFile=False*)

Constructor.

Parameters

- **tuFileId** (str) – 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** (cpip.core.IncludeHandler.CppIncludeStdOs) – A handler to file #includ'd files typically a IncludeHandler.IncludeHandlerStd. This might have user and system include path information and a means of resolving file references.
- **preIncFiles** (list([_io.StringIO])) – 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** (NoneType) – A diagnostic object, defaults to a *CppDiagnostic.PreprocessDiagnosticStd*.
- **pragmaHandler** (NoneType) – A handler for #pragma statements.

  This must have the attribute replaceTokens is to be implemented, if True then the tokens stream will be 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** (dict({})) – 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** (bool) – If True then the macros __DATE__ and __TIME__ will be automatically updated to current locale date/time. Mostly this is used for testing.
- **gccExtensions** (bool) – Support GCC extensions. Currently just #include_next is supported.
- **annotateLineFile** (bool) – 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
```

Returns NoneType

__weakref__
list of weak references to the object (if defined)

_appendTokenMergingWhitespace(*theList*, *theToken*)
Adds a token to the list merging whitespace if possible.

Parameters

- **theList** (list([]), list([cpip.core.PpToken.PpToken])) – List of tokens.
- **theToken** (cpip.core.PpToken.PpToken) – The token to append, if whitespace and the last token on the list is also whitespace then this token will be merged into the last token on the list.

Returns NoneType

_countNonWsTokens(*theTokS*)
Returns the integer count of non-whitespace tokens in the given list.

Parameters **theTokS** (list([cpip.core.PpToken.PpToken])) – List of tokens.

Returns int – Count of non-whitespace tokens.

_cppDefine(*theGen*, *theFlc*)
Handles a define directive.

Parameters

- **theGen** (generator) – Token generator.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File line and column numbers.

Returns *cpip.core.PpToken.PpToken* – Yields resulting tokens.

_cppElif (*theGen*, *theFlc*)
Handles a elif directive.

Parameters

- **theGen** (generator) – Token generator.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File, line, column.

Returns NoneType,cpip.core.PpToken.PpToken – The replacement token, a single whitespace.

_cppElse (*theGen*, *theFlc*)
Handles a else directive.

Parameters

- **theGen** (generator) – Token generator.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File, line, column.

Returns NoneType,cpip.core.PpToken.PpToken – The replacement token, a single whitespace.

_cppEndif (*theGen*, *theFlc*)
Handles a endif directive.

Parameters

- **theGen** (generator) – Token generator.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File, line, column.

Returns NoneType,cpip.core.PpToken.PpToken – The replacement token, a single whitespace.

_cppError (*theGen*, *theFlc*)
Handles a error directive.

_cppIf (*theGen*, *theFlc*)
Handles a if directive.

Parameters

- **theGen** (generator) – Token generator.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File, line, column.

Returns NoneType,cpip.core.PpToken.PpToken – The replacement token, a single whitespace.

_cppIfdef (*theGen*, *theFlc*)
Handles a Ifdef directive.

Parameters

- **theGen** (generator) – Token generator.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File, line, column.

Returns NoneType,cpip.core.PpToken.PpToken – The replacement token, a single whitespace.

_cppIfndef(*theGen*, *theFlc*)
Handles a ifndef directive.

Parameters

- **theGen** (generator) – Token generator.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File, line, column.

Returns NoneType,cpip.core.PpToken.PpToken – The replacement token, a single whitespace.

_cppInclude(*theGen*, *theFlc*)
Handles an #include directive. This handles:

```
# include <h-char-sequence> new-line
# include "q-char-sequence" new-line
```

This gathers a list of PpTokens up to, and including, a newline with macro replacement. Then it reinterprets the list using *cpip.core.PpTokeniser.PpTokeniser.reduceToksToHeaderName()* to cast tokens to possible #include <header-name> token.

Finally we try and resolve that to a 'file' that can be included.

FWIW cpp.exe does not explore #include statements when they are conditional so will not error on unreachable files if they are conditionally included.

Parameters

- **theGen** (generator) – Token generator.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File, line, column.

Returns generator – <insert documentation for return values>

_cppIncludeGeneric(*theGen*, *theFlc*, *theFileIncludeFunction*)
Handles the target of an #include or #include_next directive. theFileIncludeFunction is the function to call to resolve the target to an actual file.

Parameters

- **theGen** (generator) – Token generator.
- **theFlc** (cpip.core.FileLocation.FileLineCol([str, int, int])) – File, line, column.
- **theFileIncludeFunction** (method) – Function to process include directive.

Returns NoneType,cpip.core.PpToken.PpToken – Yields tokens.

Raises StopIteration

_cppIncludeNext(*theGen*, *theFlc*)
Handles an #include_next GCC extension. This behaves in a very similar fashion to self._cppInclude but calls includeNextHeaderName() on the include handler

_cppIncludeReportError(*theMsg=None*)
Reports a consistent error message when #indlude is not processed and consumes all tokens up to and including the next newline.

_cppLine(*theGen*, *theFlc*)
Handles a line directive. This also handles *ISO/IEC 9899:1999 (E) 6.10.4 Line control* In particular 6.10.4-4 where the form is:

```
# line digit-sequence "s-char-sequenceopt" new-line
```

digit-sequence is a a token type pp-number.

The s-char-sequenceopt is a token type 'string-literal', this will have the double quote delimeters and may have a 'L' prefix. for example L"abc".

_cppPragma(*theGen*, *theFlc*)
Handles a pragma directive. *ISO/IEC 9899:1999 (E) 6.10.6 Pragma directive*

Semantics:

1 A preprocessing directive of the form:: # pragma pp-tokensopt new-line

where the preprocessing token STDC does not immediately follow pragma in the directive (prior to any macro replacement)146) causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such pragma that is not recognized by the implementation is ignored.

Footnote 146: An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where STDC immediately follows pragma). If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; an implementation is permitted to behave as if it were the standard pragma, but is not required to.

_cppUndef(*theGen*, *theFlc*)
Handles a undef directive.

_cppWarning(*theGen*, *theFlc*)
Handles a warning directive. Not in the standard but we support it.

_diagnosticDebugMessage(*theM*)
Sends a message to the diagnostic object.

Parameters **theM** (`str`) – The message

Returns `NoneType`

_genPpTokensRecursive(*theGen*)
Given a token generator this applies the lexical rules and generates tokens. This means handling preprocessor directives and macro replacement.

With `#include`'d files this become recursive.

Parameters **theGen** (`generator`) – Token generator.

Returns *cpip.core.PpToken.PpToken* – Yields tokens.

Raises `StopIteration`

_genPreIncludeTokens()
Reads all the pre-include files and loads the macro environment.

Returns `NoneType`

Raises `AttributeError, StopIteration`

_lineFileAnnotation(*flags*)
Returns a list of PpTokens that represent the line number and file name. 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
```

Trailing numbers are described here: https://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html

`'1'` - This indicates the start of a new file.

`'2'` - This indicates returning to a file (after having included another file).

`'3'` - This indicates that the following text comes from a system header file, so certain warnings should be suppressed.

`'4'` - This indicates that the following text should be treated as being wrapped in an implicit `extern "C"` block.

We don't support '4'

_nextNonWsOrNewline(*theGen*, *theDiscardList=None*)
Returns the next non-whitespace token or whitespace that contains a newline.XXX

Parameters

- **theGen** (`generator`) – Token generator.
- **theDiscardList** (`list([cpip.core.PpToken.PpToken])`) – If theDiscardList is non-None intermediate tokens will be appended to it.

Returns *cpip.core.PpToken.PpToken* – Next non-whitespace token or whitespace that contains a newline.

_pptPop()
End a #included file.

Returns `NoneType`

_pptPostPop()
Called immediately after _pptPop() this, optionally, returns a list of PpToken's that can be yielded.

Returns `list([])` – Tokens to yield.

_pptPostPush()
Called immediately after _pptPush() this, optionally, returns a list of PpToken's that can be yielded.

Returns `list([])` – Tokens to yield.

_pptPush(*theFpo*)
This takes a *cpip.core.IncludeHandler.FilePathOrigin* object and pushes it onto the FileIncludeStack which creates a PpTokneiser object on the stack.

This returns that PpTokeniser generator function.

Parameters **theFpo** (`cpip.core.IncludeHandler.FilePathOrigin([_io.StringIO, str, NoneType, str]), cpip.core.IncludeHandler.FilePathOrigin([_io.TextIOWrapper, str, str, str])`) – FilePathOrigin.

Returns `generator` – The *cpip.core.PpTokeniser.PpTokeniser* object.

_processCppDirective(*theTtt*, *theGen*)
Processes a token as a CPP directive. *ISO/IEC ISO/IEC 14882:1998(E) 16 Preprocessing directives [cpp]* This consumes tokens and generates others.

Parameters

- **theTtt** (cpip.core.PpToken.PpToken) – Current token.
- **theGen** (generator) – Token generator.

Returns *cpip.core.PpToken.PpToken* – Yields resulting tokens.

Raises StopIteration

_reportSpuriousTokens (*theCmd*)

Reports the presence of spurious tokens in things like: #else spurious 1) tokens ... Used by #else and #endif which expect no semantically significant tokens to follow them. Typical cpp.exe behaviour: cpp.exe: <stdin>:3:7: warning: extra tokens at end of #else directive

_retDefineAndTokens (*theGen*)

Returns 1 or 0 if a macro is defined and the literal tokens as as string..

Parameters **theGen** (generator) – Token generator.

Returns tuple([int, str]) – (bool, literal tokens)

_retDefinedSubstitution (*theGen*)

Returns a list of tokens from the supplied argument with defined... and !defined... handled appropriately and other tokens expanded where appropriate.

This is used by #if, #elif.

Reporting conditional state, for example:

```
#define F(a) a % 2
#define X 5

What to say?     This?         Or?           Or?              Or?
#if F(X) == 1    F(X) == 1     F(5) == 1     (5 % 2) == 1     1 == 1
...
#else            !F(X) == 1    !F(5) == 1    !(5 % 2) == 1    !(1 == 1)
...
#endif
```

The current implementation takes the first as most useful: "F(X) == 1". This means capturing the original token stream as well as the (possibly replaced) evaluated token stream.

TODO: There is an issue here is with poorly specified #if/#elif statements For example:

```
#if deeeefined SPAM
cpp.exe: <stdin>:1:7: missing binary operator before token "SPAM"

#if 1 SPAM
cpp.exe: <stdin>:1:7: missing binary operator before token "SPAM"
```

Parameters **theGen** (generator) – Token generator.

Returns tuple([list([cpip.core.PpToken.PpToken]), list([cpip.core.PpToken.PpToken])]) – (Replacment tokens, raw tokens).

_retHeaderName (*theGen*)

This returns the first PpToken of type header-name it finds up to a newline token or None if none found. It handles:

```
# include <h-char-sequence> new-line
# include "q-char-sequence" new-line
```

This gathers a list of PpTokens up to, and including, a newline with macro replacement. Then it reinterprets the list using *cpip.core.PpTokeniser.PpTokeniser.reduceToksToHeaderName()* to cast tokens to possible `#include` header-name token.

Parameters **theGen** (`generator`) – Token generator.

Returns *cpip.core.PpToken.PpToken* – First token of header name.

_retIfEvalAndTokens(*theGen*)
Returns `(bool | None, tokenStr)` from processing a #if or #elif conditional statement. This also handles defined... and !defined...

bool - True/False based on the evaluation of the constant expression. This will be None on evaluation failure.

tokenStr - A string of raw (original) PpTokens that made up the constant expression.

Parameters **theGen** (`generator`) – Token generator.

Returns `tuple([int, str])` – (bool, literal tokens)

_retListReplacedTokens(*theTokS*)
Takes a list of PpToken objects and returns a list of PpToken objects where macros are replaced in the current environment where possible. TODO: get pragma to use this.

_tokensToEol(*theGen*, *macroReplace*)
Returns a list of PpToken objects from a generator up to and including the first token that has a newline.

Parameters

- **theGen** (`generator`) – Token generator.
- **macroReplace** (`bool`) – If `macroReplace` is `True` then macros are replaced with the current environment.

Returns `list([cpip.core.PpToken.PpToken])` – List of consumed tokens.

colNum
Returns the current column number as an integer during processing.

condCompGraph
The conditional compilation graph as a *cpip.core.CppCond.CppCondGraph* object.

Returns *cpip.core.CppCond.CppCondGraph* – The conditional compilation graph.

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 *cpip.core.FileIncludeGraph.FileIncludeGraphRoot* object.

Returns *cpip.core.FileIncludeGraph.FileIncludeGraphRoot* – The file include graph root.

fileLineCol
Returns a FileLineCol object or None.

Returns *cpip.core.FileLocation.FileLineCol* – File location as `(str, int, int)`.

fileName
Returns the current file name during processing.

Returns `str` – File name.

fileStack
Returns the file stack.

Returns `list([str])` – The file stack.

finalise()
Finalisation, may raise any Exception.

Returns `NoneType`

Raises `Exception` - Any exception.

includeDepth
Returns the integer depth of the include stack.

lineNum
Returns the current line number as an integer during processing or None.

Returns `NoneType,int` – Line number.

macroEnvironment
The current Macro environment as a *cpip.core.MacroEnv.MacroEnv* object.

> **Caution:** Write to this at your own risk. Your write might be ignored or cause undefined behaviour.

Returns *cpip.core.MacroEnv.MacroEnv* – The macro environment.

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)*.

Parameters

- **incWs** (`bool`) – If `True` then also include all whitespace tokens.
- **minWs** (`bool`) – If `True` then whitespace runs will be minimised to a single space or, if newline is in the whitespace run, a single newline.
- **condLevel** (`int`) – If != 0 then conditionally compiled tokens will be yielded and they will have 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():).

Returns *cpip.core.PpToken.PpToken* – Yields tokens.

Raises StopIteration

tuFileId
Returns the user supplied ID of the translation unit.

Returns str – Translation unit ID.

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-liter**
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 = {'character-literal': 3, 'non-whitespace': 6, 'whitespace': 7, 'pp-number': 2, 'concat': 8**
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 = {'||': ' or ', 'true': 'True', '&&': ' and ', '/': '//', 'false': 'False'}
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 an 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 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

Returns `str` – String for `eval()`.

Raises `KeyError`

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.

Returns `bool` – Flag.

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()

Returns `bool` – Flag.

isIdentifier()

Returns `bool` – `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.

Returns `bool` – Flag.

isWs()

Returns `bool` – `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*

Returns `NoneType`

Raises *ExceptionCpipTokenIllegalOperation* if I am not a whitespace token.

setIsCond()
Sets self._isCond to be True.

Returns `NoneType`

setIsReplacement(*val*)
Sets the flag that records that this token is the result of macro replacement.

Parameters **val** (`bool`) – Flag.

Returns `NoneType`

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 ' '

Returns `NoneType`

Raises *ExceptionCpipTokenIllegalOperation* if I am not a whitespace token.

subst(*t*, *tt*)
Substitutes token value and type.

t
Returns the token as a string.

Returns `str` – Token.

tokEnumToktype
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.

Returns `str` – Token type.

`cpip.core.PpToken.`**tokensStr**(*theTokens*, *shortForm=True*)
Given a list of tokens this returns them as a string.

Parameters

- **theTokens** (`list([cpip.core.PpToken.PpToken])`) – List of tokens.
- **shortForm** (`bool`) – 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) | ...`

Returns `str` – Tokens as a string.

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.

Parameters **other** (*cpip.core.PpTokenCount.PpTokenCount*) – Other object.

Returns `cpip.core.PpTokenCount.PpTokenCount` – Self.

__weakref__
list of weak references to the object (if defined)

_inc(*tokStr*, *isUnCond*, *num*)
Increment the count. tok is a string, isUnCond is a boolean that is True if this is not conditionally compiled. num is the number of tokens to increment.

Parameters

- **tokStr** (`str`) – The token type.
- **isUnCond** (`bool`) – True if this is not conditionally compiled.
- **num** (`int`) – Number of tokens.

Returns `NoneType`

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.

Parameters

- **tok** (*cpip.core.PpToken.PpToken*) – The token.
- **isUnCond** (bool) – True if this is not conditionally compiled.
- **num** (int) – Number of tokens. Default 1.

Returns NoneType

tokenCount(*theType*, *isAll*)
Returns the token count including whitespace tokens.

Parameters **isAll** (bool) – If isAll is True then the count of all tokens is returned, if False the count of unconditional tokens is returned.

Returns int – Count of tokens.

tokenCountNonWs(*isAll*)
Returns the token count excluding whitespace tokens.

Parameters **isAll** (bool) – If isAll is True then the count of all tokens is returned, if False the count of unconditional tokens is returned.

Returns int – Count of tokens.

tokenTypesAndCounts(*isAll*, *allPossibleTypes=True*)
Generator the yields (type, count) in PpToken.LEX_PPTOKEN_TYPES order where type is a string and count an integer.

Parameters

- **isAll** (bool) – If isAll is True then the count of all tokens is returned. If False the count of unconditional tokens is returned.
- **allPossibleTypes** (bool) – 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.

Returns NoneType,tuple([str, int]) – Yields (type, count).

totalAll
The total token count.

Returns int – The count.

totalAllConditional
The token count of conditional tokens.

Returns int – The count.

totalAllUnconditional
The token count of unconditional tokens.

Returns int – The count.

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 tokens6) 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.**CHAR_SET_MAP = {'lex.charset': {'source character set': {'v', '<', '\x0b', 'J', 'I', '2', 'q', '&', 'Z**
Preprocess character sets:

The `'source character set'` should be 96 characters i.e. 91 plus whitespace (5) See assertions below that check length, if not content. Note: Before jumping to conclusions about how slow this might be go and look at TestPpTokeniserIsInCharSet NOTE: whitespace is now handled by the PpWhitespace class and this entry is dynamically added to CHAR_SET_MAP on import: `'whitespace' : set('\t\v\f\n '),`

Set of ordinal characters not treated as Universal Character names i.e. treated literally. NOTE: ISO/IEC 9899:1999 (E) 6.4.3-2 "A universal character name shall not specify a character whose short identifier is less than 00A0 other than 0024 ($), 0040 (@), or 0060 (back tick), nor one in the range D800 through DFFF inclusive.61)

['lex.header']['undefined_h_words'] are from: ISO/IEC 9899:1999 (E) 6.4.7 Header names Para 3 i.e. 6.4.7-3

'lex.key': From: ISO/IEC 14882:1998(E) 2.11 Keywords [lex.key] Note these are of no particular interest to the pre-processor as they do not occur in phases 1 to 4. For example 'new' is an operator but is re-interpreted after phase 4 (probably in phase 7) as a keyword.

'lex.op' From: ISO/IEC 14882:1998(E) 2.12 Operators and punctuators [lex.operators] These contain Digraphs and "Alternative Tokens" e.g. 'and' so that 'and' will be seen as an operator and an identifier. Similarly 'new' appears as an operator but is re-interpreted after phase 4 (probably in phase 7) as a keyword.

cpip.core.PpTokeniser.**CHAR_SET_STR_TREE_MAP = {'lex.op': {'operators': <cpip.util.StrTree.StrTree object>}, 'lex.**
Create StrTree objects for fast look up for words

cpip.core.PpTokeniser.**COMMENT_REPLACEMENT = ' '**
Comments are replaced by a single space

cpip.core.PpTokeniser.**COMMENT_TYPES = ('C comment', 'C++ comment')**
All comments

cpip.core.PpTokeniser.**COMMENT_TYPE_C = 'C comment'**
C comment

cpip.core.PpTokeniser.**COMMENT_TYPE_CXX = 'C++ comment'**
C++ comment

cpip.core.PpTokeniser.**C_KEYWORDS = ('auto', 'break', 'case', 'char', 'const', 'continue', 'default', 'do', 'double', 'else'**
'C' keywords *ISO/IEC 9899:1999 (E) 6.4.1 Keywords*

cpip.core.PpTokeniser.**DIGRAPH_TABLE = {'bitand': '&', 'xor': '^', '%:': '#', '<%': '{', 'not': '!', 'and': '&&', 'bitor**
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*, *theDiagnostic=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?

_PpTokeniser__sliceCCharCharacter(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.2 Character literals [lex.ccon] - c-char character.

Parameters

- **theBuf** (`cpip.util.BufGen.BufGen, str`) – Buffer or string.
- **theOfs** (`int`) – The starting offset.

Returns `int` – Length of matching characters found.

_PpTokeniser__sliceLexKey(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.11 Keywords [lex.key].

_PpTokeniser__sliceLexPpnumberDigit(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.9 Preprocessing numbers [lex.ppnumber] - digit.

Parameters

- **theBuf** (`cpip.util.BufGen.BufGen, str`) – Buffer or string.
- **theOfs** (`int`) – The starting offset.

Returns `int` – Length of matching characters found.

_PpTokeniser__sliceLexPpnumberExpSign(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.9 Preprocessing numbers [lex.ppnumber] - exponent and sign. Returns 2 if theCharS is 'e' or 'E' followed by a sign.

Parameters

- **theBuf** (`cpip.util.BufGen.BufGen, str`) – Buffer or string.
- **theOfs** (`int`) – The starting offset.

Returns `int` – Length of matching characters found.

_PpTokeniser__sliceLongSuffix(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.1 Integer literals [lex.icon] - long-suffix.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_PpTokeniser__sliceNondigit(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.10 Identifiers [lex.name] - nondigit:

```
nondigit: one of
universal-character-name
_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_PpTokeniser__sliceSimpleEscapeSequence(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.2 Character literals [lex.ccon] - simple-escape-sequence.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_PpTokeniser__sliceUniversalCharacterName(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.2 Character sets [lex.charset] - universal-character-name.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_PpTokeniser__sliceUnsignedSuffix(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.1 Integer literals [lex.icon] - unsigned-suffix.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

__init__(*theFileObj=None*, *theFileId=None*, *theDiagnostic=None*)
Constructor. 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.

Parameters

- **theFileObj** (NoneType, _io.StringIO, _io.TextIOWrapper) – The file object.

- **theFileId** (NoneType, str) – File ID.
- **theDiagnostic** (*cpip.core.CppDiagnostic.PreprocessDiagnosticStd*) – An optional diagnostic.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

_convertToLexCharset(*theLineS*)
Converts a list of lines expanding non-lex.charset characters to universal-character-name and returns a set of lines so encoded.

ISO/IEC 9899:1999 (E) 6.4.3

Note: *ISO/IEC 9899:1999 (E) 6.4.3-2 "A universal character name* shall not specify a character whose short identifier is less than 00A0 other than 0024 ($), 0040 (@), or 0060 (back tick), nor one in the range D800 through DFFF inclusive.61).

Note: This side-effects the supplied lines and returns None.

Parameters **theLineS** (list([]), list([str])) – The source code.

Returns NoneType

_rewindFile()
Sets the file to position zero and resets the FileLocator.

Returns NoneType

_sliceAccumulateOfs(*theBuf*, *theOfs*, *theFn*)
Repeats the function as many times as possible on theBuf from theOfs. An IndexError raised by the function will be caught and not propagated.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – Offset.
- **theFn** (method) – Function.

Returns int – The index of the find or -1 if none found.

_sliceBoolLiteral(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.5 String literals [lex.bool].

_sliceCChar(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.2 Character literals [lex.ccon] - c-char.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceCCharSequence(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.2 Character literals [lex.ccon] - c-char-sequence.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceCharacterLiteral(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.2 Character literals [lex.ccon].

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceDecimalLiteral(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.13.1 Integer literals [lex.icon] - decimal-literal.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceEscapeSequence(*theBuf*, *theOfs*)
Returns the length of a slice of theCharS that matches the longest integer literal or 0. ISO/IEC 14882:1998(E) 2.13.2 Character literals [lex.ccon] - escape-sequence.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceFloatingLiteral(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.3 Floating literals [lex.fcon]:

```
floating-literal:
   fractional-constant exponent-part opt floating-suffix opt
   digit-sequence exponent-part floating-suffix opt
```

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceFloatingLiteralDigitSequence(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.3 Floating literals [lex.fcon] - digit-sequence.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceFloatingLiteralExponentPart(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.3 Floating literals [lex.fcon] - exponent-part.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceFloatingLiteralFloatingSuffix(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.3 Floating literals [lex.fcon] - floating-suffix.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceFloatingLiteralFractionalConstant(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.3 Floating literals [lex.fcon] - fractional-constant:

```
fractional-constant:
   digit-sequence opt . digit-sequence
   digit-sequence .
```

i.e there are three posibilities: a: . digit-sequence b: digit-sequence . c: digit-sequence . digit-sequence

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceFloatingLiteralSign(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.3 Floating literals [lex.fcon] - floating-suffix.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceHexQuad(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.2 Character sets [lex.charset] - hex-quad.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceHexadecimalEscapeSequence(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.2 Character literals [lex.ccon] - hexadecimal-escape-sequence:

```
hexadecimal-escape-sequence:
    \x hexadecimal-digit
    hexadecimal-escape-sequence hexadecimal-digit
```

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceHexadecimalLiteral(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.13.1 Integer literals [lex.icon] - hexadecimal-literal.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceIntegerLiteral(*theBuf*, *theOfs=0*)
Returns the length of a slice of theCharS that matches the longest integer literal or 0. ISO/IEC 14882:1998(E) 2.13.1 Integer literals [lex.icon].

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceIntegerSuffix(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.1 Integer literals [lex.icon] - integer-suffix:

```
integer-suffix:
    unsigned-suffix long-suffix opt
    long-suffix unsigned-suffix opt
```

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexComment(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.7 Comments [lex.comment].

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexHeader(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.8 Header names [lex.header]. Might raise a ExceptionCpipUndefinedLocal.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexHeaderHchar(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.8 Header names [lex.header] - h-char character.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexHeaderHcharSequence(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.8 Header names [lex.header] - h-char-sequence. Might raise a Exception-CpipUndefinedLocal.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexHeaderQchar(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.8 Header names [lex.header] - q-char.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexHeaderQcharSequence(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.8 Header names [lex.header] - q-char-sequence. Might raise a Exception-CpipUndefinedLocal.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexKey(*theBuf*, *theOfs=0*)
ISO/IEC 14882:1998(E) 2.11 Keywords [lex.key].

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexName(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.10 Identifiers [lex.name]:

```
identifier:
    nondigit
    identifier nondigit
    identifier digit
```

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexOperators(*theBuf*, *theOfs=0*)

ISO/IEC 14882:1998(E) 2.12 Operators and punctuators [lex.operators]. i.e. preprocessing-op-or-punc

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexPpnumber(*theBuf*, *theOfs=0*)

ISO/IEC 14882:1998(E) 2.9 Preprocessing numbers [lex.ppnumber]:

```
pp-number:
    digit
    . digit
    pp-number digit
    pp-number nondigit
    pp-number e sign
    pp-number E sign
    pp-number .
```

TODO: Spec says "Preprocessing number tokens lexically include all integral literal tokens (2.13.1) and all floating literal tokens (2.13.3)." But the pp-number list does not specify that.

NOTE: ISO/IEC 9899:1999 Programming languages - C allows 'p' and 'P' suffixes.

NOTE: The standard appears to allow '.1.2.3.4.'

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexPptoken(*theBuf*, *theOfs*)

ISO/IEC 14882:1998(E) 2.4 Preprocessing tokens [lex.pptoken].

ISO/IEC 9899:1999 (E) 6.4 Lexical elements

NOTE: Does not identify header-name tokens. See NOTE on genLexPptokenAndSeqWs()

NOTE: _sliceLexPptokenGeneral is an exclusive search as 'bitand' can appear to be both an operator (correct) and an identifier (incorrect). The order of applying functions is therefore highly significant _sliceLexPpnumber must be before _sliceLexOperators as the leading '.' on a number can be seen as an operator. _sliceCharacterLiteral and _sliceStringLiteral must be before _sliceLexName as the leading 'L' on char/string can be seen as a name.

self._sliceLexOperators has to be after self._sliceLexName as otherwise:

```
#define complex
```

gets seen as:

```
# -> operator
define -> identifier
compl -> operator because of alternative tokens
ex -> identifier
```

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLexPptokenGeneral(*theBuf*, *theOfs*, *theFuncS*)
Applies theFuncS to theCharS and returns the longest match.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – Integer offset.
- **theFuncS** (tuple([method, method, method, method, method])) – Sequence of functions.

Returns int – Length of the longest match.

_sliceLexPptokenWithHeaderName(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.4 Preprocessing tokens [lex.pptoken].

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

Note: This does identify header-name tokens where possible.

_sliceLiteral(*theBuf*, *theOfs=0*)
Returns the length of a slice of theCharS that matches the longest integer literal or 0. ISO/IEC 14882:1998(E) 2.13 Literals [lex.literal].

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceLongestMatchOfs(*theBuf*, *theOfs*, *theFnS*, *isExcl=False*)
Returns the length of the longest slice of theBuf from theOfs using the functions theFnS, or 0 if nothing matches.

This preserves self._cppTokType to be the one that gives the longest match. Functions that raise an IndexError will be ignored.

If isExcl is False (the default) then all functions are tested, longest match is returned.

If isExcl is True then first function returning a non-zero value is used.

TODO (maybe): Have slice functions return (size, type) and get rid of self._changeOfTokenTypeIsOk and self._cppTokType

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – Offset.
- **theFnS** (tuple([method, method, method, method, method]), tuple([method, method, method]), tuple([method, method])) – Functions.
- **isExcl** (bool) – Exclusivity flag.

Returns int – The length of the longest match.

_sliceNonWhitespaceSingleChar(*theBuf*, *theOfs=0*)

Returns 1 if the first character is non-whitespace, 0 otherwise. *ISO/IEC 9899:1999 (E) 6.4-3 and ISO/IEC 14882:1998(E) 2.4.2* States that if the character is ' or " the behaviour is undefined.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceOctalEscapeSequence(*theBuf*, *theOfs*)

ISO/IEC 14882:1998(E) 2.13.2 Character literals [lex.ccon] - octal-escape-sequence:

```
octal-escape-sequence:
    \ octal-digit
    \ octal-digit octal-digit
    \ octal-digit octal-digit octal-digit
```

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceOctalLiteral(*theBuf*, *theOfs=0*)

ISO/IEC 14882:1998(E) 2.13.1 Integer literals [lex.icon] - octal-literal.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceSChar(*theBuf*, *theOfs*)

ISO/IEC 14882:1998(E) 2.13.4 String literals [lex.string] - s-char.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceSCharCharacter(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.4 String literals [lex.string] - s-char character.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceSCharSequence(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.4 String literals [lex.string] - s-char-sequence.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceStringLiteral(*theBuf*, *theOfs*)
ISO/IEC 14882:1998(E) 2.13.4 String literals [lex.string].

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_sliceWhitespace(*theBuf*, *theOfs=0*)
Returns whitespace length.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – The starting offset.

Returns int – Length of matching characters found.

_translateTrigraphs(*theLineS*)
ISO/IEC 14882:1998(E) 2.3 Trigraph sequences [lex.trigraphs]

This replaces the trigraphs in-place and updates the FileLocator so that the physical lines and columns can be recovered.

Parameters **theLineS** (list([]), list([str])) – Source code lines.

Returns NoneType

_wordFoundInUpTo(*theBuf*, *theLen*, *theWord*)
Searches theBuf for any complete instance of a word in theBuf. Returns the index of the find or -1 if none found.

TODO: This looks wrong, buffer = ‘ abc abd’, word = ‘abd’ will return -1

Parameters

- **theBuf** (str) – Buffer or string.
- **theLen** (int) – Buffer length.
- **theWord** (str) – Word to find.

Returns int – The index of the find or -1 if none found.

_wordsFoundInUpTo(*theBuf*, *theLen*, *theWordS*)
Searches theCharS for any complete instance of any word in theWordS. Returns the index of the find or -1 if none found.

Parameters

- **theBuf** (str) – Buffer or string.
- **theLen** (int) – Buffer length.
- **theWordS** (set([str])) – Set of words, any of these can be found.

Returns int – The index of the find or -1 if none found.

cppTokType
Returns the type of the last preprocessing-token found by _sliceLexPptoken().

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

Returns cpip.core.FileLocation.FileLineCol([str, int, int]) – The FileLineCol object

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).

Parameters **theToks** (list([cpip.core.PpToken.PpToken])) – The tokens.

Returns list([cpip.core.PpToken.PpToken]) – List of 'header-name' tokens.

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."*.

Parameters **theCharS** (str) – The source code.

Returns *cpip.core.PpToken.PpToken* – Sequence of tokens.

Raises GeneratorExit, IndexError

initLexPhase12()
Process phases one and two and returns the result as a string.

Returns str – <insert documentation for return values>

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

Returns list([]),list([str]) – List of source code lines.

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)and replaces trigraphs.

NOTE: This side-effects the supplied lines and returns None.

Parameters **theLineS** (list([]), list([str])) – The source code.

Returns NoneType

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.

Parameters **theLineS** (list([]), list([str])) – The source code.

Returns NoneType

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.

Returns *cpip.core.PpToken.PpToken* – Sequence ot tokens.

Raises GeneratorExit, StopIteration

pLineCol
Returns the current physical (line, column) as integers.

Returns tuple([int, int]) – Physical position.

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.

Parameters **theToks** (list([cpip.core.PpToken.PpToken])) – The tokens.

Returns list([cpip.core.PpToken.PpToken]) – List of 'header-name' tokens.

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.**DEFINE_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 = {'\x0b', '\n', '\t', ' ', '\x0c'}**
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 preprocesing macros.

ISO/IEC 14882:1998(E) 16-2 only ' ' and ' ' as whitespace.

Parameters **theCharS** (str) – The character string.

Returns bool – True if empty or all DEFINE_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.

Parameters **theCharS** (str) – The character string.

Returns bool – True if there is a LEX_NEWLINE in the string.

preceedsNewline(*theCharS*)
Returns True if theChars ends with a newline. i.e. this immediately precedes a new line.

Parameters **theCharS** (str) – String of characters.

Returns bool – True if ends with a LEX_NEWLINE

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.

Parameters

- **theBuf** (cpip.util.BufGen.BufGen, str) – Buffer or string.
- **theOfs** (int) – Offset into the buffer.

Returns int – Number of whitespace characters.

Raises IndexError

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 preprocessed 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 reinterpreted 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.

8.2.11 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.

__getitem__ (*key*)
Implements indexing and slicing. Negative indexes will raise an IndexError.

Parameters **key** (int) – The index.

Returns str – The buffer corresponding to the key.

Raises IndexError, StopIteration

__init__ (*theGen*)
Constructor with a generator as an argument.

Parameters **theGen** (generator) – The generator to use.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

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.

Cpp

Provides various utilities for interfacing with the platform installaion of cpp. This includes using cpp as a sub-process to extract macros. This also provides some standard cpp like options used by both `cpp.py` and `CPIPMain.py`

Created on 24 Jan 2015

@author: paulross

cpip.util.Cpp.**addStandardArguments**(*parser*)
This adds standard command line arguments to an argparse argument parser.

Parameters **parser** (`argparse.ArgumentParser`) – <insert documentation for argument>

Returns `NoneType`

cpip.util.Cpp.**invokeCppForPlatformMacros**(**args*)
Invoke the pre-processor as a sub-process with *args and return a list of macro definition strings.

By default the preprocessor is `cpp` but this is overridden if `$CPP` is set in the environment.

May raise subprocess.CalledProcessError on failure.

cpip.util.Cpp.**macroDefinitionDict**(*cmdLineArgS*)
Given a list of command line arguments of the form n<=d> where n is the macro name and d the optional definition this returns an ordered dict of {n : d, ...}.

cpip.util.Cpp.**macroDefinitionString**(*cmdLineArgS*)
Given a list of command line arguments of the form `n<=d>` where `n` is the macro name and `d` the optional definition this returns a same ordered multi-line string where each line is of the form `#define n d` or `#define n` if d is not present.

Parameters **cmdLineArgS** (`list([])`) – Comman line arguments as strings.

Returns `str` – Multi line string of macro definitions.

cpip.util.Cpp.**predefinedFileObjects**(*args*)
Returns a list of file like objects to be pre-processed before the ITU. This does in this order:

1. Platform specific macros
2. Any command line defines
3. Any pre-included files

Parameters **args** (`argparse.Namespace`) – Parsed arguments.

Returns `list([_io.StringIO])` – List of file like objects.

cpip.util.Cpp.**stdPredefinedMacros**(*args*)
Returns a dict of standard predefined macros specified on the command line. See ISO/IEC 9899:1999 (E) 6.10.8 Predefined macro 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

__len__()
Returns the number of keys.

__weakref__
list of weak references to the object (if defined)

_depth(*theD*)
Recursively returns the maximum tree depth as an integer.

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:

```
                          |- AAA
                          |
                  |- AA --|- AAB
                  |       |
                  |       |- AAC
         |- A ---|
Root ----|       |- AB
         |       |
         |       |- AC ---- ACA
         |
         |- B
         |
         |- C ---- CA ---- CAA
```

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(*keyBranch*)

Returns a set of events that are a tuple of quadruples `(key_branch, value, rowspan_integer, colspan_integer)`

This is an internal recursive call.

For example: `(['a', 'b'], 'c', 3, 7)`

At the start of the a <tr> there will be a `ROW_OPEN` yielded and at row end (`</tr>`) a `ROW_CLOSE` will be yielded.

Parameters **keyBranch** (`list([]), list([str])`) – Branch

Returns `NoneType,tuple([NoneType, int, <class 'int'>]), tuple([list([str]), NoneType, int, <class 'int'>]), tuple([list([str]), NoneType, int, int]),tuple([list([str]), list([tuple([str, <class 'str'>])]), int, <class 'int'>]), tuple([list([str]), list([tuple([str, str])]), int, <class 'int'>]),tuple([list([str]), tuple([str, str, tuple([int, int, int])]), <class 'int'>, <class 'int'>])` – <insert documentation for return values>

Raises `StopIteration`

_setColSpan(*mD*, *d*)

Sets the column span.

This is an internal recursive call.

Parameters

- **mD** (`int`) – ???
- **d** (`int`) – Depth

Returns `NoneType`

_setRowSpan()

Sets self._rowSpan recursively.

Returns `int` – The row span.

colSpan

The column span.

Returns `int` – The span.

genColRowEvents()

Returns a set of events that are a tuple of quadruples `(key_branch, value, rowspan_integer, colspan_integer)`

For example: `(['a', 'b'], 'c', 3, 7)`

At the start of the a <tr> there will be a `ROW_OPEN` yielded and at row end (`</tr>`) a `ROW_CLOSE` will be yielded.

Returns `NoneType,tuple([NoneType, int, <class 'int'>]), tuple([list([str]), NoneType, int, <class 'int'>]), tuple([list([str]), NoneType, int, int]),tuple([list([str]),`

list([tuple([str, <class 'str'>])]), int, <class 'int'>), tuple([list([str]), list([tuple([str, str])]), int, <class 'int'>]),tuple([list([str]), tuple([str, str, tuple([int, int, int])]), <class 'int'>, <class 'int'>]) – <insert documentation for return values>

Raises StopIteration

retNewInstance()
A new instance of a *cpip.util.DictTree.DictTreeHtmlTable*.

Returns *cpip.util.DictTree.DictTreeHtmlTable* – A new instance.

rowSpan
The row span.

Returns int – The span.

setColRowSpan()
Top level call that sets colspan and rowspan attributes.

Returns NoneType

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.

__weakref__
list of weak references to the object (if defined)

class cpip.util.DirWalk.**FileInOut**(*filePathIn*, *filePathOut*)
A pair of (in, out) file paths

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *filePathIn*, *filePathOut*)
Create new instance of FileInOut(filePathIn, filePathOut)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0xa385c0>*, *len=<built-in function len>*)
Make a new FileInOut object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new FileInOut object replacing specified fields with new values

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 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.

Parameters **p** (str) – The path.

Returns list([str]) – The split path.

cpip.util.HtmlUtils.**retHtmlFileLink**(*theSrcPath*, *theLineNum*)
Returns a string that is a link to a HTML file.

Parameters

- **theSrcPath** (str) – The path of the original source, whis will be encoded with *retHtmlFileName()*.
- **theLineNum** (int) – An integer line number in the target.

Returns str – The link text.

cpip.util.HtmlUtils.**retHtmlFileName**(*thePath*)
Creates a unique, short, human readable file name base on the input file path. This is the file name plus a hash of the path.

Parameters **thePath** (str) – The file path.

Returns str – The name.

cpip.util.HtmlUtils.**writeCharsAndSpan**(*theS*, *theText*, *theSpan*)
Write theText to the stream theS. If theSpan is not None the text is enclosed in a element.

Parameters

- **theS** (cpip.util.XmlWrite.XhtmlStream) – The XHTML stream.
- **theText** (str) – The text to write, must be non-empty.

- **theSpan** (str) – CSS class for the text.

Returns NoneType

cpip.util.HtmlUtils.**writeDictTreeAsTable**(*theS*, *theDt*, *tableAttrs*, *includeKeyTail*)
Writes a *cpip.util.DictTree.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`.

Parameters

- **theS** (cpip.util.XmlWrite.XhtmlStream) – HTML stream.
- **theDt** (*cpip.util.DictTree.DictTreeHtmlTable*) – The table.
- **tableAttrs** (dict({str : [str]})) – Table element attributes.
- **includeKeyTail** (bool) – Include keys in the tail.

Returns NoneType

Raises StopIteration

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.

Parameters

- **valueType** (NoneType, str) – The type of the value: `None, |'list' | 'set'`
- **theS** (cpip.util.XmlWrite.XhtmlStream) – The HTML stream.
- **theKvS** (list([tuple([str, tuple([str, str, tuple([<class 'int'>, int, int])])]), tuple([str, tuple([str, str, tuple([int, int, int])])])]), list([tuple([str, tuple([str, str])])])) – A list of pairs `(file_path, value)`.
- **tableStyle** (str) – The CSS style used for the table.
- **fnTd** (function) – 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** (NoneType, function) – Optional 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>`.

Returns `NoneType`

Raises `StopIteration`

`cpip.util.HtmlUtils.`**`writeHtmlFileAnchor`**(*theS*, *theLineNum*, *theText=''*, *theClass=None*, *theHref=None*)

Writes an anchor.

Parameters

- **`theS`** (`cpip.util.XmlWrite.XhtmlStream`) – The XHTML stream.
- **`theLineNum`** (`int`) – An integer line number in the target.
- **`theText`** (`str`) – Navigation text.
- **`theClass`** (`str`) – CSS class for the navigation text.
- **`theHref`** (`NoneType, str`) – The HREF.

Returns `NoneType`

`cpip.util.HtmlUtils.`**`writeHtmlFileLink`**(*theS*, *theSrcPath*, *theLineNum*, *theText=''*, *theClass=None*)

Writes a link to another HTML file that represents source code.

Parameters

- **`theS`** (`cpip.util.XmlWrite.XhtmlStream`) – The XHTML stream.
- **`theSrcPath`** (`str`) – The path of the original source, whis will be encoded with *`retHtmlFileName()`*.
- **`theLineNum`** (`int`) – An integer line number in the target.
- **`theText`** (`str`) – Navigation text.
- **`theClass`** (`NoneType, str`) – Optional CSS class for the navigation text.

Returns `NoneType`

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.

__init__(*theList*, *theGen=None*)
Initialise the class with a list of objects to yield and, optionally, a generator. If the generator is present it will be used as a continuation of the list.

__next__()
yield the next value. The attribute listIsEmpty will be set True immediately before yielding the last value.

__weakref__
list of weak references to the object (if defined)

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.

__init__()
Constructor.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

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.

Returns NoneType

sideEffect(*theMat*)
Makes the replacement, if line/col is out of range and *ExceptionMatrixRep* will be raised and the state of theMat argument is undefined.

Parameters **theMat** (list([]), list([str])) – The matrix.

Returns NoneType

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.

__init__(*theGen*, *theFnS*, *isExclusive=False*, *yieldReplacement=False*)
Constructor that takes a generator and a list of Maximal munch functions.

Each function is required to take a generator as an object and return a triple (count, kind, replace) where:

count - is a integer kind - is arbitrary. replace - is None or an iterable.

Typically the functions should be written thus:

```
def f(theGen):
    i = 0
    for aVal in theGen:
        if not <some condition of aVal>:
            break
        i +=1
    return i, <kind>, <replace>
```

Or (note the catching of StopIteration):

```
def f(theGen):
    i = 0
    try:
        while theGen.next() <some condition>:
            i += 1
    except StopIteration:
        pass
    return i, <kind>, <replace>
```

If `isExclusive` is `True` then the first function that returns a non-zero integer will be used and the others will not be exercised for that token.

__weakref__
list of weak references to the object (if defined)

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.

MultiPassString

Converts an ITU to HTML.

class cpip.util.MultiPassString.**MultiPassString**(*theFileObj*)
Reads a file, the file can be translated any number of times and marked with word types. The latter can then be generated using BufGen for example:

```
myBg = BufGen.BufGen(self._mps.genChars())
try:
    i = 0
    while 1:
        print myBg[i]
        i += 1
except IndexError:
    pass
```

_MultiPassString__set(*idx*, *repl*)
Sets a marker.

Parameters

- **idx** (int) – Index of marker.
- **repl** (str) – The marker.

Returns NoneType

Raises ExceptionMultiPass in index error.

__init__(*theFileObj*)
Constructor.

Parameters **theFileObj** (_io.TextIOWrapper) – File like object.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

_retZeroIndex()
Returns the index to the first non-empty token in the current list.

Returns int – The index.

clearMarker()
The mark at this point in the input.

Returns NoneType

genChars()
Generates the current set of characters. This can be used as the generator for the BufGen and that BufGen can be passed to the PpTokeniser _slice... Functions.

genWords()
Generates pairs (word, type) from the original string.

TODO: Solve the overlap problem.

Returns NoneType, tuple([str, str]) – a pair of (word, type)

hasWord
True if the length of the current word is > 0.

prevChar
The previous character of the input. A slight nod to K&R this is (not) a bit like putc().

removeMarkedWord(*isTerm*)
Remove the current marked word. isTerm is a boolean that is True if the current position is a terminal character.

For example if you want to split a string into lines then n is a terminal character and you would call this with isTerm=True.

However if you were splitting a string into words and whitespace then a whitespace following a word is not the terminal character so at the pint of receiving the whitespace character you would call this with isTerm=False

Parameters **isTerm** (bool) – True if a terminal character.

Returns NoneType

removeSetReplaceClear(*isTerm*, *theType*, *theRepl*)
This provides a helper combination function for a common operation of:

- Removing the marked word from the output.
- Setting the word type in the input.
- Replacing the marked word with a replacement string.
- Clearing the current marker.

Parameters

- **isTerm** (bool) – See *removeMarkedWord()* for an explanation of this.
- **theType** (str) – See *removeMarkedWord()* for an explanation of this.
- **theRepl** (str) – See *setAtMarker()* for an explanation of this.

Returns NoneType

setAtMarker(*theRepl*)
Sets the token at the current marker to be theRepl.

Parameters **theRepl** (str) – The repl.

Returns NoneType

Raises ExceptionMultiPass no marker present.

setMarker()
Sets a mark at this point in the input.

Returns NoneType

setWordType(*theType*, *isTerm*)
Marks a word in the input as a word of type theType starting from the marker up to the current place.

See *removeMarkedWord()* for an explanation of isTerm.

Parameters

- **theType** (str) – The type.
- **isTerm** (bool) – Is a terminal character.

Returns NoneType

wordLength
The length of the current word.

Returns int – The length.

class cpip.util.MultiPassString.**Word**(*wordLen*, *wordType*)

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *wordLen*, *wordType*)
Create new instance of Word(wordLen, wordType)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0xa385c0>*, *len=<built-in function len>*)
Make a new Word object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new Word object replacing specified fields with new values

wordLen
Alias for field number 0

wordType
Alias for field number 1

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*)
A string tree or Trie.

__init__(*theIterable=None*)
Initialise the class with a optional sequence of strings.

Parameters **theIterable** (NoneType, set([str])) – A sequence of strings.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

add(*s*)
Add a string.

Parameters **s** (str) – The string to add.

Returns NoneType

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.

Parameters

- **s** (*cpip.util.BufGen.BufGen*, str) – value
- **i** (int) – index

Returns int – index.

Raises IndexError, KeyError 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.

__init__()
Constructor.

__weakref__
list of weak references to the object (if defined)

_add(*theMap*, *k*, *v*)
Adds the key/value to the existing map.

Parameters

- **theMap** (dict({})) – Existing map.
- **k** (str) – Key.
- **v** (str) – Value.

Returns NoneType

_treeFromEither(*theObj*, *theMap*)
Creates a tree from the object.

Parameters

- **theObj** (str) – The object to create a tree from.
- **theMap** (dict({str : [list([str])]})) – The map of str/str.

Returns *cpip.util.Tree.Tree* – The final tree.

_treeFromMap(*theMap*, *theStack*, *theTree*)
Creates a Tree from a dict of list of strings.

Parameters

- **theMap** (dict({str : [list([str])]})) – The dictionary.
- **theStack** (list([str])) – The stack of strings.
- **theTree** (*cpip.util.Tree.Tree*) – An existing Tree.

Returns NoneType

add(*parent*, *child*)
Adds the parent/child to both internal maps.

Parameters

- **parent** (str) – Parent.
- **child** (str) – Child.

Returns NoneType

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.

Parameters **child** (str) – The child

Returns bool – True if the argument is a child is in the map.

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',]
```

Parameters **theObj** (str) – The object to create a tree from.

Returns *cpip.util.Tree.Tree* – The final tree.

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:

```
treeChildParent('a') # returns ['a', 'c', 'b',]
treeChildParent('b') # returns ['b', 'a', 'c',]
treeChildParent('c') # returns ['c', 'b', 'a',]
```

Parameters **theObj** (str) – The object to create a tree from.

Returns *cpip.util.Tree.Tree* – The final tree.

class cpip.util.Tree.**Tree** (*obj*)
Represents a simple tree of objects.

__init__ (*obj*)
Constructor, takes any object.

Parameters **obj** (object) – Any object, usually a string.

Returns NoneType

__weakref__
list of weak references to the object (if defined)

_branches (*thisBranch*)
<insert documentation for function>

Parameters **thisBranch** (NoneType, list([str])) – Current branch, None initially.

Returns list([list([str])]) – List of branches.

addChild (*obj*)
Add a child.

Parameters **obj** (str) – Child.

Returns NoneType

branches()
Returns all the possible branches through the tree as a list of lists of self._obj.

Returns list([list([str])]) – List of branches.

youngestChild
The latest child to be added, may raise IndexError if no children.

Returns *cpip.util.Tree.Tree* – The youngest child.

XmlWrite

Writes XML and XHTML.

class cpip.util.XmlWrite.**Element**(*theXmlStream*, *theElemName*, *theAttrs=None*)
Represents an element in a markup stream.

__enter__()
Context manager support.

Returns cpip.plot.SVGWriter.SVGGroup,cpip.plot.SVGWriter.SVGLine, cpip.plot.SVGWriter.SVGRect,cpip.plot.SVGWriter.SVGText,cpip.util.XmlWrite.Element – self

__exit__(*excType*, *excValue*, *tb*)
Context manager support. TODO: Should respect RAISE_ON_ERROR here if excType is not None.

Parameters

- **excType** (NoneType) – Exception type, if raised.
- **excValue** (NoneType) – Exception, if raised.
- **tb** (NoneType) – Traceback, if raised.

Returns NoneType

__init__(*theXmlStream*, *theElemName*, *theAttrs=None*)
Constructor.

Parameters

- **theXmlStream** (cpip.plot.SVGWriter.SVGWriter, cpip.util.XmlWrite.XhtmlStream) – The XML stream.
- **theElemName** (str) – Element name.
- **theAttrs** (NoneType, dict({str : [str]}), dict({})) – Element attributes

Returns NoneType

__weakref__
list of weak references to the object (if defined)

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.**XhtmlStream**(*theFout*, *theEnc='utf-8'*, *theDtdLocal=None*, *theId=0*, *mustIndent=True*)

Specialisation of an XmlStream to handle XHTML.

__enter__()

Context manager support.

Returns cpip.util.XmlWrite.XhtmlStream – self

charactersWithBr(*sIn*)

Writes the string replacing any \n characters with
 elements.

Parameters **sIn** (str) – The string to write.

Returns NoneType

class cpip.util.XmlWrite.**XmlStream**(*theFout*, *theEnc='utf-8'*, *theDtdLocal=None*, *theId=0*, *mustIndent=True*)

Creates and maintains an XML output stream.

__enter__()

Context manager support.

Returns cpip.plot.SVGWriter.SVGWriter,cpip.util.XmlWrite.XhtmlStream – self

__exit__(*exc_type*, *exc_value*, *traceback*)

Context manager support.

Parameters

- **excType** (NoneType) – Exception type, if raised.
- **excValue** (NoneType) – Exception, if raised.
- **tb** (NoneType) – Traceback, if raised.

Returns NoneType

__init__(*theFout*, *theEnc='utf-8'*, *theDtdLocal=None*, *theId=0*, *mustIndent=True*)

Initialise with a writable file like object or a file path.

Parameters

- **theFout** (_io.TextIOWrapper, str) – The file-like object or a path as a string. If the latter it will be closed on __exit__.
- **theEnc** (str) – The encoding to be used.
- **theDtdLocal** (NoneType, str) – Any local DTD as a string.
- **theId** (int) – An integer value to use as an ID string.
- **mustIndent** (bool) – Flag, if True the elements will be indented (pretty printed).

Returns NoneType

__weakref__

list of weak references to the object (if defined)

_canIndent

Returns True if indentation is possible (no mixed content etc.).

Returns bool – True if the element can be indented.

_closeElemIfOpen()
Close the element if open.

Returns NoneType

_encode(*theStr*)
"Apply the XML encoding such as '<' to '<'

Parameters **theStr** (str) – String to encode.

Returns str – Encoded string.

_flipIndent(*theBool*)
Set the value at the tip of the indent stack to the given value.

Parameters **theBool** (bool) – Flag for indenting.

Returns NoneType

_indent(*offset=0*)
Write out the indent string.

Parameters **offset** (int) – The offset.

Returns NoneType

characters(*theString*)
Encodes the string and writes it to the output.

Parameters **theString** (str) – The content.

Returns NoneType

comment(*theS*, *newLine=False*)
Writes a comment to the output stream.

Parameters

- **theS** (str) – The comment.
- **newLine** (bool) – If True the comment is written on a new line, if False it is written inline.

Returns NoneType

endElement(*name*)
Ends an element.

Parameters **name** (str) – Element name.

Returns NoneType

id
A unique ID in this stream. The ID is incremented on each call.

Returns str – The ID.

literal(*theString*)
Writes theString to the output without encoding.

Parameters **theString** (str) – The content.

Returns NoneType

pI(*theS*)
Writes a Processing Instruction to the output stream.

startElement(*name*, *attrs*)
Opens a named element with attributes.

Parameters

- **name** (str) – Element name.
- **attrs** (dict({str : [str]}), dict({})) – Element attributes.

Returns NoneType

writeCDATA(*theData*)
Writes a CDATA section.

Example:

```
<![CDATA[
...
]]>
```

Parameters **theData** (str) – The CDATA content.

Returns NoneType

writeCSS(*theCSSMap*)
Writes a style sheet as a CDATA section. Expects a dict of dicts.

Example:

```
<style type="text/css"><![CDATA[
    ...
]]></style>
```

Parameters **theCSSMap** (dict({str : [dict({str : [str]}), dict({str : [str]})]})) – Map of CSS elements.

Returns NoneType

writeECMAScript(*theScript*)
Writes the ECMA script.

Example:

```
<script type="text/ecmascript">
//<![CDATA[
...
// ]]>
</script>
```

Parameters **theData** (str) – The ECMA script content.

Returns NoneType

xmlSpacePreserve()
Suspends indentation for this element and its descendants.

Returns NoneType

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.

From RFC3548:

```
                  Table 1: The Base 64 Alphabet
Value Encoding  Value Encoding  Value Encoding  Value Encoding
    0 A            17 R            34 i            51 z
    1 B            18 S            35 j            52 0
    2 C            19 T            36 k            53 1
    3 D            20 U            37 l            54 2
    4 E            21 V            38 m            55 3
    5 F            22 W            39 n            56 4
    6 G            23 X            40 o            57 5
    7 H            24 Y            41 p            58 6
    8 I            25 Z            42 q            59 7
    9 J            26 a            43 r            60 8
   10 K            27 b            44 s            61 9
   11 L            28 c            45 t            62 +
   12 M            29 d            46 u            63 /
   13 N            30 e            47 v
   14 O            31 f            48 w         (pad) =
   15 P            32 g            49 x
   16 Q            33 h            50 y
```

See section 3 of : http://www.faqs.org/rfcs/rfc3548.html

Parameters

- **theS** (str) – The string to be encoded.
- **theCharPrefix** (str) – A character to prefix the string.

Returns str – Encoded string.

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.

Parameters **theStr** (str) – The string to be encoded.

Returns str – Encoded string.

8.2.12 cpip.plot

Coord

Main Classes

Most classes in this module are collections.namedtuple objects.

Class	Description	Attributes
`Dim`	Linear dimension	value units
`Box`	A Box	width depth
`Pad`	Padding around a tree object	prev next, parent child
`Margin`	Padding around an object	left right top bottom
`Pt`	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.`**`BASE_UNITS = 'px'`**
Base units

`cpip.plot.Coord.`**`UNIT_MAP = {'pt': 1.0, 'mm': 2.834645669291339, None: 1.0, 'in': 72.0, 'px': 1.0, 'pc': 12.0, 'cm': 28.34`**
Map of conversion factors, base unit is pixels.

`cpip.plot.Coord.`**`UNIT_MAP_DEFAULT_FORMAT = {'pt': '%d', 'mm': '%.1f', None: '%d', 'in': '%.3f', 'px': '%d', 'pc':`**
Formatting strings for writing attributes. We are trying not to write 3.999999999mm here!

`cpip.plot.Coord.`**`UNIT_MAP_DEFAULT_FORMAT_WITH_UNITS = {'pt': '%d%s', 'mm': '%.1f%s', None: '%d%s', 'in':`**
Map of formatting strings for value and units e.g. to create '0.667in' from (2.0 / 3.0, 'in')

`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.

Parameters

- **`val`** (`float, int`) – The value
- **`unitFrom`** (`NoneType, str`) – The initial units.
- **`unitTo`** (`str`) – The new units.

Returns `float,int` – The value in the new units.

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.

Parameters **theLen** (float, int) – Length.

Returns cpip.plot.Coord.Dim([float, str]) – A new dimension of theLen in base units.

cpip.plot.Coord.**zeroBaseUnitsDim**()
Returns a Coord.Dim() of zero length and units BASE_UNITS.

Returns cpip.plot.Coord.Dim([float, str]) – A new dimension of zero.

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.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, str])]) – A new point with the values [0, 0].

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.

Parameters

- **theP** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str])])]) – The initial point.
- **incX** (NoneType, cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([int, str])) – Distance to move in the x axis.
- **incY** (NoneType, cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([int, str])) – Distance to move in the y axis.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, str])]) – The new point.

cpip.plot.Coord.**convertPt**(*theP*, *theUnits*)

Returns a new point with the dimensions of theP converted to theUnits.

TODO: Deprecate this.

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 ######  ~                     *
*                ~    #                         #    ~                     *
*                ~    #                         #    ~                     *
*                ~    #          Parent         #    ~                     *
*                ~    #                         #    ~                     *
*                ~    ## 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_land    PC_stop
                      |            |
                      x>>>>>>>>>>>>x
                     /
                    /
    x>>>>>>>>>>>>x/
    |            |
PC_roll        PC_to
```

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_stop     CP_land
    |          |
    x<<<<<<<<<<x\
                \
                 \
                  x<<<<<<<<<<<<x
                  |             |
                CP_to        CP_roll
```

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 cpip.plot.Coord.Box or None. i.e. the box ....

Returns cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Bounding box.

bbChildrenDepth
The bounding box depth of children as a *cpip.plot.Coord.Dim* or None. i.e. the depth of box ....

Returns cpip.plot.Coord.Dim([float, str]) – Depth

bbChildrenWidth
The bounding box width of children as a *cpip.plot.Coord.Dim* or None. i.e. the width of box ....

Returns cpip.plot.Coord.Dim([float, str]) – Width

bbSelfDepth
The depth of self plus padding as a *cpip.plot.Coord.Dim*. i.e. the depth of box ~~~~

Returns cpip.plot.Coord.Dim([float, str]) – Depth

bbSelfPadding
The immediate padding around self as a *cpip.plot.Coord.Pad*.

Returns NoneType,cpip.plot.Coord.Pad([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>]), cpip.plot.Coord.Dim([float, <class 'str'>]), cpip.plot.Coord.Dim([<class 'float'>, <class 'str'>])]) – Padding

bbSelfWidth
The width of self plus padding as a *cpip.plot.Coord.Dim* or None. i.e. the width of box ~~~~

Returns cpip.plot.Coord.Dim([float, str]) – Width

bbSigma
Bounding box of self and my children as a cpip.plot.Coord.Box.

bbSigmaDepth
The depth of self+children as a *cpip.plot.Coord.Dim* or None in the case that I don't exist and I have no children. i.e. the depth of box ****

Returns cpip.plot.Coord.Dim([float, str]) – Depth

bbSigmaWidth
The depth of self+children as a *cpip.plot.Coord.Dim* or None in the case that I don't exist and I have no children. i.e. the width of box ****

Returns cpip.plot.Coord.Dim([float, str]) – Width

bbSpaceChildren
The additional distance to give to the children as a *cpip.plot.Coord.Dim*.

Parameters **value** (cpip.plot.Coord.Dim([float, str])) – Spacing between children.

Returns NoneType,cpip.plot.Coord.Dim([float, str]) – Spacing between children.

box
The cpip.plot.Coord.Box of ####.

Returns cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – The box

childBboxDatum(*theDatum*)
The point x[0] above as a *cpip.plot.Coord.Pt* given theDatum as *cpip.plot.Coord.Pt* or None if no children.

Parameters **theDatum** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Datum

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Point x[0]

depth
The immediate depth of the node, if None then no BB depth or bbSpaceChildrend is allocated. i.e. the depth of box ####

Parameters **value** (cpip.plot.Coord.Dim([float, str])) – The depth.

Returns NoneType,cpip.plot.Coord.Dim([float, str]) – The depth.

extendChildBbox(*theChildBbox*)
Extends the child bounding box by the amount theChildBbox which should be a cpip.plot.Coord.Box. This extends the .... line above.

Parameters **theChildBbox** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The additional bounding box.

Returns NoneType

hasSetArea
Returns True if width and depth are set, False otherwise.

numChildren
The number of children.

Returns int – The number of children.

plotPointCentre(*theLd*)
Returns the logical point at the centre of the box shown as #### above.

plotPointSelf(*theDatum*)
The point S as a *cpip.plot.Coord.Pt* given theDatum as *cpip.plot.Coord.Pt*.

Parameters **theDatum** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Datum

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Point S

width
The immediate width of the node, if None then no BB depth or bbSpaceChildrend is allocated. i.e. the depth of box ####

Parameters **value** (cpip.plot.Coord.Dim([float, str])) – The depth.

Returns NoneType,cpip.plot.Coord.Dim([float, str]) – The depth.

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 *cpip.plot.Coord.Pt*.

Parameters

- **theLd** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical datum.
- **childIndex** (int) – Index

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Landing point.

cpRoll(*theLd*)
The me-as-child-to-parent start point given the logical datum as a *cpip.plot.Coord.Pt*.

Parameters **theLd** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical datum.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Start point.

cpStop(*theLd*, *childIndex*)
The me-as-parent-from-child stop point given the logical datum as a *cpip.plot.Coord.Pt*.

Parameters

- **theLd** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical datum.
- **childIndex** (int) – Index

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Stop point.

cpTo(*theLd*)

The me-as-child-to-parent take off point given the logical datum as a *cpip.plot.Coord.Pt*.

Parameters **theLd** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical datum.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Take off point.

pcLand(*theLd*)

The parent-to-me-as-child landing point given the logical datum as a *cpip.plot.Coord.Pt*.

Parameters **theLd** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical datum.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Landing point.

pcRoll(*theDatum*, *childIndex*)

The me-as-parent-to-child logical start point given the logical datum as a *cpip.plot.Coord.Pt* and the child ordinal. This gives equispaced points along the lower edge.

Parameters

- **theDatum** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical datum.
- **childIndex** (int) – Index

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Start point.

pcStop(*theLd*)

The parent-to-me-as-child stop point given the logical datum as a *cpip.plot.Coord.Pt*.

Parameters **theLd** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical datum.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Stop point.

pcTo(*theDatum*, *childIndex*)

The me-as-parent-to-child logical take off point given the logical datum as a *cpip.plot.Coord.Pt* ind the child ordinal. This gives equispaced points along the lower edge.

Parameters

- **theDatum** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical datum.
- **childIndex** (int) – Index

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Take off point.

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 *cpip.plot.Coord.Pt*.

cpRoll(*theDatumL*)
The me-as-child-to-parent start point given the logical datum as a *cpip.plot.Coord.Pt*.

cpStop(*theDatumL*, *childIndex*)
The me-as-parent-from-child stop point given the logical datum as a *cpip.plot.Coord.Pt*.

cpTo(*theDatumL*)
The me-as-child-to-parent take off point given the logical datum as a *cpip.plot.Coord.Pt*.

pcLand(*theDatumL*)
The parent-to-me-as-child landing point given the logical datum as a *cpip.plot.Coord.Pt*.

pcRoll(*theDatumL*, *childIndex*)
The me-as-parent-to-child logical start point given the logical datum as a *cpip.plot.Coord.Pt* ind 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 *cpip.plot.Coord.Pt*.

pcTo(*theDatumL*, *childIndex*)
The me-as-parent-to-child logical take off point given the logical datum as a *cpip.plot.Coord.Pt* ind the child ordinal. This gives equispaced points along the lower edge.

TreePlotTransform

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

__init__(*theXmlStream*, *thePoint*, *theRadius*, *attrs=None*)
Initialise the circle with a stream, a *cpip.plot.Coord.Pt* and a *cpip.plot.Coord.Dim* objects.

Parameters

- **theXmlStream** (*cpip.plot.SVGWriter.SVGWriter*) – The SVG stream.
- **ptFrom** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Starting point.
- **theRadius** – X radius.
- **attrs** (dict({str : [str]})) – Element attributes.

Returns NoneType

class cpip.plot.SVGWriter.**SVGElipse**(*theXmlStream*, *ptFrom*, *theRadX*, *theRadY*, *attrs=None*)
An elipse in SVG. See: http://www.w3.org/TR/2003/REC-SVG11-20030114/shapes.html#EllipseElement

__init__(*theXmlStream*, *ptFrom*, *theRadX*, *theRadY*, *attrs=None*)
Initialise the elipse with a stream, a *cpip.plot.Coord.Pt* and a *cpip.plot.Coord.Dim* objects.

Parameters

- **theXmlStream** (*cpip.plot.SVGWriter.SVGWriter*) – The SVG stream.
- **ptFrom** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Starting point.
- **theRadX** – X radius.
- **theRadY** – Y radius.
- **attrs** (dict({str : [str]})) – Element attributes.

Returns NoneType

class cpip.plot.SVGWriter.**SVGGroup**(*theXmlStream*, *attrs=None*)
See: http://www.w3.org/TR/2003/REC-SVG11-20030114/struct.html#GElement

__init__(*theXmlStream*, *attrs=None*)
Initialise the group with a stream.

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, ...}

Parameters

- **theXmlStream** (*cpip.plot.SVGWriter.SVGWriter*) – The SVG stream.
- **attrs** (dict({str : [<class 'str'>, str]}), dict({str : [str]})) – Element attributes.

Returns NoneType

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

__init__(*theXmlStream*, *ptFrom*, *ptTo*, *attrs=None*)
Initialise the line with a stream, and two *cpip.plot.Coord.Pt* objects.

Parameters

- **theXmlStream** (*cpip.plot.SVGWriter.SVGWriter*) – The SVG stream.
- **ptFrom** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Starting point.
- **ptTo** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Ending point.
- **attrs** (dict({str : [str]})) – Element attributes.

Returns NoneType

class cpip.plot.SVGWriter.**SVGPointList**(*theXmlStream*, *name*, *pointS*, *attrs*)
An abstract class that takes a list of points, derived by polyline and polygon.

__init__(*theXmlStream*, *name*, *pointS*, *attrs*)
Initialise the element with a stream, a name, and a list of *cpip.plot.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.

Parameters

- **theXmlStream** (*cpip.plot.SVGWriter.SVGWriter*) – The SVG stream.
- **name** (NoneType, str) – Element name.
- **pointS** – List of points
- **attrs** (dict({str : [str]})) – Element attributes.

Returns NoneType

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

__init__(*theXmlStream*, *pointS*, *attrs=None*)
Initialise the polygon with a stream, and a list of *cpip.plot.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.

Parameters

- **theXmlStream** (*cpip.plot.SVGWriter.SVGWriter*) – The SVG stream.
- **pointS** – List of points
- **attrs** (dict({str : [str]})) – Element attributes.

Returns NoneType

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

__init__(*theXmlStream*, *pointS*, *attrs=None*)
Initialise the polyline with a stream, and a list of *cpip.plot.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.

Parameters

- **theXmlStream** (*cpip.plot.SVGWriter.SVGWriter*) – The SVG stream.
- **pointS** – List of points
- **attrs** (dict({str : [str]})) – Element attributes.

Returns NoneType

class cpip.plot.SVGWriter.**SVGRect**(*theXmlStream*, *thePoint*, *theBox*, *attrs=None*)
See: http://www.w3.org/TR/2003/REC-SVG11-20030114/shapes.html#RectElement

__init__(*theXmlStream*, *thePoint*, *theBox*, *attrs=None*)
Initialise the rectangle with a stream, a *cpip.plot.Coord.Pt* and a cpip.plot.Coord.Box objects.

Typical attributes:

```
{'fill' : "blue", 'stroke' : "black", 'stroke-width' : "2", }
```

<insert documentation for function>

Parameters

- **theXmlStream** (*cpip.plot.SVGWriter.SVGWriter*) – The SVG stream.
- **thePoint** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Starting point.
- **theBox** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), <class 'cpip.plot.Coord.Dim'>]), cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]), cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([int, <class 'str'>])]), cpip.plot.Coord.Box([cpip.plot.Coord.Dim([int, str]), cpip.plot.Coord.Dim([int, <class 'str'>])])) – The box.
- **attrs** (dict({str : [<class 'str'>, str]}), dict({str : [str]})) – Element attributes.

Returns NoneType

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

__init__(*theXmlStream*, *thePoint*, *theFont*, *theSize*, *attrs=None*)
Initialise the text with a stream, a *cpip.plot.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.

Parameters

- **theXmlStream** (*cpip.plot.SVGWriter.SVGWriter*) – The SVG stream.
- **thePoint** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Place to write the text.
- **theFont** (NoneType, str) – Font.
- **theSize** (NoneType, int) – Size
- **attrs** (dict({str : [str]})) – Element attributes.

Returns NoneType

class cpip.plot.SVGWriter.**SVGWriter**(*theFile*, *theViewPort*, *rootAttrs=None*, *mustIndent=True*)
An XML writer specialised for writing SVG.

__enter__()
Context management.

Returns cpip.plot.SVGWriter.SVGWriter – <insert documentation for return values>

__init__(*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.

Parameters

- **theFile** (_io.TextIOWrapper) – Output file.
- **theViewPort** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([int, str]), cpip.plot.Coord.Dim([int, <class 'str'>])])) – Viewport.
- **rootAttrs** (dict({str : [str]})) – Root element attributes.

- **mustIndent** (bool) – Indent elements or write them inline.

Returns NoneType

cpip.plot.SVGWriter.**dimToTxt**(*theDim*)
Converts a Coord.Dim() object to text for SVG units.

Examples:

`(2.0 / 3.0, 'in')` becomes `'0.667in'`

`(2.0 / 3.0, 'mm')` becomes `'0.7mm'`

`(23.123, 'px')` becomes `'23px'`

Parameters **theDim** (cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([int, str])) – The dimension

Returns str – Text suitable for writing SVG, for example '0.667in'.

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.

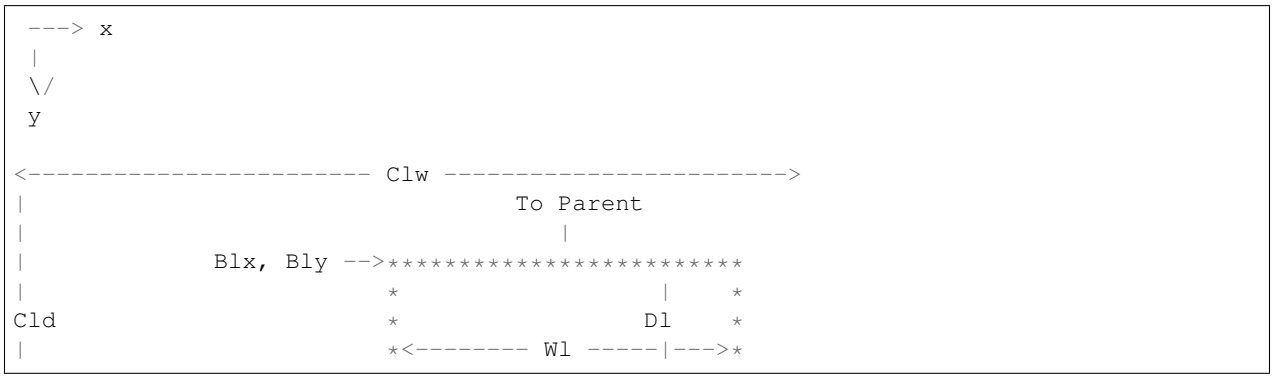
Transforming sizes and positions

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.



```
|                          *                   |    *
|       Plx, Ply ->.       *                   |    *
|                          ************************
|                              |         |        |
|                         To C[0]  To C[c]   To C[C-1]
```

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.

__init__(*theLogicalCanvas*, *rootPos='top'*, *sweepDir='-'*)
Constructor, takes a 'logical' Canvas as a Coord.Box and the orientation.

Parameters

- **theLogicalCanvas** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([int, str]), cpip.plot.Coord.Dim([int, <class 'str'>])])) – Teh logical canvas to draw on.
- **rootPos** (str) – Root position, one of ('top', 'right', bottom', 'left').
- **sweepDir** (str) – Sweep direction, one of ('-', '+').

Returns NoneType

__weakref__
list of weak references to the object (if defined)

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").

Parameters

- **theBlxy** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical point.
- **theBl** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The box.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Box datum.

boxP(*theBl*)
Given a logical box this returns a cpip.plot.Coord.Box that describes the physical box.

Parameters **theBl** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical box.

Returns cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Physical box.

canvasP()
Returns a Coord.Box that describes the physical canvass.

Returns cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – The canvas.

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.

Parameters

- **theBlxy** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Datum
- **theBl** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The box.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Next dead centre of the box.

positiveSweepDir
True if positive sweep, false otherwise.

postIncChildLogicalPos (*thePt*, *theBox*)
Post-increments the child logical datum point ('top-left') given the child logical datum point and the child.bbSigma. Returns a *cpip.plot.Coord.Pt*. This takes into account the sweep direction.

Parameters

- **thePt** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The logical point.
- **theBox** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The canvas.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – The 'physical' point.

preIncChildLogicalPos (*thePt*, *theBox*)
Pre-increments the child logical datum point ('top-left') given the child logical datum point and the child.bbSigma. Returns a *cpip.plot.Coord.Pt*. This takes into account the sweep direction.

Parameters

- **thePt** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The logical point.
- **theBox** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The canvas.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – The 'physical' point.

prevdcL (*theBlxy*, *theBl*)
Given a logical datum (logical top left) and a logical box this returns logical 'previous' dead centre of a box.

Parameters

- **theBlxy** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Datum
- **theBl** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The box.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Previous dead centre of the 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.

Parameters

- **thePt** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Logical point.
- **units** (NoneType) – Optional unit conversion.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Physical Point.

startChildrenLogicalPos (*thePt*, *theBox*)

Returns the starting child logical datum point ('top-left') given the children logical datum point and the children.bbSigma. Returns a *cpip.plot.Coord.Pt*. This takes into account the sweep direction.

Parameters

- **thePt** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The logical point.
- **theBox** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The canvas.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – The 'physical' point.

tdcL (*theBlxy*, *theBl*)

Given a logical datum (logical top left) and a logical box this returns logical top dead centre of a box.

Parameters

- **theBlxy** (cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – Datum
- **theBl** (cpip.plot.Coord.Box([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])])) – The box.

Returns cpip.plot.Coord.Pt([cpip.plot.Coord.Dim([float, str]), cpip.plot.Coord.Dim([float, <class 'str'>])]) – Top dead centre of the box.

tdcP (*theBlxy*, *theBl*)

Given a logical datum (logical top left) and a logical box this returns physical top dead centre of a box.

CHAPTER 9

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 undestanding file dependencies.

CHAPTER 10

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

10.1 Types of Contributions

10.1.1 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.

10.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

10.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

10.1.4 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.

10.1.5 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 :)

10.2 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 https://github.com/paulross/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.

10.3 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 2.7, 3.3 to 3.6, and for PyPy. Check https://travis-ci.org/paulross/cpip/pull_requests and make sure that the tests pass for all supported Python versions.

10.4 Tips

To run a subset of tests, for example to test `PpLexer.py`:

```
$ pytest -vs tests/unit/test_core/test_PpLexer.py
```

10.5 Release Checklist

In the following example the version we are moving to, in `Major.Minor.Patch` format, is `0.9.7`.

Current version should be something like `M.m.(p)rcX`, for example `0.9.7rc4`.

10.5.1 Increment version

Change the version to `M.m.p` in these places:

- *setup.cfg*
- *setup.py*
- *src/cpip/CPIPmain.py*
- *src/cpip/__init__.py*

In *src/cpip/__init__.py* change `CPIP_VERSION = (0, 9, 7)`

Update the history:

- *HISTORY.rst*
- *src/cpip/__init__.py*

Update any Trove classifiers in *setup.py*, https://pypi.python.org/pypi?%3Aaction=list_classifiers

10.5.2 Build and Test

Build and test for Python 3.6:

```
$ . ~/venvs/CPIP36/bin/activate
(CPIP36) $ python setup.py install
(CPIP36) $ python setup.py test
```

And for Python 2.7

```
$ . ~/venvs/CPIP27/bin/activate
(CPIP27) $ python setup.py install
(CPIP27) $ python setup.py test
```

Build the docs HTML to test them, from an environment that has Sphinx:

```
(Sphinx) $ cd docs
(Sphinx) $ make clean
(Sphinx) $ make html
```

10.5.3 Commit and Tag

Commit, tag and push:

```
$ git add .
$ git commit -m 'Release version 0.9.7'
$ git tag -a v0.9.7 -m 'Version 0.9.7'
$ git push
$ git push origin v0.9.7
```

10.5.4 PyPi

Prepare release to PyPi for Python 3.6:

Build the egg and the source distribution:

```
(CPIP36) $ python setup.py install sdist
```

And for Python 2.7

```
(CPIP27) $ python setup.py install
```

Check the contents of `dist/*`, unpack into `tmp/` if you want:

```
$ cp dist/* tmp/
$ cd tmp/
$ unzip cpip-0.9.7-py2.7.egg -d py27egg
$ unzip cpip-0.9.7-py3.6.egg -d py36egg
$ tar -xzf cpip-0.9.7.tar.gz
```

Release to PyPi, https://pypi.python.org/pypi/cpip:

```
(CPIP36) $ twine upload dist/*
```

10.5.5 ReadTheDocs

Build the documentation: https://readthedocs.org/projects/cpip/builds/

10.5.6 Prepare Next Release Candidate

Finally change the version to `M.m.(p+1)rc0`, in this example `0.9.8rc0` in these places:

- *setup.cfg*
- *setup.py*
- *src/cpip/CPIPmain.py*
- *src/cpip/__init__.py*, also change `CPIP_VERSION = (0, 9, 8, 'rc0')`

Commit and push:

```
$ git add .
$ git commit -m 'Release candidate v0.9.8rc0'
$ git push
```

CHAPTER 11

Credits

11.1 Development Lead

- Paul Ross <apaulross@gmail.com>

11.2 Contributors

None yet. Why not be the first?

CHAPTER 12

History

12.1 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.

12.2 0.9.5 Beta Release (2017-10-03)

- Migrate from sourceforge to GitHub.

12.3 0.9.1 (2014-09-03)

Version 0.9.1, various minor fixes. Tested on Python 2.7 and 3.3.

12.4 Alpha Plus Release (2014-09-04)

Fairly thorough refactor. CPIP now tested on Python 2.7, 3.3. Version 0.9.1. Updated documentation.

12.5 Alpha Release (2012-03-25)

Very little functional change. CPIP now tested on Python 2.6, 2.7, 3.2. Added loads of documentation.

12.6 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

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z