PIPS

Developer Guide

Corinne Ancourt
Fabien Coelho
Béatrice Creusillet
Serge Guelton
François Irigoin
Pierre Jouvelot
Ronan Keryell
Arnauld Leservot
Alexis Platonoff
Rémi Triolet
... and so many other contributors...

August 1996 – May 15, 2024

`Id: developer_guide.tex 23065 2016-03-02 09:05:50Z coelho`

You can get a printable version of this document on `http://www.cri.ensmp.fr/pips/developer_guide.htdoc/developer_guide.pdf` and a HTML version on `http://www.cri.ensmp.fr/pips/developer_guide.htdoc`. Warning if you use the HTML version and do some copy/paste, it has been noticed that some characters are changed, such as the `'` that should be a simple plain quote but is changed in something else that looks like a normal quote but is not a normal quote...

The PIPS WWW site is at `http://pips4u.org` .

Contents

1 Introduction

This document aims at presenting PIPS development environment. It is not linearly organized: PIPS is made of several, sometimes interdependent, components. This paper thus begins with a presentation of PIPS directories. Then the shell environment is described. The next two chapters are devoted to two external tools on which PIPS relies: NewGen and the Linear library. Section 7 will then present PIPS make file policy. Sections 11 and 16 are devoted to PIPS libraries and passes. The next section briefly describes some conventions usually respected when developing in PIPS. The last two sections describe the *bug policy* of PIPS and some save and restore information.

This manual is not exhaustive. You can add your own sections and update existing ones if you find missing or erroneous information.

The reader is supposed to be a PIPS user [3], and to have read the reports about NewGen [1, 2]. A good understanding of `pipsmake` mechanisms would also be helpful.

2 Getting PIPS sources

The sources of NewGen, Linear and Pips are managed under subversion (svn). They are accessible from anywhere in the world by the http protocol at `https://scm.cri.ensmp.fr/pips.html`

2.1 Directories overview

There are 5 repositories for the various files:

nlpmake common makefiles for Newgen, Linear and Pips.

newgen Newgen software.

linear Linear/C3 mathematical libraries.

pips PIPS software.

validation Pips non-regression tests

There is also *private* validation directory on another server.

The subversion repositories are organised in standard subdirectories as advised for best subversion practices:

trunk production version, should be stable enough to pass the validation;

branches development branches of all or part of the software for each developer. Developments are to be performed here and installed (joined, merged) once finished into `trunk`;

tags tagged revisions. Note these tags are not related with the tags described in section 3.4.

Moreover the `pips` repository includes

bundles: group of softwares;

bundles/trunks: convenient extraction at once of all the 3 softwares trunk versions ready for compilation.

2.2 Building PIPS

A crude script `setup_pips.sh` on the web site (`https://scm.cri.ensmp.fr/svn/nlpmake/trunk/makes/setup_pips.sh`) and in `pips/trunk/makes` allows to download `polylib`, `newgen`, `linear` and `pips` and to build a local installation of the softwares. For instance developer `calvin` can do the following to setup its own PIPS environment:

```
sh> setup_pips.sh /home/temp/MYPIPS calvin
...
sh> source /home/temp/MYPIPS/pipsrc.sh
sh> # enjoy
```

Before you *enjoy* PIPS, do not worry too much about the many error messages that are displayed: the header files are being recomputed from the C files. After a while, C files should be compiled without errors.

In order to rebuild only the PIPS infrastructure, just type `make` into the `$PIPS_ROOT` directory. If something has gone wrong, for instance because a software component is missing or outdated, it is advised to restart from scratch by runing `make unbuild` before trying again.

For more information about the PIPS Makefile infrastructure, see Section 3.6.4.

2.3 Important note

It is important to realize that:

1. PIPS build is much more efficiently if it is stored on local disks rather than on remote directories accessed through the network (*e.g.* via NFS). Hence the `/home/temp` directory choice in the above axample.

2. the local copy editions are not saved unless commits are performed.

3. in a standard development, commits should not be made directly under the `trunk`, but rather in development branches, see the next section.

4. on some Unises, the `/tmp` temporary directories may be cleaned on reboots, which can be triggered by power failures. So you could try `/var/tmp` that is usually not cleaned-up during the boot phase.

2.4 Compiling a specialized version of PIPS

A specialized version of PIPS, which contains only a subset of passes, can be built with the following procedure.

For building a special PIPS `foo` version:

- First, in `src/Documentation/pipsmake` create a basic ASCII `foo.conf` file which includes the list of passes that the specialized version of PIPS must contain, one pass name per line. `#`-comment lines can be used.

  Note that this list must be exact: other passes, even if compiled in, will not be available.

- At the root of sources, type

  `make PIPS_SPECIAL=foo compile`.

  Alternatively, the `PIPS_SPECIAL` environment variable can be set in order to work constantly on a specialized version of PIPS.

  PIPS executables are built with the libraries which contain the required passes, and their dependencies.

  For a very minimal example, try with predefined configurations `cmin` or `fmin` to generate a C or Fortran parser and prettyprinter.

- Enjoy!

  When running `tpips --version`, the name of the version is printed after the string `special:`.

  When PIPS is compiled with every passes, the special name is `full`.

The detailed of the specialized version generation are as follow:

- The compilation first generates from `foo.conf` a new `foo.deps` file which for each phase associates its containing library. This is done by the simple `pass2lib.sh`. As this script relies on basic *grep* processing, the phases must be declared simply on one line to be found:

  ```
  bool my_phase_name(const string module)
  ```

- Then a `foo.libs` is generated from both files with `pass2libs.pl`: the script performs a transitive closure on library dependencies. The dependencies are hardcoded within the script.

  Note that this dependencies are not that simple: First, some dependencies may be hidden, that is no include is used in the sources, but there is a direct `extern` declaration. Second, some libraries depend on another but may be compiled without the dependence in some cases, thanks to macros.

  This file is directly used both for defining the list a library to use and to know the directories which must be explore when compiling PIPS libraries, so that only actually used libraries are compiled.

- From this file a new `pips-libs.h` file is also generated which defines macros for each library which is included in the built.

  These macros are used to exclude some includes and functions when compiling some libraries.

- The generated `pipsmake.rc`, `phases.h`, `builder_map.h` files only includes passes defined in the `foo.conf` file.

- The generated `resources.h` and `printable_resources.h` files are not filtered, but currently includes all possible resources.

3 Developing environment

3.1 Browsing and documenting the source files with Doxygen

To help digging into the sources, Doxygen is used to generate an interactive online version of the sources that can be seen at `http://doxygen.pips.enstb.org/PIPS/graph`. The call and caller graphs are useful to figure out what are the functions used, and more subtly what are the functions that call a given function (if you need to verify a function is correctly used or you want to change a function name).

As an experimented user, you may need to generate them again. The `doxygen` *make* target is propagated downwards down to the various `doxygen` directories in the products (NewGen, Linear and PIPS).

To push them on the WWW, you can use a `make doxygen-publish`, if you have the right access.

The `make` target `doxygen-plain` or `doxygen-graph` can be used to generate only a documentation without or with call and caller graphs. Note that generating the graph version for PIPS lasts several hours on a 2009 computer...

Of course, developers in PIPS should use the Doxygen syntax to have a more useful documentation.

Modules and submodules in Doxygen are quite useful to group related objects and structure concepts in the documentation. For an example, have a look to `@defgroup`, `@addtogroup`, `@{` and `@}` in `newgen/src/genC/genClib.c` and the resulting module *NewGen quick and intelligent recursion on objects (visitor pattern)* and its submodules in the documentation. Be careful: a title group must be on the same line and cannot be split !

A good practice should be to present main useful functions and concepts to know by a PIPS programmer as modules and submodules.

3.2 Developing with Eclipse

Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system.

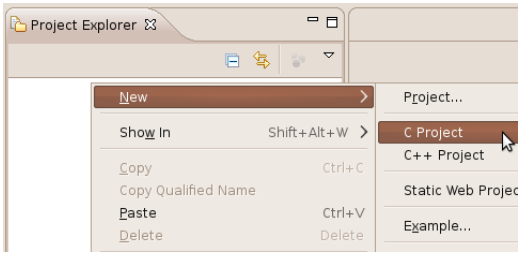
You have to use an eclipse installation with CDT plugin (have a look at `http://www.eclipse.org/cdt`) to be able to use eclipse for C/C++ development.

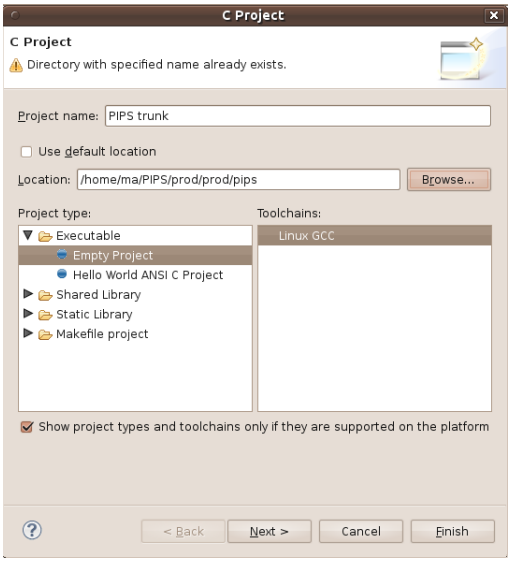
3.2.1 Workspace/Project creation

In Eclipse you have workspace and project concept. The latter is like a repository, or a subtree of a source location. In PIPS it can be a pass, whole pips or linear or newgen, or all of them in one shot !

A workspace is a set of projects that are usually related, loaded in memory, and indexed while working. For instance if you want Eclipse to provide some navigation features in newgen source code while working on PIPS project, you need to have both project in the same workspace.

The creation of a project in a workspace is easy and suppose that you already have PIPS sources somewhere on you hard drive.



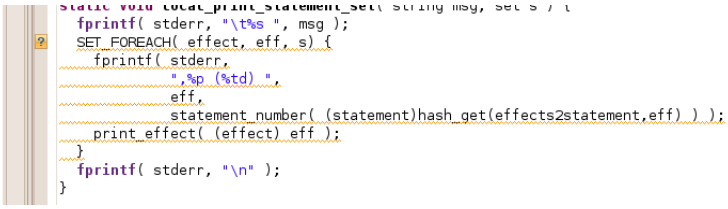


First use menu *File => New => Project* or right-click and contextual menu as shown on the screenshot, then give a project name, uncheck the checkbox to enable browsing an existing location, and find your PIPS source directory. Click finish and you have now a fresh new PIPS project in Eclipse. It may overload your computer for a little while indexing the whole project, you can see the progression in the status bar down the Eclipse window. Index can be rebuild at any time using right click on project root.

You should really repeat this operation for Newgen and Linear project.

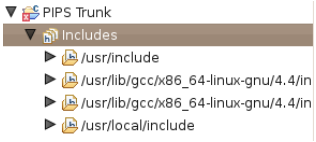
3.2.2 Missing includes

Try to open some source files, for instance src/Libs/chains/chains.c and scroll a little. You'll find some part of the underlined by Eclipse. Eclipse just like your favorite compiler detect undefined symbol and report it to you in a friendly way.

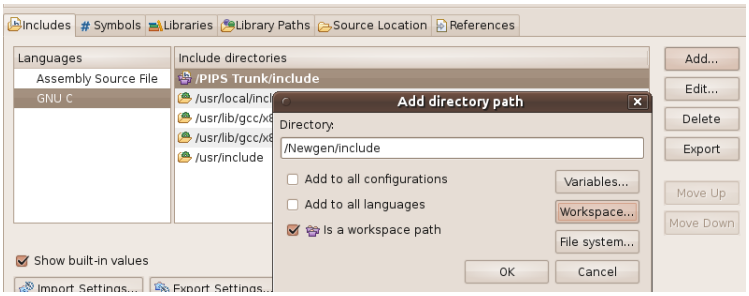


```
  fprintf( stderr, "\t%s ", msg );
  SET_FOREACH( effect, eff, s) {
    fprintf( stderr,
             ",%p (%td) ",
             eff,
             statement_number( (statement)hash_get(effects2statement,eff) ) );
    print_effect( (effect) eff );
  }
  fprintf( stderr, "\n" );
}
```

We are missing Newgen and Linear includes, we have only default Eclipse includes :



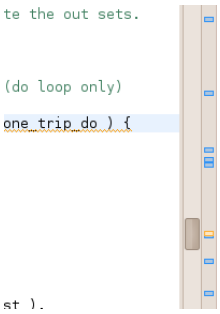
Select project root and push *alt+enter* or right click on it and select properties item in the contextual menu. Then go to C/C++ General submenu and select Paths and Symbols. In the Includes tab select language GNU C in the left part and click add. Then choose *Workspace...* and find you `pips/include/` dir. Repeat the same operation for including Newgen and Linear include dirs.



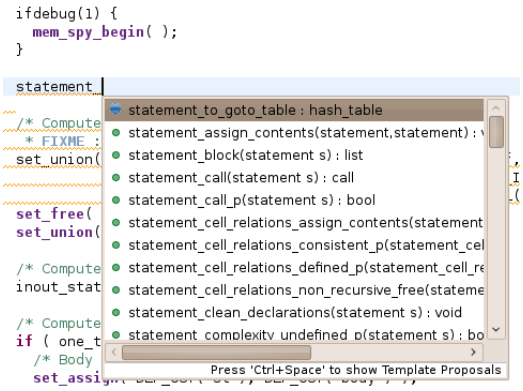
Validate two times and accept to rebuild the index when Eclipse ask. After index rebuilding you can observe that the code is no longer underlined. :-)

3.2.3 Benefits from Eclipse power

Eclipse provide many features to help you becoming more productive. Firstly is the vertical bar just right to the code :



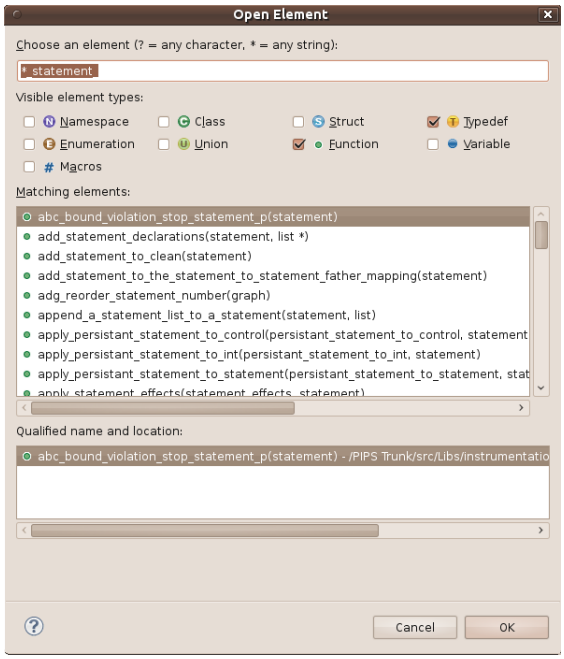
Clicking on this bar make you move inside a source file quickly, there is kind of colored *post it* that are shortcut. Color informs you from what's going on at this point of the source. For instance red is an error, white is occurence of currently focused variable (very useful !), and blue are occurence of *FIXME* stickers in comments (use it !). Yes I know that there's a lot of blue post it in chains ;-)



Eclipse provide you some completion capabilities, try to write the begin of a symbol at some place in the code and hit *ctrl+enter*

You can easily find the definition of a symbol, even if it's located in another source file or even another project. The magic key is F3, try it on *set_make* for instance, the first press will bring you to the header in Newgen/include and the second press will bring you to the definition of the function. You can always go back using *alt+left arrow*

Another very pleasant feature is the lookup windows, which you can get using *ctrl+alt+t* or menu *Navigate => Open element*. You can use the star as a joker for searching symbol by their name.



PIPS/Newgen make high usage of macro, Eclipse has a powerful macro expansion step-by-step. You can have a quick view by moving your pointer over the macro. Or you can interactively develop the macro with a right click on it and *Explore Macro Expansion* or pressing *ctrl+=* after having selected it.

ctrl+shift+g is one of my favorite feature :-). Used on a function it'll show you every call site in the workspace !

Others entries in the menu *Navigate* provide also intersting features, try them !

Eclipse provide also some refactoring power, use the right click or the menu and select *Rename...*. You can rename a variable so as a function ! Every usage and declaration in the workspace will be updated. Obviously, it'll be longer for a widely used newgen function than for a static function !

3.2.4 Using team working plugin, aka. SVN/GIT for Eclipse

TODO

3.2.5 Building from Eclipse

TODO

3.3 Developing with the Emacs editor

There are many modes in Emacs that can help developments in PIPS.

To compile the sources you can use the menu `Tools/Compile...`. It presents errors in red and if you click on them you jump to the sources at the right location.

The check-compilation-errors-on-the-fly is quite fun (it is equivalent in programming to the on-the-fly spell checker `flyspell-mode`). PIPS makefiles support it. Try it with `M-x flymake-mode`.

The `speedbar` mode helps with a navigation window. More descrete, to have access from the menu to functions and other symbols in the local source file, test the `imenu-add-to-menubar` Emacs function.

You can use tags to index source files as explained in section 3.4.

Most PIPS contributors use Emacs to develop in PIPS, so some goodies have been added to ease those developers.

Have a look for example at section 9 for debugging. You should use the `gud-tooltip-modes` to display the value of an object under the mouse. Setting the `gdb-many-windows` to `t` transform Emacs (from version 22) in an interesting graphical IDE. You can select a frame by just clicking it and so on.

Most documentation (like the one you are reading) is written in LaTeX and some parts even use literate programming with a LaTeX base, so the AUCTeX, PreviewLaTeX and RefTeX mode are quite useful.

3.4 Using tags to index source files and to ease access in your text editor

Many macros and functions are available from the Linear and NewGen libraries, but also in PIPS. Their source code can be retrieved using the *tags* indexing mechanism under editors such as **emacs** or **vi**. You should notice that this notion of tags is unrelated to the tags in a version control systems as exposed in § 3.6.3.

So the PIPS makefiles can generate tags index files to find the definition of symbols in the source files with your favorite editor and it is highly recommended

to use the Emacs or VI editors to benefit from the tags. For example in Emacs, you can find, by typing a `M-.` on an identifier, the location where it is defined, `M-,` allows to find another location. Note that completion with `TAB` works with tag searching too. With **vi**, you can use `C-]` to jump on a definition of an identifier.

PIPS can produce tags files for VI by invoking `make CTAGS` or tags for Emacs by invoking `make TAGS` in the PIPS `prod` top directory and it will produce the tags files into the `linear`, `newgen` and `pips` subdirectories.

To build the tags for both VI and Emacs, just use a simple `make tags`. The tags files at the PIPS top directories need to be remade regularly to have an up to date version of the index. should be used for **vi** users.

You can also build specifically the tags for a development version of PIPS for example by making these tags explicitly in the `newgen/trunk`, `pips/trunk` and `linear/trunk` directories.

Then you need to configure your editor to use these tags files, for example in Emacs by using the customization of the `Tags Table List` with the `Options/Customize Emacs` menu or more directly in you `.emacs` with stuff like:

- for a production version of the tags :

```
(custom-set-variables
  ;; ...
 '(tags-table-list (quote
    ("/home/keryell/projets/PIPS/MYPIPS/prod/newgen/TAGS"
     "/home/keryell/projets/PIPS/MYPIPS/prod/linear/TAGS"
     "/home/keryell/projets/PIPS/MYPIPS/prod/pips/TAGS")))
 )
```

- for a development version of the tags :

```
(custom-set-variables
  ;; ...
 '(tags-table-list (quote
    ("/home/keryell/projets/PIPS/svn/newgen/trunk/TAGS"
     "/home/keryell/projets/PIPS/svn/linear/trunk/TAGS"
     "/home/keryell/projets/PIPS/svn/pips/trunk/TAGS")))
 )
```

3.5 Using cscope to navigate source file

In a way similar to `ctags`, `make cscope` [1] creates a database suitable for use with the `cscope` tool. This tool make it possible to find callers, definition or declaration of a function, enabling advanced code navigation not possible with tags.

Here are the relevant lines for a `vim` configuration, to be added to `.vim/ftplugin/c.vim`

[1] `make cscope.out` for nlpmake users

```
" cscope
if has("cscope")
	set csto=0
	set cst
	set nocsverb
	" add any database in current directory
	if filereadable("cscope.out")
		cs add cscope.out
		" else add database pointed to by environment
	elseif $CSCOPE_DB != ""
		cs add $CSCOPE_DB
	endif
	set csverb
endif
```

3.6 Developing PIPS under SVN

A basic understanding of subversion operations and concepts such as *repository, checkout, status, commit* are necessary to develop in PIPS. See for instance the SVN book at `http://svnbook.red-bean.com/` and other resources on the Internet. This section assumes that you have a recent version of svn, version **1.5** or above, as these versions greatly improve the management of branches.

The development policy within the PIPS-related software is that the `trunk` version should hold a production quality software which should pass all non-regression tests that run daily at CRI and elsewhere. Thus day-to-day development should not occur within this part, but in independent per-developer `branches` which should be validated before being merged back into the `trunk`.

SVN can manage properties about files which help portability by declaring the end-of-line policy, the mime-type and so on. We recommand that you add the following to your `.subversion/config` file:

```
# in section [miscellany]
enable-auto-props = yes

[auto-props]
# pips-related props
*.f = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-fortran
*.F = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-fortran
*.f77 = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-fortran
*.f90 = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-fortran
*.c = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-c
*.h = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-c
*.sh = svn:eol-style=native;svn:executable;svn:keywords=Id URL;svn:mime-type=text/x-sh
*.pl = svn:eol-style=native;svn:executable;svn:keywords=Id URL;svn:mime-type=text/x-script.perl
Makefile = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-make
*.mk = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-make
*.tex = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-tex
*.bib = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-bibtex
*.txt = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/plain
*.py = svn:eol-style=native;svn:keywords=Id;svn:mime-type=text/x-python
*.tpips = svn:eol-style=native;svn:mime-type=text/x-tpips
# and other property settings that you could find useful
*.png = svn:mime-type=image/png
*.jpg = svn:mime-type=image/jpeg
*.csv = svn:eol-style=native;svn:mime-type=text/csv
*.sql = svn:eol-style=native;svn:keywords=Id URL;svn:mime-type=text/x-sql
```

3.6.1 PIPS development branches

If you are doing something non trivial or do not have commit rights on pips' trunk, you shoud develop in a branch of your own. To do so:

1. Obtain an svn user account on `scm.cri.ensmp.fr` by asking for it to Fabien Coelho.

   For instance, a login `calvin` and some password may be attributed. Calvin will have his own development area under `branches/calvin` in the subversion repository.

   The developer branch is private by default. It means that what you do in the branch is your own business, and no message is sent to the world on commits. For interns, the default policy is to send a message to the intern and the developer(s) responsible for his/her work.

   For HPC-Project related accounts, branch commits are send to the corresponding list.

   Please do commit your work at least once a day, even if it does not work yet!

2. Extract your own PIPS with the setup script (`setup_pips.sh`) mentionned above. The resulting directory may contain at the end of the building process:

   prod compiled production version of `pips`, `newgen`, `linear` and the external `polylib` library;

   pips_dev Calvin's private development area which points to `branches/calvin` and should be empty at the beginning; It can be created afterwards with a manual checkout if necessary in your `MYPIPS` directory (or wherever you want) later with:

   ```
   sh> svn co https://scm.cri.ensmp.fr/svn/pips/branches/calvin pips_dev
   ```

   Branches may also be used for *linear* and *newgen* development, in which case the local directory name for the checkout should be called `linear_dev` or `newgen_dev` respectively.

   validation a working copy of pips non regression tests for validation. See Sections 3.6.2 & 8.1 for informations about the validation, and then really read the `README` file to run it.

   pipsrc.sh sh-compatible shell initialization;

   pipsrc.csh csh-compatible shell initialization.

3. Create your own development branch for all PIPS:

   ```
   sh> cd pips_dev
   sh> svn cp https://scm.cri.ensmp.fr/svn/pips/trunk my-branch-name
   sh> svn commit my-branch-name
   ```

   It creates a new branch as a working copy, which is then committed to the repository.

   The idea is to branch a whole PIPS copy since it ensures that the developments you do in many places in PIPS (even at some place you would not have expected when you created the branch!) can be committed atomically.

   CAUTION! take care to create branches only for the *trunk*. Creating branches for a subdirectory breaks svn bookkeeping.

4. Develop, test, commit within your branch...

   CAUTION! When testing, check that the `PIPS_ROOT` environment variable points to your development branch instead of the trunk. You may also check for the right settings for `EXTERN_ROOT` `NEWGEN_ROOT` and `LINEAR_ROOT` variables, as well as your `PATH` to ensure that the right executable is used. Have a look to § 5 for more information.

5. To merge into your branch on-going developments by other people from trunk, first you need to have already committed your branch and then, in your branch (and no other directory above or under):

   ```
   sh> cd calvin/my-branch-name
   # the above cd is useful: it aims at going into the branch root,
   # i.e. the copy of the "trunk", before merging.
   sh> svn merge ^/trunk
   # this does the merging into the branch working copy
   # - consider options -x -b or -x -w
   #   to ignore space-related changes
   # - if there are conflicts, choose 'p' (postpone)
   #   and deal with them later.
   sh> svn status
   # check for conflicts (C state) or other issues, and resolve them.
   # CAUTION: DO NOT commit in a subdirectory of your branch root
   sh> svn commit
   ```

   If you change your initial idea, you can always revert back to your original version *if it was committed before* with a:

   ```
   svn revert -R .
   ```

   Well, it does not remove the new files that may be created previously in your local copy by the merge procedure, it only delete their meta-data from the subversion point of view. So the cleaner way is to merge into a new working copy of the trunk so that if things go wrong you can simply delete the working copy.

   To know about where you are compared to the trunk, you can try a:

   ```
   svn diff ^/branches/calvin/my-branch-name ^/trunk
   ```

   CAUTION! Merging must be done at the root of the branch, that is the directory which correspond to the `trunk`. **DO NOT** merge in a subdirectory as it breaks svn bookkeeping for subsequent merges, and will result in tree numerous conflicts, which when resolved may be rejected by the pre-commit hook.

6. When you are done with your developments, they are committed into your branch, you want to install them back to the pips trunk. You must have write access to the `trunk` in order to do the commit. First, you must bring your branch up-to-date with respect to the `trunk`, which is the `merge` described above. Then you can reintegrate your changes to the `trunk`:

```
sh> cd prod/pips  # cd in your pips trunk copy, or better a fresh copy
sh> svn mergeinfo \
  ^/branches/calvin/my-branch-name \
  --show-revs eligible
# revisions that need be reintegrated...
sh> svn merge \
  ^/branches/calvin/my-branch-name \
  --reintegrate
# consider options -x -b or -x -w
# to ignore space-related changes
# there should be no issue found on a reintegration.
sh> svn status
# the status or diff should reflect the branch changes.
# the ' M .' reflects the changes of branch bookkeeping informations.
# DO NOT EDIT ANY OF THE FILE
# DO NOT COMMIT FILES SEPARATELY
# you may need to do additional "svn up" if there are external
# commits between the merge & its commit.
sh> svn commit
```

CAUTION! it is important to merge and commit at the `trunk` level, all in one go. If the merging or commit is done on a subdirectory, it breaks svn merge bookkeeping used my the automatic mergings and reintegration because `svn:mergeinfo` properties are added on these subdirectories: when set, they must be manually removed in order to fix.

7. When your are fully done with your branch, you may remove it.

```
sh> cd pips_dev
sh> svn remove my-branch-name
sh> svn commit
```

3.6.2 Developing in a branch? Validate before merging!

Do not forget that if you or someone else merges your branch into the trunk, things must be compiled with an up-to-date copy of all the PIPS environment and validated first against the last global PIPS validation to avoid havoc for the other PIPS developers.

Here is a typical behaviour you should follow:

1. update your production copy of PIPS and all (the `prod`) with `svn up`, recompile the production version and apply the validation against it to have a reference;
2. do/correct your development;
3. compile and validate[2];
4. if it does not work, go to point 2

[2]If you work in a localized part of PIPS, it may be interesting to first run a reduced validation, such as if you work in the C parser, it is obvious that at least the C parser should work so you can run, instead a full validation, only a `make TARGET=C_syntax validate`. Of course, if it pass the validation, do not forget to run a full validation afterwards.

5. merge the `trunk` into your branch to get synchronized with the PIPS outside world;
6. commit your branch saying you have merged the universe into your branch[3];
7. compile and validate;
8. if it does not work, go to point 2;
9. if you are happy with the validation, merge your beautiful work into the PIPS Holly trunk with `svn merge --reintegrate` (§ 3.6.1) so that all the PIPS community can congratulate you;
10. if the merge fails because someone insidiously committed her valuable work just since your last synchronization from the trunk, go to step 5, else you have to delete your branch, it can't be use anymore after having been reintegrated !

For more information about the validation process, have a look to § 8.1. Since your validation mileage may vary according to your target architecture because of bugs or limitations (little or big Endian, native 32-bit or 64 bit architecture...) you should try to validate on various architectures.

3.6.3 Creating FULL branches or tags

The `pips_branch` script creates a full branch or tag of the same name in all 5 repositories (makefiles, 3 softwares, validation), and a *bundle* entry to extract all these files at once. The script needs full write permissions to all five repositories, thus it is restricted to core developers. The help option provides help.

To create a `foo/bla` named branch from trunk try:

```
sh> pips_branch -v -v trunk branches foo/bla
```

It shows the various `svn` commands which are going to be issued. If you are happy with them, do that again with the `-D` option to actually run them. The created *bundle* can be extracted with:

```
sh> svn co https://scm.cri.ensmp.fr/svn/pips/bundles/foo/bla
extracting newgen... linear... pips... validation... nlpmake...
```

Note that the `-R` options allows to fix the revision number to consider from each repository, instead of using the head revision.

3.6.4 Understanding the Makefile infrastructure inside the SVN infrastructure

This section is to be skipped at first reading and is only of interest for people wanting to modify more globally the PIPS infrastructure.

[3]You commit to avoid having in your working copy both modifications from the trunk and further eventual corrections you will do to have things working, which is a bad practice from a tracking point of view. Between 2 commit points it is better to have only one kind of modification so you can easily see what has changed, when and who is guilty...

The compilation directories are managed by some global variables initialized as described in section 5.

If you compile into a directory, in the `prod` directory installed by the installation script, your branch directory or whatever, the compiled stuff from a directory goes to the corresponding `bin lib` directories next to the `src` directory. The *Makefile* definitions are taken from the corresponding `makes` directory next to the `src` sources.

For example, if you have a local PIPS copy in the `svn/pips/trunk` and you compile with `make`, it will build against `$NEWGEN_ROOT` and `$LINEAR_ROOT`, it will install into `bin` and will use *Makefile* stuff from `makes`. So if you want to modify the *Makefile* infrastructure, modify the `makes` directory and commit. It will be committed into the `svn/nlpmake/trunk` repository anyway.

There is also another more modern build infrastructure you may use instead of this one that is described in the companion document of this one that describes the new `autotools` build infrastructure: `http://www.cri.ensmp.fr/pips/auto_pips.htdoc/auto_pips.pdf` and `http://www.cri.ensmp.fr/pips/auto_pips.htdoc`

3.7 The nomadic developer

The SVN centralized development model assumes that you are connected to the repository. If you are stuck in a train between Paris and Brest (in the very far west of France ☺), you may find it inconvenient not to be able to access the repository history and other details about previous commits, especially if the current version is broken. In order to help preventing this situation anyway, PIPS is checked every minute and possibly fully recompiled.

So if you are Ronan, consider using `git-svn` or `svk` to have full copy of your branch or of the trunk on your laptop. Then you can develop offline including branching and commits. You will be able to push back into the SVN server your changes once you get back to civilized surroundings.

Why not switch all this stuff to `git`, which is so *cool*? There are several reasons:

system we already use subversion extensively at CRI for many projects, which are backed-up regularly (hourly `svnsync`, daily night on tape backup).

change management subversion is hard enough, and learning a new tool with a different model is a significant investment, which can only be justified with a clear added value. GIT is harder to understand and to learn than SVN.

backup with GIT, there is no natural backup on commits, as they are stored locally: you have to push your changes. This induces a greater risk of losing developments.

development model a centralized model helps with the development coordination, with distributing mails on commits to revelant recipients, and so on. It helps to access all developer branches.

nomadic the nomadic developer special needs can be addressed by other means (*e.g.* git-svn) without necessarily impacting the common infrastructure.

coordination is easier in a centralized models, where source divergence can be detected in one place, and possibly managed.

At HPC Project we use GIT with central servers, so the previous points of view from CRI are rather out-of-focus...

3.7.1 Keeping a local copy of PIPS repositories with `svnsync`

In order to access pips logs at anytime, you may consider keeping a local copy of pips repositories anywhere you want, including on a laptop, with the `svnsync` command. The operation is mostly network bound: it takes under 15 minutes to copy the whole history of the 5 repositories from the CRI LAN. Beware that this copy must only be used read-only.

```
mkdir PIPS_SVN
cd PIPS_SVN
# initialize the copy
for r in nlpmake newgen linear pips validation ; do
  svnadmin create $r
  echo -e '#!/bin/sh\nexit 0' > $r/hooks/pre-revprop-change
  chmod +x $r/hooks/pre-revprop-change
  svnsync init file://$PWD/$r https://scm.cri.ensmp.fr/svn/$r
done

# then sync from time to time.
# it takes a some time the first time, obviously...
# in the PIPS_SVN directory do:
for r in nlpmake newgen linear pips validation ; do
  svnsync sync file://$PWD/$r
done
```

3.7.2 The git-svn gateway

This section assumes that you have a good working knowledge of both GIT and SVN. If you use GIT, do not forget to save your repository regularly.

`git-svn` is a porcelain which ensures a limited compatibility between a centralized SVN repository and a local GIT repository. It provides the ability to pull/push commits from a single branch in an SVN repository. Beware of the limitations:

- branches and tags in more than one directory (we use subdirectories in `branches`) are not clearly managed;
- SVN properties are mostly ignored (ignore, externals...);
- cloning the pips tree does take a long time;
- it does not know about branch management in subversion (since 1.5), so merging an svn branch from git and pushing back the result breaks svn branch management;
- commit messages pushed from `git-svn` may be quite unhelpful;

So just use `git-svn` to manage an already created branch which will be seen as the `master` branch by GIT. Creating new branches under GIT and hoping to have them appearing under SVN seems hopeless.

Here is a summary of operations[4]:

[4] ISTM that this ammended version does not work or is at least error prone. Fabien.

```
# create copies of the repositories' trunks that
# you may wish to edit. Example:
git svn clone --stdlayout https://scm.cri.ensmp.fr/svn/nlpmake

git svn clone --stdlayout https://scm.cri.ensmp.fr/svn/newgen
ln -s ../nlpmake/makes newgen/makes
echo /makes >> newgen/.git/info/exclude

git svn clone --stdlayout https://scm.cri.ensmp.fr/svn/linear
ln -s ../nlpmake/makes linear/makes
echo /makes >> linear/.git/info/exclude

# This one takes 1 or 2 hours on a file repos...
# Deal with the fact that branches are hierarchized per user,
# so if I'm the user coelho
git svn clone --stdlayout -branches=branches/coelho \
    https://scm.cri.ensmp.fr/svn/pips
ln -s ../nlpmake/makes pips/makes
echo /makes >> pips/.git/info/exclude

git svn clone --stdlayout https://scm.cri.ensmp.fr/svn/validation

# live your life within git
# what about properties?
# svn:ignore & .git/info/exclude?

# when online, update/resync from svn
for d in newgen linear pips nlpmake validation; do
  pushd $d
  # consider running with option --dry-run
  git svn rebase
  popd
done

# push your local commits to svn
cd pips
git svn dcommit --dry-run
git svn dcommit
```

Since in the PIPS repositories there is also some quite old history (based on RCS and SCCS[5]), the history does not fit into classical SVN `trunk`, `branches` and `tags` and thus cannot either be understood well by `git-svn`. So for ultimate PIPS historian which does not want to use `svnsync`, you can have also a copy of the repository with `git-svn` without using the `trunk/branches/tags` convention. For this, just use a plain `git-svn` with:

```
git svn clone https://scm.cri.ensmp.fr/svn/nlpmake
git svn clone https://scm.cri.ensmp.fr/svn/linear
git svn clone https://scm.cri.ensmp.fr/svn/newgen
git svn clone https://scm.cri.ensmp.fr/svn/validation
git svn clone https://scm.cri.ensmp.fr/svn/pips
```

To have GIT ignoring the same files than SVN, you should try in each GIT top-directory a

```
git svn show-ignore >> .git/info/exclude
```

How to organize this? For example Ronan `Keryell` uses this hierarchy in a `PIPS` directory:

`git-svn-total`: stores all the PIPS history without `trunk/branches/tags` convention:

- `linear`

[5]introduced at CRI by Keryell Ronan on May 17, 1993

- `newgen`
- `nlpmake`
- `pips`
- `validation`

`git-svn-work`**:** stores the PIPS working copies he uses to work in with the `trunk/branches/tags` convention:

- `linear`
- `newgen`
- `nlpmake`
- `pips`: the working copy with `keryell`'s branches;
- `pips-all-branches`: the working copy with all the user branches;
- `validation`

3.7.3 The pips_git script

To apply a same `git` command on GIT repositories below the current directory, you can use the `pips_git` script. By default, the action is to do the equivalent of a `svn update`, that is a `git svn rebase`:

```
pips_git
```

because it is the most used command: it fetches new versions from the SVN repository and rebase your current work according to these last versions.

For example to get the status of all your GIT repositories, use:

```
pips_git status
```

There are also PIPS specific instructions to `pips_git`:

- `pips link-makes`: this adds into the GIT repositories a link to `PIPS_ROOT/makes` so that you can compile PIPS components directly in them and install it to an already installed version of PIPS somewhere.

4 Coding style and conventions

The code is Open Source, every one can peek into what you write, in the source code, in the comments, in the message logs... Just think about how frightening it is. Just scaring!

All the source code should be documented with Doxygen. It is far from this state right now, but the new code should be documented. So Doxygen documentation is supposed to be known.

Development language is in US English. Even if most developers are not native English speakers, they do some effort. ☺ That means that the English words should exist in some reference dictionaries, have at least their letters in the correct order and so on... ☺ Fortunately, there are some on-the-fly spell

checkers that are quite helpful[6]. So, do not commit something without proof reading and using a spell checker, even on the commit message!

All the code should be heavily documented. It is still not the case... PIPS is already a quite old and huge project and more than 20 years later it is hard to understand many details of the code. So just think about the PIPS in 20 years more, when you will have forgotten many things...

A lazy, but still useful, approach for the community could be that when you use a function that is not yet documented, please take some time to do it.

Avoid doing some code bloat by using some heavy code inlining or code copy. Try to find if the function you need is not already somewhere, typically into `ri-util`. If not, ask to some old team member, on the discussion list, on IRC, and if not, implement it. Try to thing genericity for the common use. If the function exist somewhere but not at an enough global place, move it for example into `ri-util`.

When committing into the version control system, try to have an explicit message structured with a summary line, and then a more precise description. A common habit with some tools such as `git` but that can be quite generalized.

Try to do more small commits instead of big ones to ease bug tracking in case of problems.

You should avoid to spoil the code with trailing blanks. Yes it is in visible so why to bother since the character storage is almost zero? Because it is not clean and, if you set the `show-trailing-whitespace` Emacs variable, you can see them every where in flashy red. Set your own text editor accordingly.

You should avoid tabulations, indent with spaces only so that every one can have the same vision of the code. Ideally use two spaces so that we don't need 24 inches screen to display the code without horizontal scrolling or line wrapping.

You should not do memory leaks. First because it is wicked and secondly because it often hides (or exhibits? ☺) nasty bugs that will be discovered by some other people or your children may be 20 years later and who will want to kill you during your retirement[7]. ☺ Now we have memory debugging tools such as Valgrind (see § 9.5), adding some `free()` in the code and tracking where the allocation and deallocation of objects happen is a very powerful tool.

Libraries are named `libXXX.a` where `XXX` is the logical name of the library: `vecteur`, `prettyprint`, `misc`, etc.

(Is this paragraph still valuable?) It is theoretically unuseful to put libraries in the Makefile dependencies, because header files which are automatically generated are in these dependencies, and are systematically modified each time an installation is performed. However, this does not work if the pass `p` calls the library `a` which needs the library `b`, and if `p` does not directly needs the library `b`: modifications to `b` will not provoke the link of `p`.

Each important library uses an environment variable `XXX_DEBUG_LEVEL` to control debugging messages. This rule hase an exception: `PARSER_DEBUG_LEVEL` corresponds to the `syntax` library .

Level 0 corresponds to the normal behaviour. The highest level is 8. Great care has to be brought to calls to `debug_on()` and `debug_off()`, because successive debug levels are stored in a stack, and it could disturb its coherency. The

[6]For example in Emacs in menu `Tools/Spell Checking/Automatic spell checking (Flyspell)` and `Select American Dict`. Make sure you have the right tools and dictionaries installed.

[7]Ronan Keryell is just thinking to the old `controlizer` right now. ☺

debug of a function of another library should not unconsciously be activated.

Have a look to § 9.3 to have more information on debugging.

5 Shell environment (sh, ksh, bash, csh, tcsh)

Many environment variables may be used by PIPS executables and utilities. They mainly describe PIPS directory hierarchy, compilation options, helper programs, and so on. PIPS should compile and may be launched without any such variables, as defaults are provided and computed at run time depending on the driver or executable path.

These variables should be initialized by sourcing the `pipsrc.sh` file for `sh ksh bash` shells, or `pipsrc.csh` file for `csh tcsh` shells. An initial version of these files is created by the setup script, and can be customized as desired.

Two variables are of special interest: `$PIPS_ROOT` which stores the root of PIPS current version, and `$PIPS_ARCH`.

5.1 PIPS architecture (`PIPS_ARCH`)

The variable holds the current architecture. If not set, a default is automatically derived with the `arch.sh` script.

A PIPS architecture is to be understood not as strictly as a computer architecture. It simply a set of tools (compilers, linkers, but also compiler options. . .) to be used for compiling PIPS. Thus you can have several pips architectures that compile and run on a very same machine.

This set of tools is defined in the corresponding `$PIPS_ARCH.mk` makefile, where usual `CC`, `CFLAGS` and so make macros are defined.

This configuration file is automatically included by all makefiles, hence changing this variable results in different compilers and options to be used when compiling or running pips. Object files, libraries and binaries related to different pips architecture cannot be mixed and overwritten one by the other: they are stored in a subdirectory depending on the architecture, *à la* PVM. Thus pips versions are always compiled and linked with libraries compiled for the same architecture.

Here are examples of pips architectures:

`.` : default version used locally, so as to be compatible with the past. `gcc`, `flex` and `bison` are used.

DEFAULT : default compilers and options expected to run on any machine (`CC=cc`, and so on). The C compiler is expected to support ANSI features, includes and so.

GNU : prefer gnu tools, as `gcc flex bison g77`.

SUN4 : SUN SUNOS 4 compilers `acc lex yacc f77` etc.

GNUSOL2LL : version for SUN Solaris 2 compiled with gnu softwares and using "long long" (64 bits) integers in the mathematical computations of the C3/Linear library.

GPROF : a gnu version compiled with -pg (which generates a trace file that can be exploited by `gprof` for extracting profiling information)

IBMAIX : the compilers and options used on IBM AIX workstations.

LINUXI86 : for x86 machines under linux.

LINUXI86LL : for x86 machines under linux with long long integers.

LINUX_x86_64_LL : for 64-bit x86 machines under linux with long long integers.

OSF1 : for DEC Alpha machines under OSF1.

5.2 ROOT variables

Several variables are used to explain where to build PIPS components and from where the components are picked. It is useful when working with several working copy and branches at the same time.

`EXTERN_ROOT` precises where external stuff like the Polylib is looked for;

`NEWGEN_ROOT` is used to select where to look for the NewGen tools and libraries;

`LINEAR_ROOT` gives the location of the Linear library to use;

`PIPS_ROOT` select the main PIPS directory used to pick runtime configuration files, scripts...

There is also a `ROOT` variable that can be set to precise the target main directory for the compilation.

For example, if you want to compile an experimental version of NewGen and compile it and install into the main directory of the NewGen installation (selected by `NEWGEN_ROOT`) to compile a global PIPS version without changing its compilation recipe.

- In a terminal in a `PIPS/git-svn-total/newgen/branches/gall/doom-branch` you run:

  ```
  make ROOT=$NEWGEN_ROOT compile
  ```

  to compile the local version into the standard NewGen directory.

- To compile PIPS, in another terminal with a shell in the `PIPS_ROOT` directory, use a classical

  ```
  make compile
  ```

It avoids changing around `NEWGEN_ROOT` when recompiling PIPS with various versions of NewGen, Linear or even part of PISP (libraries in Libs...).

Another way to achieve the same behaviour could be done with:

- In a terminal in a `PIPS/git-svn-total/newgen/branches/gall/doom-branch` you run:

  ```
  make compile
  ```

to compile the local version into the local directory.

- To compile PIPS, in another terminal with a shell in the `PIPS_ROOT` directory, instruct PIPS to look at the alternate NewGen version with a:

  ```
  make NEWGEN_ROOT=...PIPS/git-svn-total/newgen/branches/gall/doom-branch compile
  ```

You just need to keep track of what you are doing, especially when working with various versions and validating...

6 PIPS Project directories

This section describes the PIPS directory hierarchy. When extracting a repository with the `setup_pips.sh` script, three subprojects are extracted: `newgen` (software engineering tool, Section 9), `linear` (mathematical library, Section 15) and `pips` (the project). The three projects are organized in a similar way, with a `makes` directory with makefiles and a `src` directory for the sources. Other directories `bin share runtime doc` are generated on compilation (`make compile` or `make build`).

6.1 The `pips` directory

This directory contains the current version, namely pips subversion `trunk` directory. The `$PIPS_ROOT` environment variable should point to this directory. You can find the following subdirectories:

6.1.1 The `makes` subdirectory

PIPS makefiles, and some helper scripts.

6.1.2 The `src` subdirectory

The PIPS Sources are here.

Having a copy of this subtree and of the makes directory is enough for fully recompiling PIPS, including the documentation, configuration files and various scripts used for running and developping the PIPS sofware.

src/Documentation : This directory contains the sources of the documentation of PIPS. From some of these are derived automatically header and configuration files.

The `newgen` sub-directory contains the description (in LaTeX) of the internal data structures used in the project, as they are used in the development and production hierarchies. The local makefile transforms the `*.tex` files describing PIPS data structures into NewGen and header files, and exports them to the `include` directory. Thus the documentation actually *is* the sources for the data structures.

The `wpips-epips-user-manual` sub-directory contains the user manual.

The `pipsmake` sub-directory contains the definition of the dependences between the different interprocedural analyses as a LaTeX file, from which are derived the `pipsmake.rc` configuration file used by PIPS at runtime.

src/Passes : This directory contains the sources of the different passes, in several sub-directories named from the passes (`pips tpips wpips fpips`). If you add a pass, it is mandatory that the name of the directory is the name of the pass.

src/Libs : This directory contains the source of the several libraries of PIPS. It is divided into sub-directories named from the libraries. The name of the subdirectory must be the name of the library.

src/Scripts : This directory contains the source of the shell scripts which are useful for PIPS, the linear library and NewGen. It is further divided into several directories. In each sub-directory a local `Makefile` performs automatic tasks such as the installation of the sources where expected (usually in `utils`), depending whether the scripts is used for *running* or *developing* PIPS.

src/Runtimes : This directory contains the sources of the libraries used when executing codes generated by PIPS; in fact, there are only two libraries: The first for the HPF compiler `hpfc`, and the second one for the WP65 project.

6.1.3 The `bin` subdirectory

Pips current binaries that can be executed.

These binaries include `pips` (the main program), `tpips` (the same program with a interactive interface based on the GNU readline library) and `wpips` (the window interface).

Non architecture-dependent executables needed or used by pips are also available, such as old `Init Select Perform Display Pips` pips shell interfaces, and `pips tpips wpips epips jepips` launchers.

Actual architecture-dependent binaries are stored in different sub-directories depending on `$PIPS_ARCH` which describes the current architecture, as discussed in Section 5.

6.1.4 The `etc` subdirectory

Configuration files, such as pipsmake settings.

Also some configuration files, automatically generated from the LaTeX documentation in `src/Documentation`. Important files are `pipsmake.rc` and `wpips.rc`.

6.1.5 The `share` subdirectory

They include shell (sh and csh), sed, awk, perl and lisp scripts.

6.1.6 The `doc` subdirectory

Pips documentation, namely many PDF files describing various aspects of PIPS, the internal data structures and so on. Important documents: `pipsmake-rc` (the dependence rules between pips interprocedural analyses), `developer_guide` (this document!), `ri` (the description of the pips Intermediate Representation, also known as the Abstract Syntax Tree (AST)).

This directory populated from the various `src` directories and is also used to build the static PIPS home page.

6.1.7 The `html` subdirectory

This HTML documentation of PIPS is populated from the various `src` directories. There are real HTML files, and some generated from LaTeX files.

This directory is also used to build the static PIPS home page.

6.1.8 The `runtime` subdirectory

Environments needed for executing pips-compiled files. Namely `hpfc` and `wp65` use a PVM-based runtime and also `xPOMP` the graphical user programming interface. This directory includes the files needed for executing the generated files (as header and make files or compiled libraries), but not necessarily the corresponding sources.

6.1.9 The `utils` subdirectory

Other architecture independent tools and files used for *developing* pips (as opposed to *running* it), such as validating PIPS, dealing with the SVN organization.

But there are also some scripts used to help PIPS usage, such as displaying some graphs or dealing with PIPS output.

6.1.10 include

This directory contains all shared files needed for building PIPS. It includes C header files automatically generated from NewGen specifications and for each library, and from some file in the documentation. Also pips architecture configuration file that define makefile macros are there.

6.1.11 lib

This directory contains PIPS compiled libraries (`lib*.a`) ready to be linked. They are stored under their `$PIPS_ARCH` subdirectory.

7 Makefiles

The GNU `make` utility is extensively used to ensure the coherency of PIPS components. However, most `Makefile`s are not written by hand, but are automatically composed according components from `nlpmake` package and automatic dependence computation. It eases development a lot, by providing a homogeneous environment all over the PIPS software parts.

If you need some specific rules or targets, you have to write some makefile whith named ending with the `.mk` extension: they will be included in the global `Makefile`.

The rationale for relying on GNU make rather than make is that in the previous situation the software was relying heavily on SUN make special features, thus was highly not portable. Now it is more portable, say as much as the

GNU softwares. The complex makefiles of PIPS rely on GNU extensions such as conditionals for instance.

7.1 Global targets

Global targets are targets useful at the top of PIPS to with global meaning such as building a global binary file from the sources.

Useful targets defined:

build: generate a running version with the documentation;

compile: generate a running (hope so...) version. It is the default target to `make` if you don't precise one;

doc: build the documentation (such as the one you are reading right now);

install: install the project in user-specified directories. Those directories can be given using the `INSTALL_DIR` makefile variable and refined using `LIB.d ETC.d DOC.d MAN.d PY.d BIN.d` ... ones.

clean: clean up the project directories;

local-clean: remove eventually some stuff in the current directory;

tags: build the TAGS index files of identifiers found in the subdirectories (see 3.4 for more precision and other specific targets);

unbuild: clean up things: generated directories are removed;

rebuild: unbuild then build;

htdoc: generate an HTML version of the documentation

full-build: like `build` but with `htdoc` too.

If you use a global compiling target in the `prod` directory, it will compile NewGen, Linear and PIPS to build coherent binaries.

If you you use a global compiling target in a sub-component such as NewGen, Linear or PIPS, it will be compiled and eventually linked with the global libraries (defined according to the environment variables, by default the `prod` directory). So it is useful to have a coherent PIPS compilation of a branch (such as in you `pips_dev`) that is linked with production NewGen and Linear.

7.2 Local targets

Local targets are actions with a more local meaning to help developing some passes or documentation parts, such as:

`local-clean`**:** remove eventually some stuff in the current directory;

`depend`**:** `make depend` must be used regularly to keep file dependencies up-to-date. The local header file corresponding to the current pass or library must have been created before, otherwise `make depend` selects the header file from `include`, and the local header will never be created;

```
# The following macros define your pass:
TARGET = rice

# Sources used to build the target
LIB_CFILES = rice.c codegen.c scc.c icm.c

# Headers used to build the target
INC_TARGET =    $(TARGET).h

LIB_TARGET =    lib$(TARGET).a

# common stuff
ROOT    = ../../..
PROJECT = pips
include $(ROOT)/makes/main.mk
```

Figure 1: Example of `Makefile`.

`fast`: `make fast` recompile the local library and relink the executables with the global libraries (defined according to the environment variables, by default the `prod` directory) to build running PIPS executables into the main directories (by default below the `prod`);

`full`: `make full` rebuild PIPS fully from the root (defined according to the environment variables, by default the `prod/pips` directory), so that all rependencies are checked, compile your local sources and link with the global libraries to build running PIPS executables into the main directories (by default below the `prod`);

`compile`: also builds the documentationinto the main directories (by default below the `prod`);

`full-compile`: also builds the documentation for web publicationinto the main directories (by default below the `prod`);

There are other rules to describe...

Note that if you work on many libraries or documentation at the same time, to have a coherent compilation, you should not use local target but rather use some global targets from § 7.1 at a higher level in your branch or your working copy of the trunk.

I cannot see how to compile without installing to the PROD directory...

THIS PARAGRAPH IS OBSOLETE. `make libxxx.a` just recompiles the local `*.c` files, and creates the local library file `libxxx.a`. This is useful when making a lot of changes to check the syntax; or when making changes in several directories at the same time: in this case, the proper policy is to a) chose one of these directories as the master directory, b) build the `libxxx.a` in the other directories, c) make symbolic links towards them in from the master directory, d) and finally link in this last directory. If your pips architecture is not `.`, don't forget to link the proper libraries in the proper sub-directory.

A typical example of a local `Makefile` file is provided in Figure 1.

7.3 Debugging

By default, PIPS `make` does not stop on error but reports them on standard error output. But if you want to have `make` stopping on the first error to help debugging, just add `FWD_STOP_ON_ERROR=1` to the `make` invocation.

8 Validation

The validation is stored in a separate subversion repository `https://scm.cri.ensmp.fr/svn/validation`

For an overview of the validation daily use case, you should also have a look to § 3.6.2.

Different phases are validated in distinct sub-directories. Running the validation uses the a set of makefiles which allow to parallelize the validation inter-directory, or even intra-directory if you assure that there are no potential issues such as colliding temporary files or directories.

See the `README` at the root of the `trunk` for details on how to run the validation. Basically, one must simply type `make` with some target, either at the root of the validation or within a subdirectory. There are two main targets:

validate-out generate `out` files in `.result` directories, which are to be compared to corresponding `test` file;

validate-test overwrite `test` file, so that `svn diff` show the changes. This is also a way to accept the current validation results by committing with `svn` the `test` files you are happy with and using `svn revert` on the `test` you are not to get back the initial state. Look at § 8.5.1 for use cases.

Note that if you are using `git` instead of `svn`, do not use `validate-test` but `validate-out` with `p4a_validate --accept` for example instead, as explained in Par4All documentation;

validate same as `validate-out`.

See various examples around to add new validation files. The preferred way is to provide a source file `MyTest.f` or `MyTest.c`, a corresponding `MyTest.tpips` script, and store the standard output of the `.tpips` script into `MyTest.result/test`.

During the validation process (when you execute a make validate), the validation output, wich is stored in `MyTest.result/out`, is compared against the reference `MyTest.result/test` to know if the validation is correct. So to install a new validation item, you need to create the reference `MyTest.result/test`.

`pips_validate -a` (see afterwards) can help to install a new validation reference file and directory. Note that an additional transformation on output and/or reference file is possible for special cases. In fact depending on the compiler used for example, you can get floating point numbers in different formats. Then you might want to post process the output to make it compliant with the reference. To make this possible a filtering phase is included in the validation process and is explained in § **??**.

For more complex validations, a `.tpips` script without a direct corresponding source file but that may use many files from various location is also possible.

CAUTION: use a relative and local path in the `.tpips` file to reference the source file.

But it is best **not** using the following validation scripts directly, but rather to rely on the `Makefile` in the validation directory, which has very convenient targets including `validate accept clean`. Using this approach does not require the validation directory to be under `$PIPS_ROOT`.

8.1 Validations

Several compilations and validations run at CRI:

CHECK on every commit (down to the few minute scale...) compilation and validation based on the old but fast `nlpmake` makefiles on Debian Squeeze (compiegne).

If a commit changes the result of the validation or break the compilation, the author of the commit is in charge of fixing it quickly. A fast solution is to reverse the commit and investigate off-line. Note that this should never happen because the validation be run before committing anything to the `trunk`.

The whole compilation and validation runs in about 10 minutes.

AUTOC on every hour, compilation and validation with the `autoconf` infrastructure, including python and fortran95, on Debian Squeeze (compiegne).

This compilation and validation requires about 20 minutes.

CROSS 32-bit cross-compilation, on Debian Squeeze (compiegne).

NIGHT compilation and validation on Ubuntu Oneiric (andorre).

PRIVE compilation and private validation on Debian Squeeze (compiegne).

32BIT compilation and validation on Debian Lenny i686 (riquet).

The different error conditions and possible status of the working copies participating to such a compilation and validation test are quite complex. There is a `LOCK` directory in the `prod` to avoid the script to run twice at the same time. It may be necessary to remove this lock manually if the host was rebooted unexpectedly. Note that subversion uses locks as well.

8.2 Validation Makefile

The main user interface is from the `Makefile` at the validation top directory and is usable through few `make` targets:

validate-out validate-test run the validation;

clean to remove the results and output from the previous validation;

The validation can also be restricted to some directories by setting the `TARGET` environment variable, such as with

```
make TARGET=LoopRecovering validate
```

The validation can be run directly in a subdirectory of the validation, with something like:

```
cd SubDirectory
make validate-test
```

Subdirectories can be used recursively to store test cases under a validation directory. They are automatically included in the validation run if they are named `*.sub`. If another name without this suffix is used, the `D.sub` makefile macro can be overriden to specify the subdirectory list. See the `HardwareAccelerator/Makefile` for an example. These subdirectories are ignored if a `.later` or `.bug` tag file exist of the same prefix, just like ordinary test cases. The `PREFIX` macro can be defined to restrict the validation to a subset of cases.

8.3 Validation output

The validation scripts are run locally in each subdirectory. If differences are found, `.err` and `.diff` files are generated for each failing test, while the output is kept in the `.result` directory as `out` (or `test`).

A `SUMMARY.short` file is created at the root which summarizes the results.

When run at the root, the history of the results is kept under the `SUMMARY_Archive` directory.

- Obviously the `SUMMARY` file contains a summary of the previous validation.
- `FolderName_TestCaseName.err` contains the execution stderr of the test case named TestCaseName that you can find in the folder named FolderName.
- `FolderName_TestCaseName.diff` contains the diff of the output with the expected one.
- `FolderName_TestCaseName.out` contains the stdout of the test case named TestCaseName that you can find in the folder named FolderName.

8.4 Validation scripts

IS THIS OBSOLETE?

pips_manual_accept interactive script to accept new validation results. It accepts a list of directories or validation files to ask for accepting or it deals with all the validation by default.

If the script is used without any parameter all the validation acceptation is asked for all the validation differences.

If the script is called with `--new`, then the validation acceptance is asked against what is newer in the last validation than in the previous one.

Once you have accepts the new states as a reference for some validation items, you need to commit these changes to the validation repository, if and only if the corresponding pips sources are already in the subversion repository of course!

If you are Emacs oriented, setting the `PIPS_GRAPHICAL_DIFF` environment variable to `pips_emacs_diff` to use graphical Ediff merge mode sounds like a good idea. You can use it like:

```
PIPS_GRAPHICAL_DIFF=pips_emacs_diff pips_manual_accept --new
```

pips_force_accept to massively accept some changes in a validation directory. The script is to be run in a validation directory.

The script accepts two types of argument:

1. A list of `.result` directories as an argument. For each of the given directory the result of the previous validation will be made the reference for future validation tests. Note that you need to run the validation first.
2. A file with some excerpts of validation `SUMMARY` or a nightly validation mail in which all changed entry will be selected. Basically, each line in the file that is marked as `changed:` or `>␣changed:` will result in the matching acceptance.

   As described previously, for all those entries the previous validation will be made the reference for future validation tests. Note that you need to run the validation first.

Once you have run the `pips_force_accept` script, you need to commit these changes to the validation repository, if and only if the corresponding pips sources are already in the subversion repository of course!

8.4.1 Writing test cases and their validation scripts

Several issues have been encountered in the past with validation scripts.

First of all, validation scripts and related test cases are defined by the developper, not by the maintainer. The developper knows what the test case is about, and he/she fails to document the goal either in the source code or in the script... because he/she is in a hurry. Sometimes, the name of the test case can help the maintenance, but this is largely untrue for large librairies like `Transformations`. So, please, comment the goal(s) of your test cases.

By the way, is it a good idea to have many goals for one test case? It's attractive for the deelopper when many different cases and combination of cases must be tested. But it's a nightmare for the maintainer when one difference shows up or when PIPS core dumps if no unit test cases also exist because he will have to isolate the buggy part from the whole test case. So unit test cases are much eaasier to maintain, even though larger test cases are also very useful to check interactions and robustness.

Secondly, most test cases are not supposed to contain user errors. So, please, set the property `PIPS_ABORT_ON_USER_ERROR` to TRUE.

Thirdly, should test cases be restricted to test the output of the corresponding library or should they also test the outcome of the local results on other libraries? It is tempting to be strict and to select the first option, but, most of the time, some of the internal output will not appear in a text form after some prettyprinting. The consequences are only visible in the output of a later phase or transformation. So it is allowed to use any output in any of the library

validation suites. For instance, the validation of a parser is not restricted to the display of `PARSED_CODE`.

As a consequence, multiple outputs may be combined in the resulting `test` file. In case of regression bug, it is necessary to know what the components of the `test` file are. So, please, use `echo` liberally in your validation scripts to comment the output. This way, the maintainer will spot quickly which phase fails or generates a different output.

Fourthly, in the same way, when a transformation or an analysis is built on top of other transformations and/or analyses, it is useful to keep track of the intermediary results, each of them being duly commented. The origin of a change can be located much faster that way.

Fifthly, to ensure portability, do not create dependences on `/usr/include` as it varies from compiler and system version to version. If header files are necessary and if they must show in the output, please use a preprocessed version of your source code as input code. It can easily be retrived from he database in directory `Tmp`. Use `%ALLFUNC` rather than `%FUNC` to avoid unwelcome output. Note: using and displaying fixed header files is not compatible with executing the resulting code as this may cause portability problems.

Finally, the input source code and the output source code should be compiled and executed whenever it is possible. Care must be taken to use only files in the database, i.e. compile the input after the `create` command. Use `unsplit` to produce the output source and compile it in directory `Src`. Do not forget to remove the executable and the object files. Executable and object files should have unique names in the validation directory to let the validation process run in parallel when multicore machines are available.

PIPS is developped to handle legal source code only. It must nevertheless detect all kinds of errors and issue warnings. To validate PIPS behavior when errors occur is tricky because errors are printed out on `stderr` together with system and version and time dependent information. Probably, an error log file should be used instead, but this is not implemented yet. Meanwhile, use the suffix `.test` to keep all PIPS output for the non-regression test. The case with warnings is easier because PIPS provides a `Warnings` file in the database. Do not forget to close the database and to explain what you are doing before you display this file.

Some test cases may become obsolete, but we do not have resources to maintain them properly.

Test cases can be factorized using `default_tpips`, `default_test` or `default_pyps.py`, but the same rules should be applied. Of course, it is less flexible to show information about a specific function among hundreds.

Another way suggested by Ronan `Keryell` for sharing test scripts between cases is to create a link to a script which uses its 0th (`$0` in shells) argument for the test case name. It is heavily used in the Par4All validation.

Scripting used to be shell based when PIPS was started. It is now mostly `tpips` based. Such scripts should end with a close, a delete of the current workspace, and a quit. For advanced stuff, consider python programming with PyPS.

When multiple source files are used, it is better to keep them in a subdirectory. The maintenance of the validation library is easier and the output of the validation process is not clobbered with information about skipped files.

Scripts can sometimes look stupid because a `quit` appears in the middle.

Do not forget that they are used to track bug and that additional information may be useful. For the same reason, the `delete` are often commented out. The cleaning is then performed by the higher-level validation process.

With the wide availability of multicore chips, the validation process could be parallelized. This implies that scripts remains as independent as possible of each other. They shoud not use file names such as `a.out`.

8.5 Parallel Validation

This infrastructure runs non regression tests in parallel, either at the directory level (distinct validation directories are run in parallel) or at the file level (distinct validation files within a directory are run in parallel). The former is useful for a full validation, the later is very useful for the developer who wants to test specific new developments on specific test cases very quickly.

These parallel validations **require** the validation directory to be an *svn working copy* of the validation repository. The infrastructure relies on **GNU Make** for running test cases in parallel while controlling the cpu load on the host. There **must** be a *Makefile* within each directory, which defines a `PARALLEL_VALIDATION` macro if the validation can run locally in parallel. The typical (very short) `Makefile` is:

```
PARALLEL_VALIDATION = 1
include ../validate.mk
```

There is a default timeout of 600 seconds per run. This may be changed by redefining the `TIMEOUT` make macro. Note that timeout does not seem to work when a subshell is started, and may interact with timout trigger used within the linear library.

A validation directory can contain subdirectories which will also be validated recursively. This is done automatically on `*.sub` subdirectories, but an explicit list of directories can also be provided as:

```
PARALLEL_VALIDATION = 1
D.sub = subdir1 subdir2 subdir3
include ../validate.mk
```

The validation will run test cases available in the current directory `.` and into the 3 subdirectories. These subdirectories are skipped if corresponding `*.later` or `*.bug` files exists to mark the directories as future or error cases. Note that all these subdirectories must include a similar *Makefile* to have the necessary targets, something like:

```
PARALLEL_VALIDATION = 1
include ../../validate.mk
```

See the `HardwareAccelerator` validation directory for a working example of validation subdirectories.

The validation makefile tries to guess whether pyps or fortran95 cases should be validated, and skips them otherwise.

8.5.1 Intra-directory parallel validation

For the intra-directory parallel validation, all validation scripts **must not** have *side effects* that may interact with other scripts, for example that temporary files should be fully isolated from those generated by other scripts. If so, you may add the define the parallel validation macro as outlined in the previous section. Otherwise, it still works but all tests are run one after the other. Then, to run an intra-directory parallel validation, with a parallelism of 8 bounded by a load of 10 on the host:

```
sh> cd validation/HardwareAccelerator
sh> make -j 8 -l 10 validate-test
```

Then local file `RESULTS` contains a summary of all results, typically whether a test *passed*, *failed*, *changed* or was ignored for some reason. To look at one/all difference:

```
sh> svn diff TEST_CASE.result/test
sh> svn diff
```

To accept one/all new result:

```
sh> svn ci TEST_CASE.result/test
sh> svn ci
```

To ignore a specific result:

```
sh> svn revert TEST_CASE.result/test
```

To reset the directory, that is to ignore all new results:

```
sh> make unvalidate
```

One can also run just one specific test:

```
sh> rm TEST_CASE.result/test
sh> make TEST_CASE.result/test
```

On can run a subset of tests based on a file prefix:

```
sh> cd validation/Transformations
sh> make PREFIX=unroll -j 8 validate-test
```

Special targets `bug/later/default/slow-validate-test/out` allows to validate some subsets (later, bug, slow cases).

8.5.2 Inter-directory parallel validation

For the inter-directory parallel validation, on defaults or only some directories by defining the `TARGET` make macro:

```
sh> cd validation
sh> make -j 8 -l 10 validate-test
sh> make -j 8 -l 10 TARGET='HardwareAccelerator Hpfc' validate-test
```

The detailed results are stored in file `validation.out`. Possible test results include, when a test is run:

passed fine!

changed there are some changes in the output.

timeout the test timed out.

failed the test failed (voluntary abort, unvoluntary segfault...).

noref there is no `test` file in a result directory.

Moreover possible warnings include:

skipped some tests are ignored in this validation run (bugs, laters, slow, defaults)

missing some result directories are missing

missing-vc some result directories are not under version control.

multi-source there are several contradictory C or Fortran source files.

multi-script there are several contradictory validation scripts (tpips, tpips2, test, python), some of which may be ignored.

nofilter the `flt` standard output and error filter script is missing for a `tpips2` script.

orphan no way (script) to run a test

keptout the case could not run (say Pyps or Fortran 95 support are not available).

bug/later/slow test case skipped because tagged as shown and such cases were explicitly asked to be ignored.

A summary of the result without passed tests is stored in `SUMMARY_Archive/SUMMARY-last`. In order to accept or reject changes, do svn reverts or commits as discussed in the previous section. To revert all changes, and cleanup everything, at the root of the validation run:

```
sh> make clean
```

9 Debugging: NewGen, macros, debug levels,...

This section describes some issues and tricks about debugging PIPS.

Displaying a textual resource from PIPS use a pager (by default the `more` command) that may interfere badly with the debugger. A way to avoid this is to set the `PIPS_MORE` environment variable to the `cat` command. A way to do this from the `gdb` debugger is to do:

```
set env PIPS_MORE=cat
```

9.1 Debugging PIPS C Source Code for the Compiler

PIPS use a lot of macro processing to ease the programmer life with NewGen, but, in some cases, it may be difficult to understand what happens to the C source code. To help the programmer, there is a special `+cpp` target to build a CPP-preprocessed version of a file. For example

```
make ricedg.c+cpp
```

make a preprocessed version of the file `ricedg.c`. To help debugging, gcc-specific options are added to have the macro definitions at the beginning of the file too, to keep the comments and display the list of included files.

9.2 Launching gdb

GDB is launched the usual way on `tpips`

```
sh> gdb /path/to/binary/of/tpips
gdb> run
...
```

or on a process, using the process number.

When a validation default `tpips` script is used:

```
sh> WSPACE=foo FILE=foo.f gdb /path/to/binary/of/tpips
gdb> run default_tpips
...
```

When the PyPS Python pass manager is used:

```
gdb --arg python my.py to debug pips parts
```

or:

```
pdb my.py to debug pyps parts
```

9.3 Debugging PIPS C Source Code Dynamically

We use environment variables `xxxx_DEBUG_LEVEL` to control debugging output at the library level. So, one library (or phase) `x` calls library `y`, the entry points in `x` should handle the debugging level as `Y_DEBUG_LEVEL` and not `X_DEBUG_LEVEL`. See for instance the main entries in library `pipsdbm`. You can detecte them because they all contain a call to `debug_on()` on entry and a call to `debug_off()` on exit.

A library with different components may also define several debug levels, with different names. For instance, `SAC_DEBUG_LEVEL`, `SAC_ATOMIZER_DEBUG_LEVEL`, `SAC_SIMD_LOOP_UNROLL_DEBUG_LEVEL`, etc... Unspaghettify is supposed to be controlled by `CONTROL_DEBUG_LEVEL`, or by something more restrictive.

Let suppose you develop your own CUDA library. Then you should have your own `CUDA_DEBUG_LEVEL`, used with `debug_on("CUDA_DEBUG_LEVEL")`, `debug_off()`, `ifdebug()` to guard debugging code and `pips_debug()` to reduce the number of `fprintf(stderr,...)` and add useful information automatically,

e.g. the current function name. Another debugging macro, `debug`, is now obsolete and must be replaced by `pips_debug` which add the current function name automatically. So, the second parameter of calls to `debug`, which is the current function name, must be deleted. This makes code maintenance easier.

The current debug level is a global variable called `the_current_debug_level`, which can be set easily under gdb. Calls to `debug_on` and `debug_off` push and pop the hidden debug level stack.

If the environement variable is not set or if it set to 0, there is no debug output. The largest value for debugging is 9.

Another macro, `pips_assert`, is useful to enforce contracts between callers and callees. The first argument describes the assertion, but it is often misused to give the current function name. This is useless because the information is added by the macro itself.

9.4 Debugging and NewGen

NewGen is a software engineering tool, from which almost all PIPS data structures are generated. It is described in ???. It is considered as an external tool, and is independent from the PIPS project. However, it is required for building pips from scratch, and the Newgen runtime library (genC) must be linked to pips.

NewGen sources are in the directory `$NEWGEN_DIR`. The global organization is similar to the PIPS organization.

`$NEWGEN_ROOT` and `$NEWGEN_ARCH` refer to the current version and architecture to be used. If `$NEWGEN_ARCH` is not defined, is defaults to `$PIPS_ARCH` and then to "`.`", so that if one develop or update NewGen as part of PIPS there is no trouble.

It may (alas!) sometimes be useful to consult them. Several functionalities are useful when debugging:

- `gen_consistent_p()` checks whether an existing NewGen data structure is (recursively) well formed.
- `gen_defined_p()` checks whether an existing NewGen data structure is (recursively) completely defined.
- `gen_debug` is an external variable to dynamically check the coherence of NewGen data structures.

  Activation:

  ```
  gen_debug |= GEN_DBG_CHECK;
  ```

  Desactivation:

  ```
  gen_debug &= ~GEN_DBG_CHECK;
  ```

  And when all else fails:

  ```
  gen_debug = GEN_DBG_TRAV_OBJECT
  ```

- To print the type (the domain) number of a NewGen object under `dbx` or `gdb`:

```
print obj->_type_
```

-

- To print the type (domain) name of a NewGen Object from its number:

  ```
  p Domains[obj->_type_].name
  ```

  or, when facing a high stack due to a `gen_recurse`, use argument `obj` of calls to `gen_trav_obj`

  ```
  p Domains[obj->i].name
  ```

- To directly dig into a NewGen object, you can access an attribute `a` of an object `obj` of type `type` with its field name `_type_a_`, thus such as `obj->_type_a_`.

  For example, picked from the RI:

  - To print the name of an entity `e` (an entity is a PIPS data structure, see `ri.tex`):

    ```
    p e->_entity_name_
    ```

  - To print the label of a statement `stmt`, by studying the RI you can see that:

    ```
    statement = label:entity x number:int x ordering:int
              x comments:string x instruction:instruction
              x declarations:entity* x decls_text:string;
    ```

    and since a `label` is an `entity` as declared with:

    ```
    tabulated entity = name:string x type x storage x initial:value
    ```

    you can get the information you want with:

    ```
    print stmt->_statement_label_->_entity_name_
    ```

- To print a statement `s` as human code, `call print_statement(s)`. You may need to position some internal variable to have the correct meaning, such as if you are dealing with C program with:

  ```
  set is_fortran=0
  ```

- To print an expression `e` as human code, use

  ```
  call print_expression(e)
  ```

- To print some kind of Newgen object, e.g. statement or expression, use

  ```
  call dprint(e)
  ```

  which uses the dynamic typing of Newgen to identify the related print function. It also works for lists of Newgen objects, as long as they are properly typed: all objects must be of the same type. Function `dprint` is always ongoing work. It has to be extended to cope with new types. or types that have not been dealt with yet.

Note that NewGen cannot be simplified to use `stmt->label->name` in the general case because of potential clashes with C reserved keywords. In our case, since we represent C code in the RI, we would have naming conflicts on `for`, `while`, `to`, `return` and so on.

It should be also possible to use PIPS and NewGen macros (accessors and so on) from GDB instead of these NewGen internals, if the sources are compiled with the good options (`-ggdb -g3`). You may remove the `-O` option from the `CFLAGS` in the makefiles to avoid variable optimizations into registers. A good and simple practice to achieve this is to use:

```
make CFLAGS='-ggdb -g3 -Wall -std=c99'
```

Now `gdb` (7.2 at the time of writing) can be easily extended with scripts (*sequences*) and even Python with an API to program directly in GDB and inspecting the binary. So we could imagine that NewGen or PyPS could provide functions compatible with NewGen generated C macros and functions.

To ease this kind of inspection, some keyboard accelerators can be defined inside the Emacs debugger mode by adding into your `.emacs` such as:

```
(add-hook 'gdb-mode-hook
 (function
  (lambda ()
   (gud-def pips-display-type "print Domains[%e->_type_].name" "t"
            "Display the domain (that is the type) of the selected NewGen object")
   (gud-def pips-display-entity-name "print %e->_entity_name_" "n"
            "Display the name of the selected entity")
   (gud-def gud-affiche-expression "call print_expression(%e)" "e"
            "Print a PIPS expression")
   (gud-def gud-affiche-statement "call print_statement(%e)" "s"
            "Print a PIPS statement")
   (setq
    ;; Use Emacs with many GDB buffers as an IDE:
    gdb-many-windows t
    ;; Don't mix up gdb output with PIPS output
    gdb-use-separate-io-buffer t
    )
   )
  )
 )
```

These function are then available with the `C-x C-a` binding, so you click on an interesting variable and then use

`C-x C-a t` to display the type of a NewGen object;

`C-x C-a n` to display the name of a PIPS entity;

`C-x C-a e` to display the type of a PIPS expression;

`C-x C-a s` to display the type of a PIPS statement.

9.5 Debugging Memory Issues

There are tools quite useful for tracking memory issues that may happen when dealing with complex structures in C as in PIPS.

The more powerful is Rational Purify, but unfortunately it is non free. But since it is a good tool, that makes sense to invest in such a tool. It works by instrumenting all the memory access at link time. The execution is slower.

Valgrind is another tool to run a program in a virtual machine that monitor every bit of the memory to detect memory corruption or the use of an uninitialized bit (yes, not byte) of memory. Of course the execution is quite slower. Interesting use is:

```
valgrind --track-origins=yes --freelist-vol=1000000000 tpips
```

but other options are described in the manual, especially in the section *MEMCHECK OPTIONS*. You can put the options in your `~/.valgrindrc` or the `VALGRIND_OPTS` environment variable. For a power user, some interesting options to begin with:

```
--malloc-fill=a5 --free-fill=e1 --freelist-vol=1000000000
         --track-origins=yes --leak-resolution=high
         --num-callers=50 --leak-check=full
```

Electric fence (`libefence`) also proves to be usefull when you need to stop exactly when an illegal memory operation happens. Note that because of `regcomp`, the `EF_ALLOW_MALLOC_0` environment variable has to be defined when debugging `tpips`.

Another solution is to ask the compiler to instrument the code at compile time to detect memory corruption later at run time. This is done with `gcc` by compiling with `-fmudflap` option (or `-fmudflapth` when PIPS is multi-threaded...). The run-time behaviour is controlled at execution with the `MUDFLAP_OPTIONS` environment variable.

9.6 Debugging the internal representation in a browser (IR Navigator)

A web interface to the internal representation in built-in. A convenient way to access it is provided in PyPS. For instance the following python sample opens 2 tabs in your default web browser, each with a dump of the IR after a specific transformation :

```
import ir_navigator
...
m = w.fun.main

m.localize_declaration()
w.fun.main.ir_navigator(openBrowser=True)

m.clean_declarations()
w.fun.main.ir_navigator(openBrowser=True,keep_existing=True)
```

The symbol table is huge and can overload the browser, thus it's not displayed by default. You can activate its display:

```
w.fun.main.ir_navigator(openBrowser=True,symbol_table=True)
```

10 Documentation

10.1 Source documentation

The various documentations, including the source of the web pages, are stored in the `src/Documentation` directory.

For example this guide is stored in `src/Documentation/dev_guide`.

Have a look also on § 3.1 about Doxygen documentation of the sources.

10.2 Web site

The various PIPS documentations can be published on the web site `http://www.cri.ensmp.fr/pips/` with the `pips_publish_www` once you have compiled it.

To publish the Doxygen documentation of the sources, see § 3.1, once you have compiled the Doxygen version.

There is also a new dynamic web site `http://pips4u.org` that reference better the information stored on`http://www.cri.ensmp.fr/pips/` . The idea is to have the content edited naturally and collaboratively onto `http://pips4u.org` and from time to time injecting back the content into the `http://www.cri.ensmp.fr/pips/` by editing the `src/Documentation/web` directory.

11 Library internal organization

The source files for the library *L* are in the directory `$PIPS_ROOT/src/Libs/`*L*.

A `Makefile` must be created in this directory. It can be copied from an existing library since it ends with PIPS special include instructions used to simplify the make infrastructure, but it must be updated with the names of the local source and object files. Additionnal rules compatible with the `make` syntax can also be added in this `Makefile`, but now the preferred method is to put them in `local.mk` as explained later. For local LEX and YACC files, use the `$(SCAN)` and `$(PARSE)` macros, and do not forget to change the 'yy' prefixes through some sed script or better with the correct parser generator options to avoid name clashes when linking PIPS (which already includes several parsers and lexers).

The `Makefile` calls the PIPS make infrastructure to define for example a default `recompile` entry to install the library in the production hierarchy, typically in directories `include` and `lib` at the root.

Great care must be taken when creating the library header file *L*`.h`. This file includes the local header file *L*`-local.h` (which contains macros, type definitions...) written by the programmer and the external functions declared in the local source files, which are automatically extracted using `cproto`.

L`.h` must never be directly modified as indicated by the warning at the beginning. It is automatically generated when invoking `make header`, or when *L*`-local.h` has been modified, but not when the source files are modified. This avoids inopportune recompilations when tuning the library, but can lead to coherency problems.

If a main program to test or use the library exists, then it is necessarily a pass (see section 16), or a `main()`. But the easiest way to test a library is to test through the PIPS infrastructure after declaring it as explained in section 11.3.5.

11.1 Libraries and data structures

PIPS data structures are managed by NewGen. For each data structure declaration file in `$PIPS_ROOT/src/Documentation/newgen`, NewGen derives a header file that defines the object interfaces, which is placed in `$PIPS_ROOT/include`.

For instance, the internal representation (i.e. the abstract syntax tree) is called `ri`[8]. It is described in the LaTeX file `ri.tex` in `$PIPS_ROOT/src/Documentation/newgen`. It is then automatically transformed into a NewGen file `ri.newgen`. And finally, NewGen produces an internal description file `ri.spec`, a C file with the object method definitions and a header file `ri.h`, which must be included in each C file using the internal representation.

Thus, it is not possible to build a new library with the name `ri`, because the corresponding header file would be called `ri.h`. The library in which higher order functions concerning the `ri` are placed, is rather called `ri-util`. It should be the case for all the other sets of data structures specified using NewGen. Such existing libraries are `text-util` and `paf-util`. However, mainly for historical reasons, there are numerous exceptions. Many functions are in fact in the library where they were first necessary. An example is the `syntax` library (the parser). It should be refactored out some day.

11.2 Library dependencies

Library dependence cycles must be avoided. Links are not easy to manage when a module a from library A calls a module b form library B which itself calls a module a' from library A. This situation also reflects a bad logical organization of libraries. Therefore, PIPS libraries must be organized as a DAG, i.e. they must have a partial order. With modern linkers, it is no longer an issue but we still have the issue at the `.h` level if we have cross-dependencies such as cross-recursion.

Several utilities, such as `analyze_libraries`, `order_libraries`, `grep_libraries`, `order_libraries_from_include` (see `$PIPS_ROOT/src/Scripts/dev`), can be used to determine the dependences between libraries, a total order compatible with the library partial order, and the possible cycles. The results are stored in `/tmp/libs.?` where the question mark stands for several values:

- u: uses, modules used by each library.
- d: defs, modules defined by each library
- j: join between uses and defs
- o: order between libraries
- etc...

11.3 Installation of a new phase (or library)

Some libraries implement functionalities which are accessible by the users via the `pipsmake` mechanism. For a good comprehension of the material contained in this subsection, the reader is referred to the corresponding documentation.

[8]French for *Représentation Interne*, *i.e.* Internal Representation.

A PIPS phase *MYPHASE* is implemented as a C function named `myphase` with a single argument, which is the name of the module (a function or procedure) to analyze. It returns a boolean value, which indicates whether the computation has performed well. It therefore resembles the following dummy function:

```
bool␣myphase(const␣string␣module_name)
{
␣␣bool␣good_result_p␣=␣true;

␣␣//␣some␣computation...

␣␣return␣good_result_p;
}
```

This phase is located either in a new file[9] in an existing library or in a new library named `myphase`. The latter is not compulsory, but it is advised for new contributors. If the new phase requires global variables to tune its behavior, the *properties* mechanism must be used. The new phase must acquire the necessary resources by invoking `pipsdbm`. Recursively invoking `pipsmake` is forbidden at this level.

This section does not deal with the case where a new kind of resource is introduced. This is the object of Section 11.4 instead.

Here are now the steps to install a new library named `mylib` with a new C file named `myphase`.

You should also have a look to the document describing the new build infrastructure using `autotools` and update the various files accordingly: `http://www.cri.ensmp.fr/pips/auto_pips.htdoc/auto_pips.pdf` and `http://www.cri.ensmp.fr/pips/auto_pips.htdoc`

11.3.1 In the `src/Libs` directory

- First, create[10] a new directory `mylib` in `src/Libs`;
- add the new directory name to the `$PIPS_ROOT/src/Libs/local.mk`, so that the recursive make will consider it.
- add the new directory name to the `$PIPS_ROOT/libraries.make`, taking care of library dependences.
- also add it to `$PIPS_ROOT/src/Libs/Makefile.am` for the build infrastructure using `autotools`; make sure you modify `PIPS_SUBDIRS` *and* `libpipslibs_la_LIBADD`.
- once you have created the new directory or added new library source files, update include dependencies for make: `rm deps.mk ; make deps.mk`. Note that there should be no cycles in library dependencies (alas, this is not currently the case...), so at least do **not** add more cycles.

Skip this step if you are using an already existing library.

[9] It is also possible to define several phases in one C file, but a new user should avoid it.
[10] Beware of svn. You can either perform a `svn mkdir` now, or a `svn add` later.

11.3.2 At the root directory

A new library needs to be declared for the full compilation.

- Update file `libraries.make` at the root of the project. It contains the ordered list of PIPS libraries for linking. The order may not be important with modern `ld`, but if so a library must appear *after* the libraries it uses.
- Also update file `configure.ac` to add a reference to the automale generated makefile for the autotools compilation.

11.3.3 In the `src/Libs/mylib` directory

- Add some `.c` files, at least the one containing the C `bool myphase(const string name)` function, which corresponds to the phase to add, generally is);
- create a `Makefile` (see examples in the other libraries) if `mylib` is a new library. This file must list the sources of the library and the targets to be built, then it must include the common makefile `$(ROOT)/makes/main.mk`; if an existing library is used, update its `Makefile` to add the new source files;
- Then run the commands `make`. The file `mylib.h` is installed in the `$PIPS_ROOT/include` directory, and `libmylib.a` in the `$PIPS_ROOT/lib/$PIPS_ARCH` directory. The library functions are now accessible by the other libraries and passes.

  Note that the `cproto` command is used to generate the library header file `mylib.h`. To force this file to be build again, type `make header`.
- Create a `Makefile.am` for the `autotools` build infrastructure (see examples in the other libraries).

Since we use `svn` to deal with the code, it is a good practice to add `svn:ignore` properties to ignore considering automatically generated files, such as for `Libs/gpu`:

```
.build_*
.depend.*
.header
gpu.h
LINUX*
```

11.3.4 In directory `src/Scripts/env`

If you need to introduce new environment variables or to modify old ones (not likely):

- possibly update `pipsrc.sh` file. It contains the shell environement variables for PIPS. Do not modify `pipsrc.csh` since it is automatically generated by the following item;
- run the `make` command to automatically build the files and to install them in `$PIPS_ROOT/share`.

Note that the `pipsrc.*sh` files created at setup time by the setup command is not affected, you must copy them again by hand since they were copied from an older version at download time. So this change will only affects future installations. Another possibility to synchronize with the production version is to download a new version of the `setup_pips.sh` script (or better make a link to `MYPIPS/prod/pips/makes/setup_pips.sh` for example to have an always up-to-date version) and run again `setup_pips.sh`. It won't erase the directories but will pull the new version of subversion and will reconstruct the `pipsrc.*sh` files.

11.3.5 In directory `src/Documentation/pipsmake`

- Declare the new phase to `pipsmake` by adding the necessary rules in the `pipsmake-rc.tex` file in the proper section. Each rule describes the dependences between the input and output resources of each phase. Just have a look at the other rules to build your owns. You can also add aliases to obtain nice menus in the `wpips` interface.

  For instance, for a phase *myphase* which reads a resource *res1* and produces a resource *res2* for the current module, the rule and its alias are[11]:

  ```
  \begin{PipsMake}
  alias myphase 'My Phase'

  myphase > MODULE.res2
          < PROGRAM.entities
          < MODULE.res1
  \end{PipsMake}
  ```

  You can document your new phase using the `PipsPass` environment. It takes the phase name as a mandatory argument. Its body briefly document the phase.

  ```
  \begin{PipsPass}{myphase}
  A very nice phase !
  \end{PipsMake}
  ```

  If you need to further refer to this phase, use `\PipsPassRef{myphase}`.

  Note that all phases generally use the program entities, for example if you display some code in debug routines..., that is all the objects from the current C or Fortran program and from C and Fortran run-times that have a name. This resource, the global symbol table, is named `PROGRAM.entities`.

- You may want to declare new PIPS global properties to parametrize your new phase, or use previously defined properties. In the first case, simply add a

[11] Beware not to add any space before the new phase name (here *myphase*). This results in an error message from function `get_builder` (`no builder for myphase`).

```
\begin{PipsProp}{propname}
propname default_value
\end{PipsProp}
```

where `propname` is the name of your property, `default_value` is a boolean, an integer or a string ((`TRUE`—`FALSE`), `[0-9]+`, `"[^"]*"`). In the second case, add a `\PipsPropRef{propname}` reference to the used property.

- Run the command `make` to derive the various files `pipsmake.rc`, `phases.h`, `resources.h`, `wpips.rc` and `builder-map.h` , and to copy them into `$PIPS_ROOT/doc`, `$PIPS_ROOT/include` and `$PIPS_ROOT/etc`.
- If this is a new library without phases (like ri-util, text-util, naming, pipsdbm, pipsmake...), add the library to the core library list in script `pass2libs.pl` so that the library is included in a partial compilation. Alternatively, consider adding the library to the library-dependency dictionary in the same script so that it is included only when needed.

11.3.6 In directory `src/Libs/pipsmake`

Execute the command `make recompile` to recompile the `pipsmake` library.

New *properties* (that are global variables with a user interface in PIPS) may be necessary for the fine control of the new phase. They must be declared in the `pipsmake-rc.tex` file.

11.3.7 In directory `src/Passes`

Beware that for the autotool compilation there are hardcoded directory lists for includes in some `Makefile.am` for tpips and others, that may need to be extended if new directories are added and required to compile some passes. See also `fortran95/*.mk`.

Execute the command `make recompile` to recompile and install updated versions of `pips`, `tpips` and `wpips`.

With fast computers, it may be easier to skip the previous steps and to rebuild PIPS from directory `PIPS_ROOT` using `make build`.

You might also need to modify other `Makefile.am`, for instance in `src/Scripts/tpips2pyps/`...

11.3.8 The final touch

It can also be necessary to update the shell scripts `Init`, `Build`, `Select`, `Perform`, and `Display` in directory `src/Scripts/drivers`.

Are they still in use since tpips? Should be automatically generated anyway... RK

11.4 Dealing with a new resource

A new resource is declared by its use or its production by one of the rules in the file `pipsmake-rc.tex` (see below). Here are the steps to perform to declare a new resource `myres` once the phase `myphase` from the library `mylib` has been declared and installed as shown in the previous section. This assumes that the function

```
bool myphase(char *module_name)
```

is available.

11.4.1 In directory `src/Documentation/pipsmake`

- Declare the new resource `myres` by modifying the file `pipsmake-rc.tex`. Each rule describes the dependances between the input and output resources of each phase. Just have a look at the other rules to build your owns. You can also add aliases to obtain nice menus in the **wpips** interface.

  For instance, for a phase *myphase* which reads a resource `another_res` and produces a resource *my res* for the current module, the rule and its alias are:

  ```
  alias myphase 'My Phase'

  myphase > MODULE.myres
          < PROGRAM.entities
          < MODULE.another_res
  ```

- Then you can do as the end of section 11.3.5.

11.4.2 In the directory `src/Libs/pipsdbm`

The new resource (if any) must be declared to the resource manager `pipsdbm`:

- Update the file `methods.h` by adding a line for the `DBR_MYRES` resource. Macros are predefined for saving, loading, freeing and checking a resource if it is a newgen domain, a string, and so on. For instance a NewGen resource is simply declared as:

  ```
  { DBR_MYRES, NEWGEN_METHODS },
  ```

  For file resources, the `pipsdbm` data usually is a string representing the file name within the pips database. All such resources are suffixed with `_FILE`. It is declared as:

  ```
  { DBR_MYRES_FILE, STRING_METHODS },
  ```

- Run the commands `make` to update `pipsdbm` and install it.

11.4.3 The final touch

Then follow with the end of section 11.3.5 and then section 11.3.7.

11.4.4 Remark

It is necessary to recompile PIPS before any test, because changing the header files generated from `pipsmake-rc.tex` can lead to inconsistencies which only appear at run time (the database cannot be created for instance).

11.5 Modification or addition of a new NewGen data structure

NewGen data strutures for PIPS are defined in LaTeX files which are in the directory `$PIPS_ROOT/src/Documentation/newgen`. These files include both the NewGen specifications, and the comments in LaTeX format. DDL[12] files are automatically generated from the previous LaTeX files. When declaring a new data structure, two cases may arise:

1. The new data structure is defined in a *new* file `mysd.tex`:

   This file has to be declared in the local `newgen.mk` included by `Makefile` (in `$PIPS_ROOT/src/Documentation/newgen`) for its further installation in the PIPS hierarchy. It merely consists in adding its name `mysd.c` to the `F_c` variable list.

   If the data structure is to be managed by `pipsdbm`, *i.e.* it is declared as a resource in `pipsmake-rc.tex`, then

   `$PIPS_ROOT/src/Lib/pipsdbm/methods.h`

   must be updated.

   It is not compulsory to immediately execute the command `make` in these directories. But it will be necessary before any global recompilation of PIPS (see below).

2. The new data structure is added to an already existing file `mysd.tex`: modifying this file is sufficient, and the above steps are not necessary.

Then, in both cases, the next steps have to be performed:

- In the directory `$PIPS_ROOT/src/Documentation/newgen`, execute the command `make`. This command builds the DDL files `mysd.newgen`, `mysd.spec`, `mysd.h`, `mysd.c` and some common files such as `specs.h` and this copies the generated files to the `$PIPS_ROOT/include` directory;

- NewGen globally handles all the data structures defined in the project. Hence any minor modification of one of them, or any addition, potentially leads to a modification of all the interfaces which are generated. It is thus necessary to recompile everything in the production directory. It will also recompile `$PIPS_ROOT/src/Documentation/newgen` and `$PIPS_ROOT/src/Lib/pipsdbm/methods.h` if they have been modified in a previous step. To recompile PIPS, see section 2.2.

11.6 Global variables, modifications

Global variables should be avoided, or at least carefully documented to avoid disturbing already existing programs. Properties (variables with a user interface) may be used instead.

If new functionalities are required, it is better to keep the existing module as such, and to create a new one, with another name.

[12] Newgen types are called *domains* and are defined using a high level specification language called DDL for *Domain Definition Language*.

For global variables, initialization, access and reset routines must be provided. Do not forget that there may be several successive requests with `wpips` or `tpips`, whereas tests performs with shell interfaces (for instance `Build`) are much simpler.

11.7 Adding a new language input

- Define the language (subset) to be used by PIPS;
- gather a use-case file basis to be used as a validation of the work;
- define what transformations and analysis phases will be used and adapted to the new language for your need;
- integrate an existing parser (should be simple);
- create a preprocessor system that split a source file into as many files there are modules (functions, subroutines...) in the source file. This may use a skeleton parser for the language to detect and split module definitions from the source file;
- adapt the parser to generate the PIPS internal representation (should be more complex);
- extend the internal representation if is cannot cope with the new features and concepts of the language. It may need some deep thoughts if the language is quite different of those previously dealt by PIPS;
- modify all the needed analysis and transformations to cope with the new features added in the internal representation to deal with the new language on the selected validation programs..

That means that the work depends on the objectives and the complexity of the new language. If the Pascal language is added it should be easy, if an object-oriented is added, it should be hard...

12 Common programming patterns in PIPS

12.1 Using NewGen iterators on the RI

NewGen iterators are very effective to visit the PIPS internal representation, as any other NewGen objects. For more documentation, look at the Doxygen module *NewGen quick and intelligent recursion on objects (visitor pattern)* and its submodules.

Considering a call to:

```
gen_recurse(some_statement,
            statement_domain,
            some_filter,
            some_action);
```

it will walk on all statements (because of `statement_domain`) included in the tree-like structure `some_statement`.

Consider a module

```
void foo()
{
        int i,j;
        i=10;
        for(j=0;j<i;j++)
                i/=2;
        printf("%d\n",j);
}
```

Its structure is more or less (from the module statement point of view):

```
block statement
\_  call statement // i=10
for loop  statement
\_ block statement
\_ call statement i/=2
call statement // printf ...
```

`some_filter` will be called in a depth-first fashion (prefix-order) with the visited statement given as parameter, so that `some_filter` can investigate iteratively every statement. If `some_filter` returns `FALSE`, `gen_recurse` stop descending (but not visiting somewhere else).

That means here: first on the block statement, recurse into it, the call, the loop, recurse, the block, etc

So it recurses into a statement only if the return value of the filter is `TRUE`, otherwise it does not recurse.

Then on all statements previously visited, `some_action` is called, in a bottom-up fashion (postfix-order).

13 NewGen

NewGen is the tool used in PIPS to describe the objects which PIPS deals with. NewGen generate data structures and methods to relief the programmer from gory details in a portable way.

There are many interesting features coming with NewGen:

- automatic generation of generic methods: constructors, destructors, accessors, setters...
- language independence: since objects are described in a declarative way, NewGen generators can generate methods for any language. Right now there is support for C and Common Lisp but one can easily add generic support through SWIG for example;
- add object oriented featured to non-object-oriented languages such as C that is heavily used in PIPS;
- add automatic persistence to objects with a private or XML backend;
- add XML DOM and XPath support to the object to ease programming and to help integration with other projects;
- add various library functions for common data type such as lists, set, mappings, generic functions, strings, arrays...

13.1 XML DOOM backend

The XML backend is called DOoM in NewGen because it implements many parts of the DOM (Document Object Model) in XML.

XML support is included in NewGen if it is compiled with the `USE_NEWGEN_DOOM` macro-constant defined. Then, the XML support is used if the `NEWGEN_DOOM_WRITE_XML` environment variable is set to `1`.

MAY BE OBSOLETE DOWN FROM HERE...

14 Development (PIPS_DEVEDIR)

This directory contains the version of PIPS currently being developed. It is a mirror of the organization under `$PIPS_ROOT/src`. Each library is developed and tested under this subtree, linking by default with the *installed* libraries in `$PIPS_ROOT/Lib`.

Thus new versions can be developed without interfering with other developments. Once a new version of a library (or pass, or script, or anything) is okay and validated, it is installed under the `$PIPS_ROOT/Src` subtree and becomes the reference for all other developers who may use its services.

When developing or debugging several related libraries at the same time, you can used the development version of these libraries by using Unix links (`ln -s`) to one of the directory under the corresponding `$PIPS_ARCH` subdirectory. When building a new binary, the linker takes these local libraries first.

14.1 Experiments

At CRI, this directory is a large piece of disk space to be used temporarily to analyze large programs and to core dump large `pips` processes; no permanent, non-deductible information should be stored there.

15 Linear library

The linear library is also independent from PIPS. Its development was funded by the French organization CNRS.

Its root directory is `/projects/C3/Linear/` (`$LINEAR_DIR`). This directory is further divided into similarily to Pips and Newgen. Thus there are Production (`$LINEAR_ROOT`) and Development directories. `$LINEAR_ARCH` (which defaults to `$PIPS_ARCH` and `.` (dot)) can be used for building the library.

16 Organization of a PIPS pass

The source files for a pass `p` can be found in the directory `Development/Passes/p`. A pass, as opposed to a library, corresponds to an executable, which can be used by the users.

TODO : update this section since config.makefile does no longer exist !

In this directory, a local `config.makefile` must be created. It must be updated with the names of the source files. Then the local `Makefile` can be automatically derived using `pips-makemake -p`. This `Makefile` contains among others an `install` entry to install the pass in the `Production` hierarchy.

The libraries used to build the pass are in the directories `Production/Libs` and `Externals`. The corresponding header files are in `Production/Include` and `Externals`.

To maximize code reuse, it is recommended to limit code development at this level to functionalities really specific to passes. Everything else should be placed in libraries (for instance in `top-level`).

At the library level, it is also possible to create local executables by invoking `make test`, `make ttest` or `make wtest`.

If a pass and a library are simultaneously developed, `misc` and `parallelize` for instance, one of the library must be chosen for the link, `parallelize` for instance). It is then necessary to look for `misc.h` and `libmisc.a` in `Development/Lib/misc` instead of `Production/Libs/misc`. This can be done very simply in the config.makefile file by giving the names of the libraries to use:

```
...
# List of libraries used to build the target
TARGET_LIBS=    -lprivatize -lusedef -lprettyprint -lsemantics \
                -ltransformer \
                -lcontrol -leffects -lnormalize \
                -lsc -lcontrainte -lvecteur -larithmetique \
                -lri-util ../../Libs/libmisc.a \
                -lproperties -lgenC /usr/lib/debug/malloc.o \
                -lproperties

$(TARGET): ../Lib/misc/misc.a
```

17 Bug policy

Warning: all the following is obsolete !

17.1 Bug detection

When a bug has been detected, a small Fortran program (`bug.f` for instance) must be written to reproduce the bug.

If the library `xxx` is responsible for this bug, move bug.f in `Tests/Bugs/xxx/`. This responsibility can be found using `XXX_DEBUG_LEVEL` or a debugging tool.

Then record the new bug in `$PIPS_DOCDIR/pips-bugs.f.tex`.

17.2 Bug correction

Go to the directory `/Tests/Bugs/xxx`. In order to use the executable of the library `xxx`, create a symbolic link towards it (`ln -s Development/Lib/xxx/pips` for instance). Here are now the different steps to perform:

- Find the sources responsible for the bugs and correct them (we assume that only one library is responsible for the bug; the case of multiple libraries is presented below).
- Run `make test` in `Development/Lib/xxx` to build an executable.
- Back in `Tests/Bugs/xxx`, run:

```
Init -f bug.f bug
Display -m -v bug
```

and verify that the bug has been eliminated.

- In `Development/Lib/xxx/`:
  - Launch `Validate Xxx` to run the non regression tests specific to the library `xxx`.
  - Run `Validate` if the changes may affect the results in several phases.
  - If the results are satisfactory, run `make install` to install the new sources in the `Production` hierarchy. Beware that this does not build new executables in `$PIPS_BINDIR`. This can be done with the `make-pips` command, but this is not compulsory, because this command is automatically run each night.

Once these operations have been done, the program `bug.f` and the results obtained for its parallelization must be added to the non-regresson tests in the directory `/Tests/Validation/Xxx`. For that purpose, you can run the command `bug-to-validate bug.f`. But beware that this command is a script shell which provides a mere parallelization of `bug.f`. To transfer all the programs from the directory `Tests/Bugs/xxx` to `/Tests/Validation/Xxx`, you can use the command `dir-to-validate`. Again, this is very convenient when the desired test is the default test (see Section 17.3).

17.2.0.1 Remark: When several libraries are responsible for the bug, say `xxx1`,..., `xxxk`, chose one of the libraries to build the executable by linking with the other libraries. Once the problems are fixed, do not forget to run `make install` in all the libraries.

17.3 The `Validation` directories

TODO : update this section to the valid svn directory.

These directories `/Tests/Validation/Xxx` contain the non-regression tests for each significant library `xxx`, as well as demonstration programs, benchmarks, ...

For each Fortran file `bug.f`, there exists a test file `bug.test`, and a subdirectory `bug.result` in which the results are stored in a `test` file. All these files can be generated by `bug-to-validate bug.f` when dealing with a bug in `Tests/Bugs/xxx`. If this command has been used, the reuslt directory contains the results of a mere parallelization. For more complex results, a specific test file must be written, as well as a script inspired from `bug-to-validate` to install the necessary files in the validation directory.

If the same test file is valid for a whole directory (ex : `Validation/Flint`), a generic file `default_test` can be created in this directory. In this file, the generic names for the program are `tested_file` and `TESTED_FILE`. Here is the `default_test` script of `Validation/Flint`:

```
#!/bin/sh
Init -d -f $PIPSDIR/Tests/Validation/Flint/tested_file.f \
```

```
          tested_file 2>/dev/null >/dev/null
Perform -m tested_file flinter 2>/dev/null
cat tested_file.database/TESTED_FILE.flinted
Delete tested_file 2>/dev/null
```

Notice that if there exists a local test for the file (`bug.test`), it will be used first. The priority starts from local files to general ones.

17.4 Other validation

The command `Validate` can be used to measure the impact of a modification by comparing the new behavior of PIPS with the preceding one.

To use it to check the parallelization process, put your test file, say `mytest.f` which contains the main program MAIN and a subroutine SUB, into one of the directories in `~pips/Pips/Tests/Validation` directory. You can also create your own directory there if you want to ensure that a particular aspect of PIPS behaves the correct way.

Once `mytest.f` is in such a directory, say `kludge`, you should do the following thing.

```
Validation/kludge: Init -f mytest.f mytest
Validation/kludge: mkdir mytest.result
Validation/kludge: Display -m main > mytest.result/MAIN
Validation/kludge: Display -m sub > mytest.result/SUB
Validation/kludge: Delete mytest
```

Check that the output in the MODULE1, MODULE2, ... files is what you want ... and that's it!

After a while, if you want to check that PIPS still does that it was supposed to do, go into Validation and type

```
Validate kludge
```

If there is any problem, you will get a mail message that tells you want the problem is. You can also type `Validate` to check everything.

- `Validate` with no argument validates all the directories which are listed in the file `Validation/defaults`.
- When a directory of `Validation` is validated, if for the program `foo.f` there exists a file `foo.test` it is executed; otherwise `default_test` is ran. The output is compared to `foo.result/test` which must have been created before. The result can be as complex as desired, and not a mere parallelized version.
- `tpips` scripts can also be used. In this case, do not forget to build a local `tpips` by invoking `make ttest` after creating your local `pips`.

18 Miscellaneous

To print a nice version of PIPS output, you can use commands like `tgrind`:

```
tgrind -lf extr_doall.f
```

All the source of PIPS can be pretty-printed with Doxygen and are browsable. For example, see `http://doxygen.pips.enstb.org`.

18.1 Changing the dynamic allocation library

Errors which are the most difficult to find generally are dynamic allocation errors: The allocated space is accesssed after being freed (*dangling pointer*), or a zone larger than the allocated zone is referenced (too long copy of a character string for instance).

A first checking level is available with a set of SUN primitives (`man malloc.h`), by linking the program with `/usr/lib/debug/malloc.o`. The choice of the dynamic allocation library can be made in `pipsrc.ref`, or in the curretn environment, but without any guarantee for the link.

A second level, a lot more efficient, but also much slower, is offerd by a public domain library, `malloclib-pl11`. This library systematically overwrites freed zones with a special pattern, and checks that string operations are valid. It also records in which files allocations and desallocations are performed.

To use it efficiently, all the source files which dynamically allocate or free memory must include `malloc.h` with double quotes (and not < > signs). This is achieved in most cases because NewGen includes `malloc.h`. But some files, such as character strings manipulation files, do not include `GenC.h`: `#include "malloc.h"` must be added to them.

PIPS must then be entirely recompiled (see section 2.2), as well as the libraries in `$PIPS_EXTEDIR`, after renaming (`mv`) `dbmalloc.h` into `malloc.h` in `$PIPS_EXTEDIR` and in the directories containing the sources of the external libraries (NewGen and Linear). Before linking, the environment variable `PIPS_LIBS` must have been modified, either in the current environment, or in `pipsrc.ref`, to include the correct library.

It is recommended to keep an version of PIPS linkeed with a reasonnably fast library to be able to create large databases for the tests. To avoid compiling and linking too many things each time, it is also recommended to modify the shell variables `$PIPS_BINDIR` and `$PIPS_LIBDIR` to keep both versions of PIPS.

References

[1] Pierre Jouvelot and Rémi Triolet. Newgen : A langage-independent program generator. rapport interne EMP-CRI 191, CRI, École des Mines de Paris, July 1989. 5

[2] Pierre Jouvelot and Rémi Triolet. *NewGen User Manual*. CRI, École des Mines de Paris, December 1990. 5

[3] Ronan Keryell. *WPips and EPips User Manual*. CRI, École des Mines de Paris, April 1996. Available from `http://www.cri.ensmp.fr/pips` and as report A/288. 5

Index