revCPP: A reversible C++ preprocessor

Quentin Hocquet and Benoit Sigoure

Technical Report n^o0742, June 2007
revision 1450

Abstract: The Transformers project aims at creating a generic framework for C++ source to source transformation. "Source to source" transformation consists in refactoring the code and producing a modified source. The resulting code may be reread, reused, re-modified... by programmers and thus must be human-readable. Moreover it should respect the original coding style. This process of preserving the original layout is called "high fidelity program transformation".

Transformers targets the C/C++ language. Unlike many other languages, C++ source code is preprocessed to obtain the actual source code. In our program transformation context we need to un-preprocess the code to give back a human-readable code to the programmer.

This document presents the work and research carried out to implement a reversible C++ preprocessor and a postprocessor, i.e. a tool to obtain the original code from the preprocessed one.

Abstract: Le but de du projet Transformers est de créer un framework générique pour de la transformation source à source de code C++. Une transformation "source à source" consiste à retravailler le code et produire un fichier de code source modifié. Ce code peut être relu, réutilisé, modifié ... par des programmeurs et doit donc être lisible. De plus, il doit respecter le *coding style* d'origine. Ce processus de préservation du *layout* est appelé *"High fidelity program transformation"*.

Transformers cible les langages C et C++. Contrairement à de nombreux langages, le C++ est un langage *préprocessé* pour obtenir le code source effectif. Dans le contexte de la transformation de programmes, il faut *dé-préprocesser* le code pour le rendre lisible au programmeur.

Ce document présente le travail de recherche que nous avons mené pour implémenter un préprocesseur de C++ reversible et un *postprocesseur*, c'est-à-dire un outil permettant d'obtenir le code d'origine à partir du code préprocessé.

Keywords
revCPP, unCPP, CPP, C++, Transformers, program transformation, C preprocessor, C++ preprocessor

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
transformers@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Chapter 1

Introduction

RevCPP is a reversible C++ preprocessor released under the GNU GPL[1]. This project will be used as part of Transformers to deal with source-to-source transformations.

Although revCPP will strive to be as close to the standard as possible, our first goal is usability: we'll first cover most common use cases and we might leave some FIXMEs as an exercise for the readers interested to get a fully ISO-14882-compliant preprocessor.

This document is presented as follows: Chapter 2 presents how to install revCPP. Chapter 3 explains how to preprocess code with revCPP. Chapter 4 explains how to postprocess ("un-preprocess") the code preprocessed by revCPP. These two chapters have been written for revCPP users and can be used as reference manuals. Chapter 5 then explains the internals of revCPP and how it was implemented. It has been written for future maintainers of revCPP or anyone who would like to patch, contribute, fork or simply understand how revCPP works. Chapter 6 concludes this report.

1.1 Mission statement

Our goal is to provide a reasonably complete and standard implementation of CPP[2] as described in the chapter 16 of the C++ standard. We might provide compatibility with the C preprocessor but this is not our main goal.

This implementation of CPP adds extra-comments, called *metatags*, in its output in order to be able to reverse the preprocessing. We are not aiming at an efficient implementation, nor strict standard conformance. We want the code to be easy to understand and maintain. Our project will be successful if we can preprocess Boost and most of the test suite of GCC[3]'s CPP.

[1] GNU General Public License.
[2] C PreProcessor.
[3] GNU Compiler Collection.

1.2 Motivations

One of the numerous reasons that hinder projects trying to perform source-to-source transformations on C++ code (besides the ambiguous grammar) is CPP. CPP is a macro processor that is used automatically by C++ compilers before compilation. It is used to include a file in another one or replace a word (or more) with others.

It is thus impossible to parse a C++ source file unless it has been processed by CPP. Most of the C++ files need to include their "dependencies" including files coming from dependent libraries. For high fidelity source-to-source transformation, the user wants their compilation unit to be parsed and transformed and pretty printed back to something as close as possible to the original code. This implies that they want their layout and comments to be preserved as much as possible.

In order to do this, a *reversible* preprocessor is needed. As of today there are several implementations of CPP but none offers this feature while being free, i.e. released under the GPL.

Chapter 2

Installation

2.1 Requirements

- OCaml 3.09 or newer (3.10 recommended).
- menhir, version 20070215[1] or newer.
- ExtLib 1.4 or newer.
- Ruby 1.8.5 or newer.

2.2 Getting the dependencies

You can get these dependencies using Godi, the OCaml source distribution system:

- Download and install Godi.
- Launch godi_console (`$godi_prefix/sbin/godi_console`).
- Type "1" then return to update your package list.
- Type "x" to go to the main menu, then "2" and return to select packages.
- Look for godi-extlib and godi-menhir in the list.
- For each of them, type their number, then "b" and "x" to schedule them for installation.
- Back in the package list, type "s" and "o" to start installation.
- When done, type "x" then "3" to exit.

[1] revCPP might work with earlier versions but only the version 20070215 has been tested.

2.3 Installation from source

You can easily retrieve the sources of revCPP with subversion.

```
svn checkout https://svn.lrde.epita.fr/svn/revcpp/trunk/ revcpp
cd revcpp
./bootstrap
./configure
make
make check
sudo make install
```

Chapter 3

revCPP user's manual

revCPP is an implementation of the Chapter 16 of the ISO/IEC 14882 standard which has the ability to reverse its preprocessing. This is done by adding tags in the form of comments (named metatags) in the output file in order to preserve the information otherwise lost by the preprocessing.

Our reference implementation is GCC's preprocessor and revCPP exhibits a compatible interface so that one can use it as a replacement to `g++`.

Our preprocessor is not yet fully compatible with ISO C99 (at least it has only been tested for C++).

3.1 General usage

RevCPP works mostly like the GNU CPP implementation. It will read the text to preprocess on the standard input, or from a file specified on the command line. Unlike a classical CPP implementation, revCPP will output several files, one per possible evaluation of all the conditional inclusions[1] in the file. `revcpp foo.cc` will create `foo.1.i`, `foo.2.i`, ...in the current directory. You might specify specific constraints for these condition evaluations, as explained in Section 3.2.

Note that some behaviors will differ from the standard implementation. No error will be generated on `#error` for example, because we want to be able to preprocess this directive and postprocess it back later.

```
#ifndef HAVE_BOOST_H
# error You need boost to compile this program
#endif
```

This `#error` would be lost if the code path where `HAVE_BOOST_H`-is-not-defined was not covered. Thus it is important that the entire code be covered.

[1] `#if`

3.2 Usage

revCPP accepts most of the arguments of `gcc` and `g++`. It is usable as a replacement in a build system, for instance:

```
make CXX='revcpp <revcpp-options>'
```

revCPP accepts the following additional arguments:

`--show-tree`	Show conditional inclusions tree.
`--show-variables`	Show all macros implied in conditional inclusions.
`--show-sets`	Show all possible macro sets that generate different files.
`--count-sets`	Count this number of sets.
`-n | --constraint`	Add a constraint.
`--show-constraints`	Show all constraint, including built-in ones.

The irrelevant GCC options will be ignored. The unsupported options will raise an error. The following options are currently implemented:

`-I dir`	Add the directory dir to the list of directories to be searched for header files. Directories named by `-I` are searched before the standard system include directories.
`-D name`	Predefine `name` as a macro, with definition 1.
`-D name=definition`	The contents of `definition` are tokenized and processed as if they appeared during translation phase three in a `#define` directive. In particular, the definition will be truncated by embedded newline characters.
`-U name`	Cancel any previous definition of `name`, either built in or provided with a `-D` option.
`-v`	Enable verbose mode.
`-nostdinc`	Do not search the standard system directories for header files. Only the directories you have specified with -I options (and the directory of the current file, if appropriate) are searched.
`-nostdinc++`	Same behavior as `-nostdinc`. This differs slightly from GCC, which continues to search in the other non-C++-specific standard directories.
`-version`	Print version information and exit successfully.
`-help`	Display this information.

3.3 Specifying constraints

You can add constraints to predefined ones using the `-n <constraint>` option, where a constraint can be:

- A literal (`true` or `false`)

- A macro name (FOO)
- A logical negation (!<constraint>)
- A logical and (<constraint> && <constraint>)
- A logical or (<constraint> || <constraint>)
- A logical exclusive or (<constraint> ^ <constraint>)
- An implication
  (<constraint> => <constraint> or <constraint> <= <constraint>)
- An equivalence (<constraint> <=> <constraint>)
- A parenthesized expression.

This feature has two main usages:

- Specifying mutual exclusion or other dependencies for user-defined macros.
  For example, a user might develop a library that can use either the STL, Boost or Qt, defined by the USE_STL, USE_BOOST and USE_QT macros. By adding the USE_STL ^ USE_BOOST ^ USE_QT constraint, all generated macro sets will contain one and only one of these macros, thus avoiding generating useless files that wouldn't even compile. As another example, the user may define a macro USE_QSTRING to replace std::string's with QString's. He might then want to add the !USE_QT => !USE_QSTRING constraint. Thus sets that use Qt with std::string will be generated, but not inconsistent ones that does not use Qt but use QStrings.
- Refining the transformation.
  If the user wants to perform his transformation on Linux-specific code, or without debugging support. This can be obtained by adding the LINUX or NDEBUG.

There are some default built-in constraints, such as LINUX ^ WIN32.

3.4 Known bugs and limitations

- Stringification (#FOO) and concatenation (FOO##BAR) operators are not yet handled.
- Preprocessing might be slow when using many complex logic expressions with disjunctions.

Chapter 4

unCPP user's manual

4.1 Usage

UnCPP transforms a set of transformed files, resulting from the different conditional inclusions, in several postprocessed files corresponding to the original file and its inclusions. All the files will be serialized on the standard output, separated by headers indicating file names. You can also directly overwrite files instead of printing to the standard output with `--in-place`.

Consider the following example:

```
/* file foo.hh */
#if A
/* comment */
#endif

/* file bar.hh */
#include "foo.hh"
```

Invoking revcpp on `bar.hh` will generate `bar.1.i` and `bar.2.i` (for the `A` and `!A` sets). Invoking `uncpp` on these two files will generate on the standard output:

```
>>> FILE /path/to/files/foo.hh
/* file foo.hh */
#if A
/* comment */
#endif
>>> FILE /path/to/files/bar.hh
/* file bar.hh */
#include "foo.hh"
```

This system will allow the modification of the included files. This can be disabled with the `--preserve-includes` option.

UnCPP accepts the following command line arguments:

`-i	--in-place`	Directly overwrite files, do not print on the standard output.
`-p	--preserve-includes`	Do not overwrite includes, and fail if they are modified.

4.2 Known limitations

- The merge algorithm might simplify `#if`. For example, `#if A && A` will be simplified in `#if A`, or two contiguous blocks with the same condition will be merged. This is due to the fact that conditionnal blocks (guarded by "if"-sections) are not simply preserved with a metatag. All combinations of if sections are explored (see Section 5.3.6) and the result is merged back again (see Section 5.4.4). Since we recreate the if-sections, they might be different, although semantically equivalent.

Chapter 5

Behind the scene: revCPP internals

This chapter documents the internals of revCPP (for both the preprocessor and the post-processor) and presents the tricky issues and workarounds used throughout the code. The first source of documentation should remain the comments in the code (which is fairly well commented) and it is possible that future evolutions of revCPP will make this document deprecated. Future maintainers are encouraged to maintain the comments in the code rather than this document, should they chose between one alternative or the other.

Thus this section is only here to help the reader to understand and maintain the project, but reading the code alone should be enough to be able to work on it.

This section assumes that you already know OCaml, how to use `ocamllex` and `menhir` and that you have already read and understood (ISO/IEC, 2003, chap. 16).

5.1 Modules

5.1.1 Foreword

During the development of revCPP we felt the need to implement several helping modules. Some of them were made quite generic and are probably re-usable under other circumstances. Some other are re-used throughout the project and did not fit in any particular section. That is why they are documented here.

5.1.2 Getopt

We use a modified version of Getopt to handle GCC-style arguments. This implementation of GNU getopt is almost seen as a standard in the OCaml community and is distributed in Godi.

The problem with this implementation is that it can only handle short options of the form "`-X`" where *X* is a single character or "`--foo`" where *foo* is any string, just like GNU getopt. GCC, however, uses most of its long options of the form "`-foo`". Since the code of Getopt mainly consisted in a single ML file, we decided to fork it. The code

was cleaned up a bit (mainly factored so that it is more readable). The change in itself is rather simple: when an argument starts with a dash, it is first checked against known long options. If none matches, it is then checked against short options.

5.1.3 ScanUtils

We use ocamllex for all our scanners, which is part of the standard OCaml library. During the development we were hindered by the limitations of ocamllex. We had difficulties handling locations properly (although menhir seems to provide facilities to deal with them). We also needed to associate a context with the scanner in order to store various things such as a stack of states (that we use to choose the start condition of the scanner), the layout seen since the latest token and whether the latest token was an end of line or not. We also wanted to share scanning rules among different sub-scanners in order to minimize the redundancy of the rules valid in many states.

This section explains how we tackled each of these problems. Note that all our scanners are re-entrant.

5.1.3.1 Handling locations

The only feature automatically handled by ocamllex is the count of the number of characters since the beginning of the file. The module `ScanUtils` provides the following functions:

The function `init` initializes the `lexbuf` of ocamllex with a file name.

The function `next_line` is used to increment the current line number.

The function `scan_error` is useful to raise a `Fatal` error with an error message that contains the current location of the scanner.

The functor `Context` holds the current context of the scanner. It must be instantiated with a module that defines a type `t` and a value `string_of_state` that converts values of type `t` to strings. The type `t` is used to represent the current state of the scanner. The module provides a type `context` that holds:

- `state_stack`: the name says it all.
- `layout`: a string that holds the layout seen since the latest token.
- `last_token_is_eol`: a Boolean (here again the name says it all). This value is used only in the scanner of revCPP in order to detect preprocessing directives (that is, when a "#" is the first token on a line).

The implementations of scanners should not rely on the internal representation of the `context` type.

The module returned by the `Context` functor contains the following functions:

- `init_context`: instantiate an empty context with a first state given in argument.
- `push`: push a state on the state stack.

- `peek`: return the current state.
- `pop`: pop a state off the state stack.
- `nol`: specify that a token has been seen and that we are thus not right after an end of line (mnemonic: "Not end-Of-Line"). This function is useless out of the preprocessor scanner.
- `eol`: specify that the token about to be returned to the parser is an end of line. This function is useless out of the preprocessor scanner.
- `append_layout`: store some layout in the context. This way the layout is saved between the calls of the scanner.
- `layout`: return the layout stored so far and sets the current layout to the empty string.

5.1.3.2 Associating a context with an ocamllex scanner

As usual, the parser calls the scanner. The rules labelled `%start` are OCaml functions that must be invoked with two arguments: the scanning function and the input stream of the scanner. In order to be re-entrant, our scanner needs to be invoked with the `context` argument by the parser. This is straightforward in OCaml thanks to partial applications:

```
MyParser.entryPoint (MyScanner.lex context) lexbuf
```

5.1.3.3 Sharing rules between sub-scanners

Although ocamllex supports sub-scanners (they are called "entry points") it provides no way to share rules between them. This is very unfortunate because the scanner of the preprocessor has numerous sub-scanners that must handle the layout almost identically in each sub-scanner. Even worse, a couple of sub-scanners are identical to the main one excepted for a couple of rules.

The idea to share rules between the sub-scanners is to use a wildcard to match a single character and to push this character back in the scanner's internal buffer (hereafter named the `lexbuf`) and to invoke another sub-scanner. This way, a sub-scanner can first override or extend a set of rules and then, if nothing matches, it can "delegate" the scanning to another sub-scanner.

```
rule lex_initial context = parse
  | (* rules *)
and lex_foo context = parse
  | (* other rules *)
  | _ as c { push_back lexbuf c; lex_initial context lexbuf; }
```

After a quick analysis of the interface exposed by the `Lexing` library (the official interface of ocamllex) we found that we could access the scanner's internal data structure. The module `ScanUtils` thus provides a function `push_back` that accesses the internal structure of ocamllex to push a character back in the `lexbuf`.

This function was implemented by observing the way ocamllex used its internal data structure and by quickly going through the code of ocamllex and the `Lexing` library. It is by no means bullet-proof and must be used with great care. As explained in the comment above the definition of this function, only characters matched by the current semantic action must be pushed back in the `lexbuf`, no more.

An additional function `push_back_string` is provided for convenience in order to push back all the characters of a string in the `lexbuf`.

There is a pitfall due to the way the layout is handled. Most of the time, it is discarded (but saved) and the current sub-scanner invokes itself recursively. Consider the following example:

```
rule lex_initial context = parse
  | layout { append_layout context lexbuf;
             lex_initial context lexbuf; }
  | '#' { do_something; }
  | (* Other rules *)
and lex_foo context = parse
  | '#' { do_something_else; }
  | _ as c { push_back lexbuf c;
             lex_initial context lexbuf; }
```

Here `lex_foo` is basically trying to override the action taken for "#" but this is flawed. If the input starts with layout, the wildcard will match and `lex_initial` will be invoked. It will do something with this layout and invoke itself recursively. If the next token is a "#", then `do_something` will be executed instead of `do_something_else`. In order to avoid this problem, we had no other choice than to duplicate all the rules that call themselves recursively (basically the rules that handle the layout).

5.1.4 Env

This module maintains an environment during the preprocessing. It mainly defines a parameterized type `env` and some functions to manipulate it.

The type `env` holds a list of macros and (macro-)functions as well as an "ID" counter. This ID is needed because is expansion is tagged with a unique ID. Reading the interface of this module will suffice for the reader to understand it.

5.1.5 MetaStd

The Caml List library provides extremely useful generic functions on list, such as `map`, `fold_left`, `filter`...During the project, we felt the need to use these algorithms on other structures than lists. For example, to collect `#if` nodes from the AST, we'd like to

be able to write: "`filter (function If _ -> true | _ -> false) my_ast`". The MetaStd module describes the `ITERATOR` signature and the `Meta` functor . By implementing an `Iterator` on their own data structure, and applying the `Meta` functor to it, a user can use standard list algorithms on their data types.

MetaStd is used to manipulate the AST of CPP, C++ and logic expressions elegantly.

5.1.5.1 The ITERATOR interface

```
module type ITERATOR = sig
  type input
  type internal
  type output
  val init: input -> internal
  val next: internal -> (output * internal) option
end
```

- `input` is the iterated type (e.g. an AST).
- `output` is the type pointed to by an iterator (e.g. an AST node).
- `internal` is an internal type the `ITERATOR` can use to carry information from an iteration to the next.
- `init` transforms the input data into an iterable data.
- `next` takes an iterable data, and returns either the next element and the remaining elements or `None` if there is no more elements.

5.1.5.2 Example

This example redefines the standard `List` module's algorithms over int lists using MetaStd:

```
module ListIterator = struct
  type i = int list
  (* Our internal storing type is still an int list *)
  type internal = int list
  type o = int

  let init l = l
  let next = function
    | h :: t -> Some (h, t)
    | [] -> None
end

module List = MetaStd.Meta(ListIterator)
```

```
let _ =
  List.map (fun x -> x + 1) [41; 50; 68]
```

Et voilà. It would be possible to define a generic List module by making `ListIterator` a functor instead of hard-coding `int`.

5.1.6 The syntax extension

We use a syntax extension for the OCaml language. Originally written with CamlP4, we had to drop it because we wanted to be compatible with both OCaml 3.09 and OCaml 3.10. It is a good thing that CamlP4 was completely redesigned and improved, but the fact that the OCaml team made no effort providing a backward compatible interface is very disappointing. Since the new interface of CamlP4 is still unstable and undocumented, we had to drop the CamlP4 extension completely. We needed to port the code to 3.10 because 3.09 does not work properly on new Intel-based MacBooks.

The `-pp` parameter of `ocamlc` makes it possible to preprocess OCaml sources with any external preprocessor. We designed a small Ruby script that does the same thing as the extension we originally wrote in CamlP4.

The following syntax extensions are thus used throughout the project:

- `"$s"` where `s` is a `string`.
- `"$#i"` where `i` is a `int`.
- `"${e}"` where `e` is a `string` expression.
- `$__LINE__` expands to the current line number.
- `$__FILE__` expands to the current filename.
- `$__LOC__` is equivalent to `$__FILE__: $__LINE__`.

As a side effect, when the compilation fails, temporary files are left in `$TMPDIR` (usually `/tmp`) by `ocamlc`.

5.1.7 The logic module

The logic modules defines a ternary logic library. Indeed, any macro in CPP can be either *true*, *false*, or *undefined*. The logic library includes:

- A logic expressions type.
- A logic scanner/parser, to handle constraints passed on the command line.
- Desugaring system, that converts any logic expression in an expression composed exclusively of `And`, `Not`, `True`, `False` and `Undefined`.

- An evaluator.
- A canonizer, that transforms an expression in a normal form.
- A constraint propagator, that simplifies an expression knowing that another expression is true.

5.2 General method

One of the main challenges of revCPP is to be able to preserve the layout at all stages of the transformation. The only difficulty is that we must be very careful in our code to avoid losing it when transforming our internal AST.

The project is organized this way:

- `src/utils` holds the various modules that are re-used throughout the project and that were described in Section 5.1.
- `src/extension` contains the syntax extension described in Section 5.1.6.
- `src/cpp-parser` contains the main scanner and parser of the preprocessor (revCPP). This is where the AST is defined, in `cppAst.ml`.
- `src/cpp-pp` contains a pretty printer for the CppAst.
- `src/cxx-parser` contains the scanner and parser of the postprocessor (unCPP). Here also, a small AST is defined, in `cxxAst.ml`.
- `src/cxx-pp` contains a pretty printer for the CxxAst.
- `src/logic` contains a logic evaluator and constraint propagator.
- `src/revcpp` is where the main code of the preprocessor lays.
- `src/uncpp` same thing but for the postprocessor.

The file is processed almost normally, as specified in the Chapter 16 of the ISO/IEC 14882 standard. All preprocessor directives are replaced with metatags, special comments of the form /∗# *TagName* <*tag data*> #∗/. All forms of expansions are also tagged with metatags. The postprocessor uses only these metatags to recover the original files.

If the input file of revCPP already contains comments that would inadvertently look like a metatag, they are escaped. Also since `<tag data>` can be anything, comments are also escaped (because we often store C++ code in this data) because C++ comments do not nest.

5.3 RevCPP

The overall goal of revCPP is to transform a CppAst into a CxxAst. In this section we will go through the whole pipeline and explain each part.

5.3.1 The CppAst

The module `CppAst` defines two major enumerated types: `entity` and `code`. Entities are preprocessing directives and lines of code. An entity can also be a `Meta` entity, which is a convenient way to nest entities into each other.

The preprocessing does not accept any kind of CppAst as input, it must first be simplified with `CppAst.simplify`.

The second major type, `code`, holds all the constructs we are dealing with, that is, the preprocessing tokens, an AST for constant expressions, and result of the various expansions.

These two major types are wrapped in a record type prefixed with a `l` (that is: `lentity` and `lcode`) that holds additional layout associated with the node.

The module `CppAst` provides some functions to easily create nodes (namely, `mkent` and `mkcode`), dump AST nodes in a simple format and iterate over the AST.

5.3.2 The scanner

The scanner uses many regular expressions to implement the various lexical rules given in the form of grammar rules in the standard. These regular expressions comply with the strict minimum required by the standard, however they probably do not handle universal character sets. Since they are implementation defined, this is not of an issue.

The scanner uses 6 sub-scanners and has 7 possible states. We will now describe each of them, when and why they are used.

One of the limitations of the scanner that makes it non standard compliant is that it does not handle backslashed end of lines. Normally they can appear really anywhere but we only handle them when they don't split a token in two.

5.3.2.1 Initial

As its name implies, it is the initial state. It scans everything at the top-level of the input file and enters the sub-scanners. It returns preprocessing tokens to the parser.

There is nothing special to report here, besides that when a sharp token ("#") is seen, if it is the first token of the line, it is not returned but instead added in the current layout and the scanner enters the state `CPPDirective`.

5.3.2.2 CPPDirective

This state is used whenever we just encountered a sharp at the beginning of the line. Its role is to handle the different preprocessing directives ("if", "ifdef", "include", "define", etc).

For most of the directives, the scanner will simply return the corresponding token the parser and go back in the state `Initial`. There are several exceptions:

- `if` and `elif` are followed by a constant expression. Several tokens need to be handled specially in this context so the scanner will enter the state `Constant-Expression`.
- `include` is followed by a sort of double-quoted string or a string delimited by curly brackets ("<" and ">") but they have to be scanned with different rules then standard strings. That is why in this case the scanner will enter the state `Include`.
- `define` is followed by something very similar to what we can find in the state `Initial` besides that the token "#" is always a `Sharp` token (the stringification operator) and "##" is always a `SharpSharp` token (the concatenation operator). So in this case, the scanner will enter the state `DefineBegin`.

If nothing matches, the scanner will assume that this line is called what is called a "#-line" and enter the `Line` state.

5.3.2.3 DefineBegin

This sub-scanner is used to handle a unique special case in the world of the preprocessor. After a "`#define`" there should be either an identifier or an identifier followed by "the left-parenthesis character without preceding white-space". This sub-scanner is also used to check that the user is not trying to define a macro named with the name of an "alternative token" (such as "and", "or", "not", "defined", etc...). Once the scanner knows whether a macro or a macro-function is being defined, it will enter the state `Define`.

5.3.2.4 Define

This sub-scanner is identical to that of the `Initial` state excepted that it always returns "#" as a `Sharp` token and "##" as a `SharpSharp` token. This sub-scanner will go back in the state `Initial` after an end of line.

5.3.2.5 Include

This sub-scanner handles the header names (delimited by double quotes or curly brackets "<" and ">") and then goes back in state the `Initial`.

5.3.2.6 ConstantExpression

This sub-scanner handles specially all the tokens that can be possibly found in a C++ constant-expression, in the context of the preprocessor. This includes tokens such as `true`, `false` and `defined` but also operators (which are normally all returned with the same constructor because they are all seen as preprocessing operators in the `Initial` state).

Some operators and tokens are also disabled here (such as the various assignment operators, braces and brackets) because they just do not make sense in this context. Floating point literals are also matched and refused.

The scanner will go back in state `Initial` after an end of line.

5.3.2.7 Line

This sub-scanner is exactly like the `Initial` scanner excepted that it will pop the current state after an end of line.

5.3.2.8 Other constructs

The CppLexer provides two functions that are placed here solely because it was the only place where the dependency system of OCaml would allow it.

The function `code_of_string` is used to transform a string in a list of `code`. The function `code_of_macro_def` is used to transform the string that holds the definition of a macro (or macro function) in a list of `code`.

5.3.3 The parser

Unlike Transformers, we did not try to strictly abide by the grammar rules of the standard. Instead we started off with the standard grammar and then we simplified and cleaned it up.

Most tokens hold at least one string, the layout preceding them.

5.3.3.1 Entry points

The parser has 4 entry points:

- `preProcessingFile`: the main entry point.
- `code`: used by `code_of_string` and `code_of_macro_def` in `CppLexer`.
- `constantExpression_of_string`: used to transform a string in a constant-expression.
- `parseDefine`: used to parse the "`-D`" arguments.

5.3.3.2 AST simplifications

Most entry points will return a simplified AST in which `Meta` notes were removed (flattened in the top-level AST) and consecutive `Code` nodes merged in a single `Code` node (the same thing is done for `Eol` nodes).

The `Meta` nodes are removed because they would make it rather inconvenient to work on the AST during the preprocessing.

The sequences of `Code` are merged in order to reduce the size of the AST. During this merge, `Eol` nodes are merged together. This makes it possible to have some optimizations such as the detection of traditional CPP guards (although this optimization is not implemented at the time of writing this report).

5.3.3.3 Constant expressions

When the code is first parsed, constant-expressions are parsed with a simplified parser which simply returns a list of tokens. Then in a later step, the preprocessing will try to macro-expand the constant-expression which will then be re-parsed in order to eventually return the final AST of the constant-expression.

This mechanism is necessary in order to be able to handle this kind of construct:

```
#define FOO 3*
#if FOO 1+2
```

5.3.3.4 Layout in constant expression operators

Most operators have a unique representation (e.g., "+" or "*") but some have more. They may be written with an alternative token (e.g., "and", "not") or with digraphs or trigraphs. For these operators, in order to preserve the original syntax, the entire operator is saved in the layout. That is, instead of having something like:

CppAst.OpOrPunc("+")<< >>

which represents a plus operator preceded by a single space, we will have something like:

CppAst.OpOrPunc("")<< and>>

for the logical binary "and" operator preceded by a single space of layout.

5.3.3.5 if-sections

The `CppAst` has only one node ("`If`") to hold if sections. At parse time, the parser will desugar the various possibilities ("ifdef", "ifndef" and "elif") so that they are all represented with a single `If` node.

The pretty-printer can always match these patterns to re-create these nodes. This is the only place where the front-end of revCPP does not exactly preserve the original input. This is not a problem since the whole process cannot always preserve the if-sections anyway.

5.3.4 File inclusion

At parse time, "`#include`" directives are stored in the AST. The actual file inclusion will occur later, during the preprocessing.

Like standard preprocessors, revCPP needs to know where to look for standard headers. The user could be asked to invoke revCPP with "`-I`" arguments pointing to the standard include directories but this would be tedious and inconvenient for users.

RevCPP uses g++ at configure time to guess these standard include directories. The problem is that g++ never prints C++ standard include directories, only C standard directories are printed. It turns out that all installs we used so far have their C++ headers under the C standard directories. So configure uses these directories as a search list for C++ headers.

If the user does not use g++ or if configure cannot find all standard include directories, the user can use `--with-std-include`.

These directories are then transformed in a Caml list of strings and used to generate the file `sysInclude.ml`.

5.3.5 Macro expansions

One of the main role of the preprocessor is to macro-expand the code. The function `Preprocess.expand_macros` performs this task by traversing the AST (which is represented as a list of `lentity`).

Unless otherwise specified, all the functions described in the section belong to the `Preprocess` module.

5.3.5.1 Macro definitions

`#define`s are simply recorded in the environment and transformed in metatags.

The `#undef` directive is also simply transformed in a metatag and the corresponding macro is removed from the environment if it existed.

5.3.5.2 Macro expansion algorithm

The algorithm behind `expand_macros` is rather simple. First off, if the unary operator `defined` is used (and is thus followed by either an identifier or a left parenthesis, then an identifier, then a right parenthesis) the identifier is obviously not macro expanded.

If an identifier is followed by a left parenthesis and is known to be the name of a macro function defined earlier, `expand_macro_function` is used to expand this macro-function invocation (see next section).

Finally, if an identifier (not followed by a left parenthesis) is known to be a macro defined earlier, its definition is looked up in the environment, then macro expanded, then finally replaced in the AST (with metatags added just before and after the expansion).

5.3.5.3 Macro-function expansions

The first thing to do when expanding a macro-function is to collect its arguments. This task is carried out by the function `find_args` which basically looks for the right

parenthesis that matches the left parenthesis located right after the name of the macro-function in the invocation. Arguments are delimited with commas but there is a small pitfall: the intervening matched pairs of parenthesis must be skipped while gathering the arguments.

```
#define FOO(X) <X>
FOO((,))
```

In this example, the macro `FOO` is passed only one argument: `(,)`.

Once the arguments are collected, the function `expand_macro_function` first checks whether there is the right number of them. Then the arguments are expanded in the definition of the macro function. When an argument is expanded, it is tagged and the result of the expansion is macro-expanded again, as required by the standard (16.3.1). In order to avoid messing up the locations of the compiler error messages, we transform all end of lines that could appear in the macro argument in a single space.

Now that the arguments have been expanded, the final result is macro expanded again and the final result is tagged in the AST.

5.3.5.4 Macro expansions in constant-expressions

They are not allowed. At this stage of the preprocessing, constant-expressions are not yet fully parsed (see Section 5.3.3.3) so this should never happen.

5.3.6 Conditional inclusions handling

All forms of if-sections are desugared by the parser in a single form of if statement (see Section 5.3.3.5). The function `eval_if` is responsible of the macro-expansion in constant-expressions and will use the module `ConstantExp` to evaluate the resulting expression.

To be able to postprocess the code, we can't simply evaluate conditional inclusions and remove parts of the code: we have to generate the multiple possible files, let the user apply his transformation over every file and merge them all back. This is done by collecting all macros implied in conditional inclusions, compute all possible affectations sets, and generate a preprocessed file for each of these sets.

Some sets are inconsistent, for instance consider a set where `LINUX` and `WIN32` are true. The resulting file wouldn't make sense, and would even probably not compile. To avoid generating it, the user can define constraints, as explained in Section 3.3.

5.3.7 Other CPP directives

5.3.7.1 #-line and no-op

These two directives are simply transformed in the metatags. Note that this implies that the locations reported by the compiler will be wrong because revCPP does not produce nor preserves the `#-lines` in the code.

5.3.7.2 #error and #warning

Although `#warning` is a GCC extension, it is supported by revCPP. Both `#error` and `#warning` will emit a non-fatal warning on stderr and will then be transformed in metatags.

This behavior does not exactly comply with the C++ standard because 16.5 requires that `#error` makes the program ill-formed. Since the standard does not seem to require that no output file be produced for ill-programs, this is not a big infringement. This, however, requires that the transformation toolchain ignores the file produced (which is most likely to be invalid) while still postprocessing it like other transformed files.

5.3.8 Reducing the number of output files

The first naive implementation of the conditional inclusions collected all conditional macros, computed all the combinations, and discarded those that did not verify the constraints. It worked for testing purpose, but the algorithm was exponential in the number of conditional variables in the file. Since some standard include files (like `stdio.h`) contain more than 30 variables, this system would not scale.

The actual system only computes sets that verify all constraints. Moreover, it merges equivalent sets. Actually, it does not compute sets of assignments for each variable, but sets of constraints to verify.

5.3.8.1 Example

For example, consider this file:

```
#ifndef FILE_HH
# if FOO
# else
#  if BAR || BAZ
#  endif
# endif
#endif
```

The resulting sets would be:

- `!FILE_HH`
- `FILE_HH && FOO`
- `FILE_HH && !FOO && (BAR || BAZ)`
- `FILE_HH && !FOO && !(BAR || BAZ)`

Note that only the strictly necessary sets were generated, and that every set is sufficient to evaluate every condition. This set computation also takes constraints in account. If we add the `!FOO => BAR` constraint, the resulting sets are:

- `!FILE_HH`
- `FILE_HH && FOO`
- `FILE_HH && !FOO`

These sets are not obtained by computing the previous sets and then discarding those not verifying the constraints, but by not exploring some branches.

5.3.8.2 Algorithm

The set computation system works as follow:

- A condition tree is built. A tree is a list of nodes. Each node contains a condition, and the "then" and "else" branches, which are trees.
- The tree is traversed: constraints are applied to the condition.
  - If the condition evaluates to true or false, the corresponding branch is visited.
  - Either, the "then" branch is evaluated with the condition added to the constraints, the "else" branch is evaluated with the negation of the condition added to the constraint, and the two results are concatenated.
- The algorithm is repeated with the new constraints set on the next node.

5.3.8.3 Results

This system computes the minimum number of files. However, the constraint propagation algorithm is in "exponential of the number of `and` in the expression" complexity. This might be of a problem with very complex conditions, and work is currently being carried out on possible optimizations. Unfortunately, after asking persons skilled on this topic, exponential may be the least complexity to solve this problem. More work on this part of the project is needed anyway.

5.4 UnCPP

The overall goal of UnCPP is to transform a `CxxAst` in a gross `CppAst` and pretty print this AST. In this part (called the "postprocessing") we only look at the metatags of the input file in order to reverse the preprocessing. The scanner and parser are thus quite simple.

5.4.1 Inclusions handling

Reverting inclusions (`#include`) is straightforward: Inclusion start and end metatags are detected, and files body are extracted. By default, all the files are postprocessed and printed on the standard output.

If `--in-place` is present, the original files are overwritten (their absolute paths are contained in the inclusion start metatag).

If `--preserve-include` is present, the postprocessed files are compared with the files on disk, and an error is raised if differences are found.

5.4.2 Conditional inclusions handling

The first step of the postprocessing pipeline is to collect each file present in the preprocessed (with `collect_includes`) source and postprocess them separately. This work is carried out by `allcpp_of_cxx`.

5.4.3 Macro handling

To revert macros and macro function definitions, the corresponding metatags are simply inlined. For instance, `/* MacroDef FOO BAR */` is postprocessed in `#define FOO BAR`.

Reverting macro expansion is quite simple:

- Expansion site is detected thanks to the metatags.
- The expanded body is compared with the macro definition (modulo the expansion of the arguments).
  - If they match, the expansion is replaced with the macro call.
  - If they differ, the expanded body is left (and metatags are, of course, removed).
- If none of the expanded body matched the definition, but they are all identical, the macro definition is altered.

Reverting macro-function expansions is more challenging. Consider the following example:

```
#define C ,
#define FOO(X) BAR(X)
#define BAR(A, B) <B - A>
FOO(a C b)
```

The (simplified) preprocessed result is:

```
#define C ,
#define FOO(X) BAR(X)
#define BAR(A, B) <B - A>
</* MacroEnd C */ b - a /* MacroStart C */>
```

Note that the comma resulting from the expansion of `C` disappeared, because it became an argument separator for the macro-function `BAR`, and that its enclosing tags are inverted.

To handle such a case, postprocessing operations must be performed in this order:

- Postprocess the expanded body without touching the arguments.
- Revert the macro function itself.
- Postprocess the arguments list.

To be able to do this, inlined arguments are also surrounded by metatags. A closer version (yet quite unreadable) of the preprocessed result would be:

*/∗FunStart FOO∗//∗FunStart BAR∗/</∗ArgStart BAR B∗//∗MacroEnd C∗/***b**)*/∗ArgEnd FOO X∗//∗ArgEnd BAR B∗/− /∗ArgStart BAR A∗//∗ArgStart FOO X∗/*(**a** */∗MacroStart C∗//∗ArgEnd BAR A∗/>/∗FunEnd BAR∗//∗FunEnd FOO∗/*

5.4.4 Conditional inclusions handling

The merge step consists in merging all the files generated by the different constraint sets in one single file, sliced according to `#if` directives. Two files are merged using a diff-like algorithm. Any line present in one file but not the other is included in the resulting file, guarded by a `#if` on the constraint for the file. The result is then merged with the next file, and so on.

A simplification step then occurs:

- Condition of `#if` is propagated in the "then" branch, and its negation is propagated in the "else" branch.
- Logical expressions are canonized, to simplify expressions such as `A && A` or `!!A`.
- Compatible contiguous blocks are merged:
  - Two contiguous `#if` blocks with the same condition are merged.
  - Two contiguous `#if` blocks with opposite condition are merged with `#else`;

The first implementation was a naive, four-line exponential algorithm, well suited for simple tests. This algorithm not being able to handle 15 line long files, it was quickly replaced with a dynamic algorithm, linear in time but quadratic in space. To handle very large files such as standard includes, it was finally replaced by an algorithm linear in space and time, based on D. Hirschberg's work on the longest common subsequence problem Hirschberg (1975).

Chapter 6

Conclusion

In this paper we explained how to use revCPP and how we successfully implemented a reasonably complete C++ preprocessor and its postprocessor. We achieved high fidelity postprocessing by preserving the layout at all stages of the preprocessing system. This is of utmost importance for anyone willing to work on the transformed source code.

This implementation of a reversible preprocessing is close to that used by Proteus (Waddington and Yao (2005)) but is the first to be free[1]. Now that we have a free reversible preprocessor, we will be able to integrate it as the first and last components of the transformation pipeline of Transformers.

RevCPP does not yet support C because it was only written for C++ but the implementation is not bound to C++ and can easily be ported to have a C89 and C99-compatible mode

By exhibiting an interface compatible with that of GCC, we hope to make of revCPP a practical tool that is easy to use with existing build systems.

[1] free as in the GNU Public License

Chapter 7

Bibliography

revCPP's trac, http://trac.lrde.org/revcpp.

SVN repository https://svn.lrde.epita.fr/svn/revcpp/.

Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343.

ISO/IEC (2003). ISO/IEC 14882:2003 (e). programming languages — C++.

Waddington, D. G. and Yao, B. (2005). High-fidelity c/c++ code transformation. *Electr. Notes Theor. Comput. Sci.*, 141(4):35–56.

Index