What is phc?

phc is a framework for static analysis of PHP scripts, PHP source to source transformations, and ultimately compiling PHP scripts down to native machine code.

The current release **does not yet compile PHP** and is therefore not yet useful for end-users. It is however useful for writing tools that operate on PHP scripts, such as refactoring tools, aspect weavers, or obfuscators (see Spinoffs for some other suggestions). See What's in Store to get an idea of what is planned for next releases of phc.

phc provides the programmer with a well-defined representation of PHP scripts (an abstract syntax tree or AST), an API to define analyses and transformations over this representation, and an unparser to output the AST back to PHP syntax. See the Demo for a quick idea of what phc can do for you, or the Documentation to get started with phc.

New: XML support

If you are not comfortable with C++, or for some other reason would prefer not to work directly with the phc framework, phc might still be useful to you, because it can output the AST to XML format. Moreover, phc is also able to read this XML back in, and can write it back to PHP syntax. This means that you can write applications that operate on PHP scripts in XML syntax, using our abstract syntax for PHP, and use phc to convert from PHP syntax to XML syntax and back.

Contact Us

We are more than happy to answer any questions about phc. So please do not hesitate to join the phc mailing list, where we will try our best to answer any queries as quickly as we can. You can join below.

Email

News

8 September 2006. I am delighted to announce the first spin-off based on phc. Written by Daniel Barreiro, it embeds HTML/XML into PHP. See spinoffs for details. In further news, we have released version **0.1.7** as no major programs were reported in 0.1.7rc2, and we have added a new article on writing a reentrant parser with Bison and Flex.

21 July 2006. Fixed a few (minor) user-reported bugs in **0.1.7rc2**.

13 July 2006. Released version **0.1.7rc1**. We now support extending phc through **plugins**: it is no longer necessary to re-compile phc to add functionality to it! This also should make binary packages for phc much more useful. See the ChangeLog for more details.

7 July 2006. Bugfix (**0.1.6rc2**). Scripts with open ended PHP (no closing tag) had an

extraneous echo at the end; also, use of identifiers as array indices inside strings caused the identifier to be duplicated in the string to follow the variable access.

4 July 2006. After a long period of no releases, we are happy to announce version **0.1.6rc1**. New in this release are an improved tree transformation API, support for pattern matching on AST nodes, an XML unparser, and various other features and bugfixes. As always, see ChangeLog for details.

See the news archive for older news.

$LastChangedDate: 2006-09-08 13:50:57 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Home | Downloads | Documentation | Plugins | Spinoff Projects | Mailing List

News Archive

27 January 2006. Version 0.1.5 did not compile on systems with old versions of Flex. Our apologies. This is solved in **0.1.5.1**.

26 January 2006. Version **0.1.5** released. phc is now released under the **BSD license**. This release also contains numerous minor feature enhancements and bug fixes; see ChangeLog for details.

9 January 2006. Lots of minor feature enhancements. See ChangeLog for details. **IMPORTANT**: this release is not downwards compatible with the previous!

1 December 2005. Bug fix. Null vectors are now unparsed properly by the dot unparser, and Windows-style line-breaks are handled properly.

17 November 2005. Added functionality to the TreeTransform API to modify the traversal order. See Tutorial 5 for details.

2 November 2005. Bug fix. Strings containing variables were not parsed correctly; corrected in release **0.1.1**.

29 September 2005. First release of phc! This release offers the framework for modifying PHP (parser, unparser, tree transformation interface) but as yet nothing else.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Authors

phc is written by Edsko de Vries, John Gilbert and Paul Biggar.

Acknowledgements

The following people have submitted bug reports/patches/ideas (in no particular order, hopefully we haven't forgotten anyone): David Abrahamson, Sven Klemm, Andras Biczo, Daniel Barreiro, Andreas Korthaus, Conor McDermottroe, Daniel Fabian, Dan Libby.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Documentation

Since version 0.1.7 the recommended way of extending phc is through plugins. Getting Started explains how to compile plugins for phc.
Because phc does not generate machine code, it not yet useful to end users. However, it can be used as a framework for writing applications that process PHP code. phc parses PHP code into an internal representation known as an **abstract syntax tree** or **AST**. Applications can process PHP code by analysing and modifying this abstract representation in one of two ways:

- phc supports **plugins**. Plugins are modules that can be loaded into phc, which get access to the AST. phc provides sophisticated support for writing operations over the AST through the **Tree Transformation API**.
- Alternatively, you can export the AST to **XML**. You can then process the XML in any way you like, and then use phc to convert the XML back to PHP syntax.

The **Usage** documentation below explains how to explain and run phc, and how to convert from PHP to XML and back. **Writing Plugins** explains how to write plugins for phc, and provides numerous examples. You will find the **Reference Documentation** very useful when writing serious applications using phc. Finally, **Miscellaneous** lists a few other articles you may find interesting.

Note that you can download the documentation for this and for older versions from the archive. Moreover, although we have tried to document phc as well as we can, if anything is still unclear, please let us know by sending an email to the mailing list.

Usage

Installation Instructions

The Installation Instructions explain how to compile phc after you have downloaded it.

Running phc

Running phc explains how to run phc and how to use phc to convert from PHP to XML and back.

Porting and packaging phc

Porting and packaging phc discusses issues in how to maintain, port, and package phc.

Reference

Writing Plugins

Getting Started

Getting Started introduces writing plugins for phc. It then explains how phc represents PHP scripts internally, and shows how to write a simple plugin that counts the number of classes in a PHP script.

Tutorial 1: Traversing the Tree

Tutorial 1 introduces the support that phc offers for traversing (and transforming) scripts. It shows how to write a program that counts the number of function calls in a script.

Tutorial 2: Modifying Tree Nodes

Tutorial 2 shows how you can modify nodes in the tree (without modifying the structure of the tree). It shows how to replace calls to

The Grammar

phc represents PHP scripts internally as an abstract syntax tree. The structure of this tree is dictated by the (abstract) grammar. The grammar definition is a very important part of phc.

Representing PHP

phc's view on the world (as dictated by the grammar) does not completely agree with the PHP standard view. Representing PHP describes how the various PHP constructs get translated into the abstract syntax.

Overview of the AST classes and transformation API

This gives an overview of the AST classes, the tree visitor API and the tree transformation API from a programmer's perspective.

maketea Theory

maketea is a tool bundled with phc which, based on a grammar definition of a language, generates a C++ hierarchy for the corresponding abstract syntax tree, a tree transformation and visitor API, and deep cloning, deep equality and pattern matching on the AST. The maketea theory explains some of the theory behind maketea; in particular, the grammar formalism, the mapping from the grammar to the AST classes, and the derivation of the tree transformation API.

Miscellaneous

Memory Layout for Multiple and Virtual Inheritance

Programmers wishing to enhance their knowledge of C++ might find Memory Layout for Multiple and Virtual Inheritance an interesting article. It explains the difficulties C++ compilers face when deciding the (runtime) layout for objects in the presence of multiple inheritance, and the consequences this has for programmers.

Writing a Reentrant Parser with Flex and Bison

`mysql_connect` by calls to `dbx_connect`.

Tutorial 3: Restructuring the Tree

Tutorial 3 shows how you can modify the structure of the tree. It works through an example that removes unnecessary string concatenations (for example, `$a . ""` is replaced by just `$a`).

Tutorial 4: Using State

Tutorial 4 explains an advanced features of pattern matching, and shows an important technique: the use of state in transformations (where one transformation depends on a previous transformation). It shows how to write a program that renames all functions `foo` in a script to `db_foo`, if there are calls to a database engine within `foo`.

Tutorial 5: Modifying the Traversal Order

Tutorial 5 explains how to change the order in which the children of a node are visited, avoid visiting some children, or how to execute a piece of code in between visiting two children.

Tutorial 6: Returning Lists

Tutorial 6 shows how to define transformations that replace nodes in the tree by multiple other nodes, and how to delete nodes from the tree. It also shows to call the phc parser and unparsers from plugins.

Writing a Reentrant Parser with Flex and Bison explains by means of an example how to create a reentrant parser with Flex and Bison, and how to use more than one parser in one application.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

System Requirements

These instructions have been updated for version 0.1.7.
phc needs a Unix-like environment to run (it has been tested on Linux, Solaris, and Mac OS X). To compile phc, you will need

- `gcc` version **3.4.0** or higher (other compilers may work too, but they need to support covariant return types; YMMV)
- `make`

To make full use of phc, you will also need

- Xerces-C++ if you want support for XML parsing (you don't need Xerces for XML unparsing).
- a DOT viewer such as graphviz if you want to be able to view the graphical output generated by phc (for example, syntax trees)

Finally, if you want to modify the internals of phc (in other ways than through the explicit API we provide for doing so), you will need the following tools:

- `flex` (if you need to modify the lexical analyser)
- `bison` (if you need to modify the syntax analyser)
- `patch` (if you modify either the lexical or the syntax analyser)
- ghc (to compile maketea, which is written in Haskell; you will need to compile maketea either if you want to modify the phc grammar (not an easy thing to do), or modify maketea itself)
- gengetopt (if you need to add additional command line arguments; you will need version 0.16 or higher)
- gperf (if you need to modify the list of keywords recognized by the lexical analyser)

However, most people should not need these tools (even if you are implementing tools based on phc).

It is possible to link phc to the Boehm garbage collector (known as `libgc` in many Linux distributions), but this feature is experimental and seems to lead to runtime errors on some systems. Use with care.

Installation Instructions

First of all, you must download the latest release of phc, and save it to some temporary location, for example `/tmp`. If `VERSION` is the version number of the copy of phc you have downloaded, the file will be called `phc-VERSION.tar.gz`. Thus, you should now have a file `/tmp/phc-VERSION.tar.gz` (for example, `/tmp/phc-0.1.6.tar.gz`).

Next you must decide where you want to extract phc. Here, we will assume that you want to extract it to your home directory (~). Extract phc as follows.

```
cd ~
tar xvfz /tmp/phc-VERSION.tar.gz
```

This will create a new directory `~/phc-VERSION` that contains the phc source tree. Finally, you must compile phc. You should be able to simply type

```
cd ~/phc-VERSION
./configure
make
```

This should compile without any warnings or errors. If this step fails, please send a bug report to the mailing list with as much information about your system as you can give, and we will try to resolve it. Finally, install phc using

```
make install
```

For information on running phc, see Running phc. If you can follow those instructions and you get the output you should get, congratulations! You have successfully installed phc :-)

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Running phc

Once you have installed phc, run it by typing

```
./phc
```

You should see something like

```
phc 0.1.6

Usage: phc [OPTIONS]... [FILES]...

  -h, --help                  Print help and exit
      --full-help             Print help, including hidden options, and exit
  -V, --version               Print version and exit
      --dump-php              Dump PHP code back immediately after parsing to
                                standard output (pretty printing)
                                (default=off)
      --dump-ast-dot          Dump the AST from the source in dot format
                                (default=off)
      --dump-ast-xml          Dump the AST from the source in XML format
                                (default=off)
      --compile-time-includes When possible, replace include() statements with
                                the parsed contents of the specified file
                                (default=off)
      --tab=STRING            String to use for tabs while unparsing
                                (default=`     ')
```

(the exact output will depend on the version of phc)

Now write a very small PHP script, for example

```
<? echo "Hello world!"; ?>
```

and store it in a file, for example in `helloworld.php`. Then run phc as follows:

```
./phc --dump-php helloworld.php
```

This should output a pretty-printed version of your PHP script back to standard output:

```
<?php
   echo "Hello world!";
?>
```

Graphical Output

phc represents PHP scripts internally as trees (this is further explained in Tutorial 1). If you have a DOT viewer installed on your system (for example, graphviz), you can view this tree graphically. First, ask phc to output the tree in DOT format:

```
./phc --dump-ast-dot helloworld.php > helloworld.dot
```

You can then view the tree (`helloworld.dot`) using Graphviz. In most Unix/Linux systems, you should be able to do

```
dotty helloworld.dot
```

And you should see the tree (it should look similar to the one we generated).

Writing and Reading XML

phc can also output an XML representation of the tree. You can use this representation if you want to process PHP scripts without using the phc framework, but yet using the phc abstract representation of PHP scripts. To generate an XML version of the tree, run

```
./phc --dump-ast-xml helloworld.php > helloworld.xml
```

phc can also read the XML back in, after which all the usual features of phc are again available; in particular, it is possible to read an XML file, and write PHP syntax. To convert the XML file we just generated back to PHP syntax, run

```
./phc --read-ast-xml --dump-php helloworld.xml
```

The generated XML uses the schema http://www.phpcompiler.org/phc-1.0.

Compile-time Includes

phc now has initial support for compile-time processing of PHP's `include` built-in. Enabling this feature inserts the included statements in the AST in the place of the `include` statement. Included functions become part of the %MAIN% class, and included classes become part of the script. In the event that phc is not able to process the `include` statement (for example, if it would require using the `remote` feature of `include`), a warning is issued, and the `include` statement is left in place. To enable this support, run

```
./phc --compile-time-includes script_with_includes.php
```

The include support is intended to mimic PHP's include built-in, as far as can be achieved at compile time. phc supports:

- Moving included statements to the point at which `include` was called. Naturally, these statement's use the variable scope at the point at which they are included,
- Preserving `__FILE__` and `__LINE__` statements,
- Moving included functions to the `%MAIN%` class, and importing the included classes,
- `include`, and `require`. If the specified file cannot be found, parsed, or if the argument to `include` is not a string literal, the include statement is left in place.

phc does not support:

- Return values in included scripts. We intend to support these in the future. They will likely be supported in a later stage of the compilation process, instead of in the AST,
- Calling `include` on anything other than a literal string containing the filename of a local file. This excludes variables and remote files. These may be supported when more static analyses are available,

- `include_once` and `require_once`, as we cannot guarantee that the file to be included is not included elsewhere. These statements will not be processed, and combinations of `include` or `require` and `include_once` or `require_once` may cause incorrect behaviour with this option set,
- Updating `get_included_files()` to reflect the included files.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

We need porters, packagers and maintainers

Now that phc has a plugin architecture, it is no longer necessary for users to integrate their source with it. As a result, it is much more useful to package phc and integrate it within various distributions' package management systems. If you are interested in packaging phc for your favourite OS, please contact us.

Currently, phc runs on x86 Linux. Most of the testing occurs on i686 Ubuntu Dapper, with occasional testing on i686 Debian Etch, and infrequent testing on sparc Solaris (SunOS 5.9). If you have access to other machines, architectures and operating systems, and would be willing to test phc on it, please contact us.

Packaging hints

Do not strip the binaries. Since the plugins use `dlopen()`, and link dynamically against the `phc`, the plugins will not work unless the symbol information is available.

Test suite

Although we included tests in phc up to version 0.1.5.1-1, it is no longer practical to do so, mostly due to the size of the regression tests. However, in order to help potential porters and maintainers, we provide the full test suite (20MB). Extract the tests in the `src/` directory, and run

```
make test
```

to run the short version of the tests, or

```
make long-test
```

to run the entire suite. Note that the line numbers test is expected to fail (this is a known bug and will be fixed in a later release).

phc packages

Dries Verachtert has RPMs for phc in DAG RPM repositories. Versions 0.1.3 to 0.1.5.1-1 are packaged for x86_64 and i386, for Red Hat Linux 7 and 9, Red Hat Enterprise Linux 2, 3 and 4, and Fedora Core 1, 2, 3, and 4. Source RPMs are also available.

Conor McDermottroe has created a new port for the FreeBSD ports collection, which should be available soon.

We're looking for people to create and/or maintain packages for more systems, including Debian/Ubuntu (especially Debian/Ubuntu), Gentoo, Slackware, Darwin and Solaris.

$LastChangedDate: 2006-05-30 09:48:43 +0100 (Tue, 30 May 2006) $. Contents © the authors.

Home | Downloads | Documentation | Plugins | Spinoff Projects | Mailing List

Demo

This demo is intended as a quick introduction outlining what the current release of phc can do for you. It does not explain everything in detail. For more information on implementing your own tools based on phc, read Getting Started.

The Source Program

Consider the following simple PHP script.

```
<?php
   function foo()
   {
      return 5;
   }

   $foo = foo();
   echo "foo is $foo<br>";
?>
```

Internally this program gets represented as an abstract syntax tree (open demo.png to see the tree).

The Transform

Suppose we want to rename function `foo` to `bar`. This is done by the following (C++) program:

```
#include "Tree_visitor.h"

class Rename_foo_to_bar : public Tree_visitor
{
   void pre_method_name(Token_method_name* in)
   {
      if(*in->get_value == "foo")
         in->value = new String("bar");
   }
};
```

Finally, we need to instruct phc to run our transform:

```
Rename_foo_to_bar f2b;
php_script->transform(&f2b);
```

The Result

Running phc gives

```
<?php
   function bar()
   {
```

```
        return 5;
    }

    $foo = bar();
    echo "foo is " . $foo . "<br>";
?>
```

where the name of the function has been changed, while the name of the variable remained unaltered, as has the text `"foo"` inside the string. It's that simple! Of course, in this example, it would have been quicker to do it by hand, but that's not the point; the example shows how easy it is to operate on PHP scripts within the phc framework.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Getting Started

For this introductory tutorial, we assume that you have successfully downloaded and installed phc, and that you know how to run it (see the Installation Instructions and Running phc). This tutorial gets you started with using phc to develop your own tools for PHP by writing plugins.

Compiling a Plugin

To get up and running, we'll first write a “hello world” plugin that does nothing except print a string. Create a new directory, say `~/myplugins` and create a new file `helloworld.cpp`:

```
#include <phc/ast.h>
#include <iostream>

using namespace std;

extern "C" void process_ast(AST_php_script* php_script)
{
   cout << "Hello world (I'm a phc plugin!)" << endl;
}
```

This is an example of a minimal plugin. Every plugin you write must contain a `process_ast` method with this exact signature. To compile the plugin, run

```
~/myplugins$ phc_compile_plugin helloworld.cpp -o helloworld.so
```

(`phc_compile_plugin` is a small shellscript that makes the task of compiling plugins easier; all it does is call `g++` with a couple of options; if you're curious, you can open it in any text editor.) Finally, run the plugin using

```
~/myplugins$ phc --run helloworld.so sometest.php
```

(You need to pass in an input script to `phc` even though our plugin does not use it.) If that worked as expected, congratulations: you've just written your first phc plugin! :-)

About `extern "C"`

You may have been wondering what the `extern "C"` in the definition of `process_ast` is for; the reason is that phc uses the Unix `dlopen` interface to load your plugin; if you do not declare `process_ast` as `extern "C"`, phc will not be able to find the `process_ast` symbol in your plugin because the name of that function will have been mangled by the C++ compiler. Incidentally, this does not mean that you cannot write C++ code inside `process_ast`.

If you don't understand any of that, don't worry about it: just remember that you need to declare `process_ast` as `extern "C"` and everything will be fine. (You don't need `extern "C"` for any other functions you might define.)

The Abstract Syntax

To be able to do anything useful in your plugins, you need to know how phc represents PHP code internally. phc's view of PHP scripts is described by an *abstract grammar*. An abstract grammar describes how the contents of a PHP script are structured. A grammar consists of a number of rules. For example, there is a rule in the grammar that describes how `if` statements work:

```
if ::= expr iftrue:statement* iffalse:statement* ;
```

This rules reads: "*An `if` statement consists of an expression* (the condition of the if-statement), *a list of statements called `iftrue'* (the instructions that get executed when the condition holds), *and another list of statements called `iffalse'* (the instructions that get executed when the condition does not hold)". The asterisk (`*`) in the rule means "list of".

As a second example, consider the rule that describes arrays in PHP. This rule should cover things such as `array()`, `array("a", "b")` and `array(1 => "a", 2 => "g")`. Arrays are described by the following two rules.

```
array ::= array_elem* ;
array_elem ::= key:expr? val:expr ;
```

(Actually, this is a simplification, but it will do for the moment.) These two rules say that "*an array consists of a list of array elements*", and an "*array element has an optional expression called `key', and a second expression called `val'*". The question mark (`?`) means "optional". Note that the grammar does not record the need for the keyword `array`, or for the parentheses and commas. We do not need to record these, because we already *know* that we are talking about an array; all we need to know is what the array elements are.

The Abstract Syntax Tree

When phc reads a PHP script, it builds up an internal representation of the script. This representation is known as an *abstract syntax tree* (or AST for short). The structure of the AST follows directly from the abstract grammar. For people familiar with XML, this tree can be compared to the DOM representation of an XML script (and in fact, phc can output the AST as an XML document).

For example, consider `if`-statements again. An `if`-statement is represented by an instance of the `AST_if` class, which is (approximately) defined as follows.

```
class AST_if
{
public:
   AST_expr* expr;
   AST_statement_list* iftrue;
   AST_statement_list* iffalse;
};
```

Thus, the name of the rule (`if ::= ...`) translates into a class `AST_if`, and the elements on the right hand side of the rule (`expr iftrue:statement* iffalse:statement*`) correspond directly to the class members. The class `AST_statement_list` inherits from the STL `list` class, and can thus be treated as such.

Similarly, the class definitions for arrays and array elements look like

```
class AST_array
{
public:
```

```
   AST_array_elem_list* array_elems;
};

class AST_array_elem
{
public:
   AST_expr* key;
   AST_expr* val;
};
```

When you start developing applications with phc you will find it useful to consult the full description of the grammar, which can be found in the Grammar Definition. A detailed explanation of the structure of this grammar, and how it converts to the C++ class structure, can be found in the Grammar Formalism. Some notes on how phc converts normal PHP code into abstract syntax can be found in Representing PHP.

Working with the AST

When you want to build tools based on phc, you do not have to understand how the abstract syntax tree is built, because this is done for you. Once the tree has been built, you can examine or modify the tree in any way you want. When you are finished, you can ask phc to output the tree to normal PHP code again.

Let's write a very simple plugin that counts the number of class definitions in a script. If you look at the grammar, you will notice that class definitions are represented by a (C++) class called `AST_class_def`. So, we need to count the number of objects of type `AST_class_def` in the tree. Create a new file `~/myplugins/count_classes.cpp`. Recall the skeleton plugin:

```
#include <phc/ast.h>

extern "C" void process_ast(AST_php_script* php_script)
{
}
```

You will notice that `process_ast` gets passed an object of type `AST_php_script`. This is the top-level node of the generated AST. If you look at the grammar, you will find that `AST_php_script` corresponds to the following rule:

```
php_script ::= interface_def* class_def+ ;
```

Thus, as far as phc is concerned, a PHP script consists of a number of interface definitions, followed by a number of class definitions (see Representing PHP). The plus (+) in this rule is similar to an asterisk (`*`), but indicates that there must at least be one item in the list. In other words, a PHP script may not have any interface definitions, but it must have at least one class definition.

By now you should be able to deduce that the class `AST_php_script` will have two members, called `interface_defs` and `class_defs`, both of which are lists. So, to count the number of classes, all we have to do is query the number of elements in the `class_defs` vector:

```
#include <phc/ast.h>

extern "C" void process_ast(AST_php_script* php_script)
{
   printf("%d class definition(s) found\n", php_script->class_defs->size());
}
```

Save this file to `~/myplugins/count_classes.cpp`. Compile:

```
~/myplugins$ phc_compile_plugin -o count_classes.so count_classes.cpp
```

And run:

```
./phc --run count_classes.so hello.php
```

Actually...

If you actually did try to run your plugin, you might think right now that something went wrong: phc appears to report one class definition too many! However, there is a very good reason for this. We said earlier that as far as phc is concerned, a PHP script consists of a number of interface definitions, followed by at least one class definition. So where does the code that is defined outside of any class go?

The answer is that any code defined outside any class goes into a special class called `%MAIN%`. Any functions you define that do not belong to any class, become members of `%MAIN%`, and any code you write that does not belong to any function, becomes part of a special method in `%MAIN%` called `%run%`. When phc outputs the tree back to normal PHP code, `%MAIN%` disappears; however, when you work with the tree, there is no distinction between code defined inside and outside classes; in the tree, everything is defined as part of some class. This makes the tree simpler and easier to work with.

More details about how the various PHP constructs are represented in the abstract grammar can be found in Representing PHP.

Writing Stand Alone Applications

If you prefer not to write a plugin but want to modify phc itself to derive a new, stand-alone, application, you can modify `process_ast` in `process_ast/process_ast.cpp` in the phc source tree instead. This has the effect of “hardcoding” your plugin into phc (in versions before 0.1.7, this was the only way to write extensions). However, in the rest of the tutorials we will assume that you are writing your extension as a plugin.

What's Next?

In theory, you now know enough to start implementing your own tools for PHP. Write a new plugin, run the plugin using the `--run` option, and optionally pass in the `--dump-php` option also to request that phc outputs the tree back to PHP syntax after having executed your plugin.

However, you will probably find that modifying the tree, despite being well-defined and easy to understand, is actually rather laborious. It requires a lot of boring boilerplate code. The good news is that phc provides sophisticated support for examining and modifying this tree. This is explained in detail in the follow-up tutorials:

- Tutorial 1: Traversing the Tree
- Tutorial 2: Modifying Tree Nodes
- Tutorial 3: Restructuring the Tree
- Tutorial 4: Using State
- Tutorial 5: Modifying the Traversal Order
- Tutorial 6: Returning Lists

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Tutorial 1: Traversing the Tree

In Getting Started, we explained that phc represents PHP scripts internally as an abstract syntax tree, and that the structure of this tree is determined by the grammar. We then showed how to make use of this tree to count the number of classes. In this tutorial, we will consider an equally simple task: we want to count the number of function calls in a script. So, for the following PHP script,

```
<?php
   echo "Hello ";
   echo "world!";
?>
```

we should report two function calls.

Note that all the plugins that we will develop in these tutorials are included in the phc distribution. For example, in this tutorial we will be developing two plugins: a difficult solution to the problem and an easy solution to the problem. You can run these plugins by running

```
phc --run plugins/tutorials/count_function_calls_difficult.so hello.php
```

or

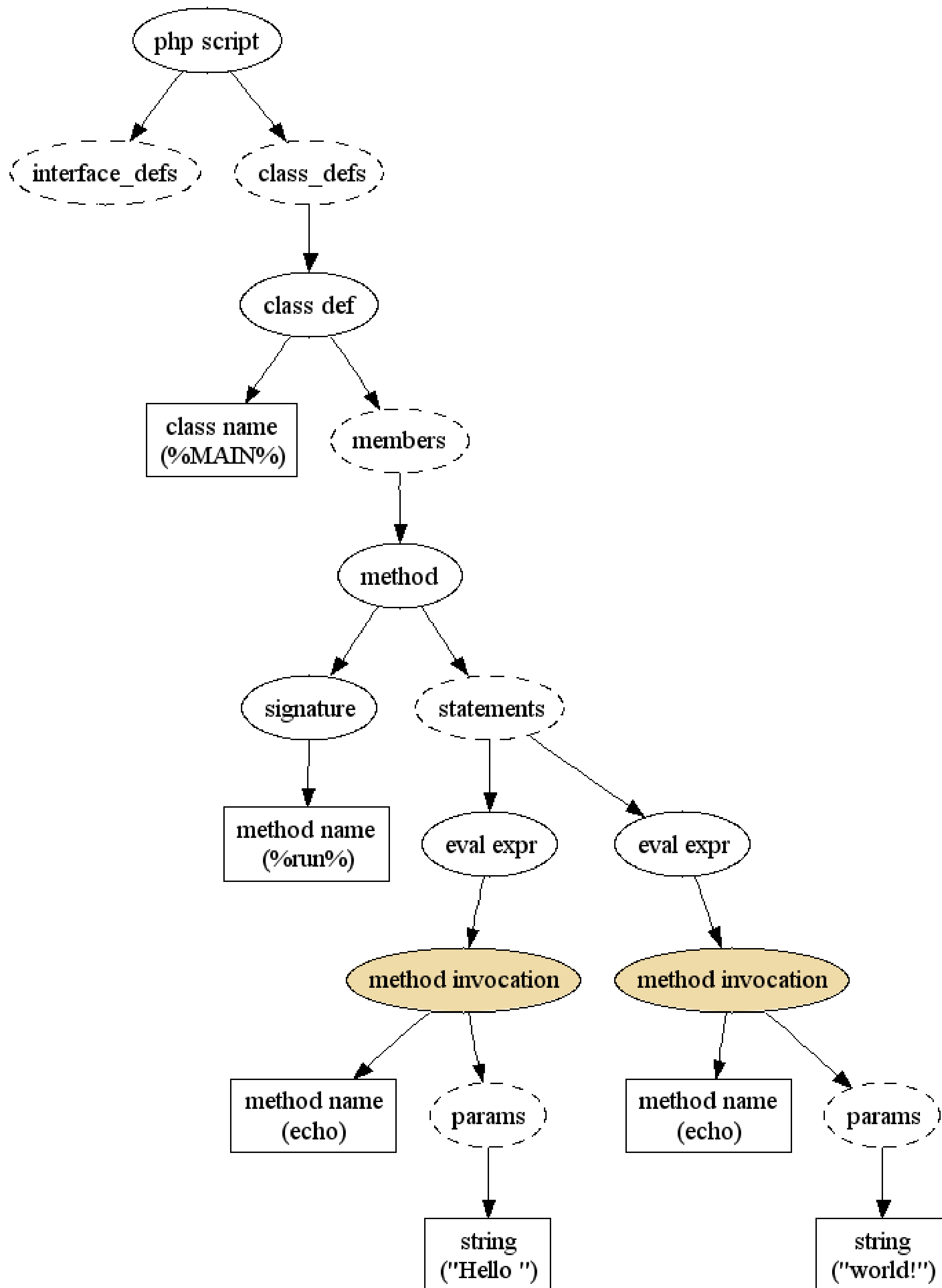
```
phc --run plugins/tutorials/count_function_calls_easy.so hello.php
```

The Grammar (Revisited)

How do we go about counting the number of function calls in a script? Remember that, as far as phc is concerned, a PHP script consists of a number of classes (and interface definitions). Each of these classes may have one or more methods, and each method can have one or more statements in them. Simplified, the grammar would state this as:

```
php_script ::= class_def+ ;
class_def ::= CLASS_NAME member* ;
member ::= method | attribute ;
method ::= signature statement* ;
signature ::= METHOD_NAME formal_parameter* ;
```

where the vertical bar (|) means “or”. Thus, our running example is represented by the following tree.



(Note that this tree is simplified from the real tree; not all nodes are shown. You can also view the full tree).

Statements and Expressions

The two nodes that we are interested in are the "method invocation" nodes. The `eval expr` nodes just above them probably need some explanation. There are many different types of statements in PHP: `if`-statements, `while`-statements, `for`-loops, etc. You can find the full list in the grammar. If you do look at the grammar, you will notice in particular that a function call is not actually a statement! Instead, a function call is an *expression*.

The difference between statements and expressions is that a statement *does* something (for example, a `for`-loop repeats a bunch of other statements), but an expression has a *value*. For example, “5” is an expression (with value 5), “1+1” is an expression (with value 2), etc. A function call is also considered an expression. The value of a function call is the value that the function returns.

Now, the node `eval expr` makes a statement from an expression. So, if you want to use an expression where phc expects a statement, you have to use the grammar rule

```
statement ::= ... | eval_expr ;
eval_expr ::= expr ;
```

The Difficult Solution

The following plugin counts the number of function calls in a tree. If you do not understand the code, do not worry! We will look at a much easier solution in a second. If you understand the comments, that is enough.

```
#include <phc/ast.h>

extern "C" void process_ast(AST_php_script* php_script)
{
   AST_class_def_list::const_iterator ci;
   AST_member_list::const_iterator mi;
   AST_statement_list::const_iterator si;

   AST_method* method;
   AST_eval_expr* eval_expr;
   AST_method_invocation* invoc;

   int num_function_calls = 0;

   // Inspect all classes
   for(
      ci = php_script->class_defs->begin();
      ci != php_script->class_defs->end();
      ci++)
   {
      // Inspect all members in the class
      for(
         mi = (*ci)->members->begin();
         mi != (*ci)->members->end();
         mi++)
      {
         // Check whether this member is a method or an attribute
         method = dynamic_cast<AST_method*>(*mi);
         if(method == NULL) continue;

         // Check all statements in the method
         for(
            si = method->statements->begin();
            si != method->statements->end();
            si++)
         {
            // Check if the statement is of type "eval_expr"
            eval_expr = dynamic_cast<AST_eval_expr*>(*si);
            if(eval_expr == NULL) continue;

            // Finally, check if the expression is a function call
            invoc = dynamic_cast<AST_method_invocation*>(eval_expr->expr);
            if(invoc == NULL) continue;

            // Yeah! We found a function call
```

```
            num_function_calls++;
         }
      }
   }

   printf("%d function calls found\n", num_function_calls);
}
```

Why is this code so complicated? First of all, it has to search through the entire tree, looking for function calls. Because function calls are fairly deep down in the tree, we need a lot of code simply to find them. The second complication is the fact that, for example, a class consists of “members“. A member is either an attribute or a method, but we are interested only in methods. Similarly, a method consists of “statements“. A statement can be one of many things; but we are only interested in `eval_expr` statements. Thus, we have to test nodes for their type (using `dynamic_cast`).

The Easy Solution

Fortunately, phc will do all this for you automatically! There is a standard “do-nothing” tree traversal predefined in phc in the form of a class called `Tree_visitor` (defined in `<phc/Tree_visitor.h>`). `Tree_visitor` contains methods for each type of node in the tree. phc will automatically traverse the abstract syntax tree for you, and call the appropriate method at each node.

In fact, there are *two* methods defined for each type of node. The first method, called `pre_something`, gets called on a node *before* phc visits the children of the node. The second method, called `post_something`, gets called on a node *after* phc has visited the children of the node. For example, `pre_method` gets called on an `AST_method`, before visiting the statements in the method. After all statements have been visited, `post_method` gets called. Thus, the very first method that gets called is `pre_php_script` (because that is the top-level node in the tree), and the very last method that gets called is `post_php_script`.

So, here is an alternative and much easier solution for our problem.

```
#include <phc/Tree_visitor.h>

class Count_function_calls : public Tree_visitor
{
private:
   int num_function_calls;

public:
   // Set num_function_calls to zero before we begin
   void pre_php_script(AST_php_script* in)
   {
      num_function_calls = 0;
   }

   // Print the number of function calls when we are done
   void post_php_script(AST_php_script* in)
   {
      printf("%d function calls found\n", num_function_calls);
   }

   // Count the number of function calls
   void post_method_invocation(AST_method_invocation* in)
   {
      num_function_calls++;
   }
};
```

```
extern "C" void process_ast(AST_php_script* php_script)
{
   Count_function_calls cfc;
   php_script->visit(&cfc);
}
```

The real work in this transform is now done by the visitor; the only task left that `process_ast` still has to do is instanstiate the visitor and run it over the tree.

Counting All Statements

(The plugin explained in this section is available as `plugins/tutorials/count_statements.so` in the phc distribution.)

Suppose we wanted to count all statements, rather than just function calls. We could define methods for `eval_expr`, `for`, `while`, `if`, etc., but there is an easier way. If you check the grammar, you will find the following rules:

```
statement ::= if | ... ;
commented_node ::= ... | statement | ... ;
node ::= ... | node | ... ;
```

You could read this as "an if-statement *is-a* statement, a statement *is-a* commented node (a node that may have comments associated with it), and a commented node *is-a* node. This *is-a* relation is reflection using inheritance (class `AST_if` inherits from `AST_statement`), but the tree visitor API also has corresponding "generic" methods. For example, the following suffices to count all statements:

```
void pre_statement(AST_statement* in)
{
   num_statements++;
}
```

We need to be precise about the order in which phc calls all these methods. Suppose we have a node `Foo` (say, an if-statement), which *is-a* `Bar` (say, statement), which itself *is-a* `Baz` (say, commented node). Then phc calls the visitor methods in the following order:

1. `pre_baz`
2. `pre_bar`
3. `pre_foo`
4. `children_foo` (visit the children of `foo`)
5. `post_foo`
6. `post_bar`
7. `post_baz`

Just to emphasise, if all of the visitor methods listed above are implemented, they will *all* be invoked, in the order listed above. So, implementing a more specific visitor (`pre_foo`) does not inhibit the more general method (`pre_bar`) from being invoked. You can run the `plugins/tutorials/show_traversal_order.so` from the phc distribution to see this in action.

What's Next?

To find out how you can modify the tree, continue with Tutorial 2.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Tutorial 2: Modifying Tree Nodes

Now that we have seen in Tutorial 1 how to inspect the tree, in this tutorial we will look at modifying the tree. The task we set ourselves is: replace all calls to `mysql_connect` by calls to `dbx_connect` (dbx is a PECL extension to PHP that allows scripts interface with a database independent of the type of the database; this conversion could be part of a larger refactoring process that makes a script written for MySQL work with other databases.)

The tutorial we develop in this tutorial is available as `MySQL2DBX.so` in the phc distribution. To see its effect, run phc as follows:

```
phc --run plugins/tutorials/MySQL2DBX.so --dump-php test.php
```

First Attempt

As in the previous tutorial, we are interested in all function calls. However, now we are interested only in function calls to `mysql_connect`. Let us have a look at the precise definition of a function call according to the grammar:

```
method_invocation ::= target method_name params:actual_parameter* ;
method_name ::= METHOD_NAME | reflection ;
actual_parameter ::= is_ref:"&"? expr ;
```

(The `target` of a method invocation is the class or object the function gets invoked on. It is explained in Tutorial 3, and need not worry us here.) For now, we are only interested in the `method_name`. The grammar tells us that a `method_name` is either a `METHOD_NAME` or an `expr`. If a symbol is written in CAPITALS in the grammar, that means it refers to a “literal”. In this case, to an actual method name (such as `mysql_connect`). In PHP, it is also possible to call a method whose name is stored in variable; in this case, the function name will be a `reflection` node (which contains an `expr`). In this tutorial, we are interested in “normal” method invocations only.

Literals (or “*terminal symbols*”) do not get represented by a class called `AST_something`, but by a class called `Token_something` instead. All classes `Token_something` have an attribute called `value` which corresponds to the value of the token. For most tokens, the type of `value` is an STL `String*`. However, for some tokens, for example `INT`, `value` has a different type (e.g., `int`). If the token has a non-standard type, it will have an additional attribute called `source_rep`, which corresponds to the representation of the token in the source. For example, the real number `5E-1` would have `value` equal to the (`double`) 0.5, but `source_rep` equal to (the `String*`) “5E-1”.

Thus, we arrive at the following first attempt.

```
#include <phc/Tree_visitor.h>

class MySQL2DBX : public Tree_visitor
{
public:
   void post_method_invocation(AST_method_invocation* in)
```

```
    {
        Token_method_name* name;

        name = new Token_method_name(new String("mysql_connect"));
        if(in->method_name->match(name))
        {
            in->method_name = new Token_method_name(new String("dbx_connect"));
        }
    }
};

extern "C" void process_ast(AST_php_script* php_script)
{
    MySQL2DBX m2d;
    php_script->visit(&m2d);
}
```

Note: phc uses a garbage collector, so there is never any need to free objects (you never have to call `delete`). This makes programming much easier and less error-prone (smaller chance of bugs).

`match` compares two (sub)trees for deep equality. There is also another function called `deep_equals`, which does nearly the same thing, but there are two important differences. `match` does not take comments, line numbers and other "additional" information into account, whereas `deep_equals` does. The second difference is that `match` supports wildcards; this will be explained in Tutorial 3.

Modifying the Parameters

Unfortunately, renaming `mysql_connect` to `dbx_connect` is not sufficient, because the parameters to the two functions differ. According to the PHP manual, the signatures for both functions are

```
mysql_connect (server, username, password, new_link, int client_flags)
```

and

```
dbx_connect (module, host, database, username, password, persistent)
```

The `module` parameter to `dbx_connect` should be set to `DBX_MYSQL` to connect to a MySQL database. Then `host` corresponds to `server`, and `username` and `password` have the same purpose too. So, we should insert `DBX_MYSQL` at the front of the list, and insert `NULL` in between `host` and `username` (the `mysql_connect` command does not select a database). The last two parameters to `mysql_connect` do not have an equivalent in `dbx_connect`, so if they are specified, we cannot perform the conversion. The last parameter to `dbx_connect` (`persistent`) is optional, and we will ignore it in this tutorial.

Now, in phc, `DBX_MYSQL` is an `AST_constant`. Because phc deals with everything as being classes, a constant must also be defined in a class. Constants defined in the PHP standard library, such as `DBX_MYSQL`, should be defined in the special "class" `%STDLIB%`. (For more information on how PHP gets converted to the abstract syntax tree, see Representing PHP.)

Finally, `NULL` is represented by `Token_null`.

We are now ready to write our conversion function:

```
#include <phc/Tree_visitor.h>
```

```
class MySQL2DBX : public Tree_visitor
{
public:
   void post_method_invocation(AST_method_invocation* in)
   {
      AST_actual_parameter_list::iterator pos;
      Token_constant_name* module_name;
      AST_constant* module_constant;
      AST_actual_parameter* param;
      Token_method_name* name;

      name = new Token_method_name(new String("mysql_connect"));
      if(in->method_name->match(name))
      {
         // Check for too many parameters
         if(in->actual_parameters->size() > 3)
         {
            printf("Error: unable to translate call "
               "to mysql_connect on line %ld\n", in->get_line_number());
            return;
         }

         // Modify name
         in->method_name = new Token_method_name(new String("dbx_connect"));

         // Modify parameters
         module_name = new Token_constant_name(new String("DBX_MYSQL"));
         module_constant = new AST_constant("%STDLIB%", module_name);

         pos = in->actual_parameters->begin();
         param = new AST_actual_parameter(false, module_constant);
         in->actual_parameters->insert(pos, param); pos++;
         /* Skip host */ pos++;
         Token_null* null = new Token_null(new String("NULL"));
         param = new AST_actual_parameter(false, null);
         in->actual_parameters->insert(pos, param);
      }
   }
};

extern "C" void process_ast(AST_php_script* php_script)
{
   MySQL2DBX m2d;
   php_script->visit(&m2d);
}
```

If we apply this transformation to

```
$link = mysql_connect('host', 'user', 'pass');
```

We get

```
$link = dbx_connect(DBX_MYSQL, "host", NULL, "user", "pass");
```

Refactoring

A quick note on refactoring. Refactoring is the process of modifying existing programs (PHP scripts), usually to work in new projects or in different setups (for example, with a different database engine). Manual refactoring is laborious and error-prone, so tool-support is a must. Although phc can be used to refactor PHP code as shown in this tutorial, a dedicated refactoring tool for PHP would be easier to use (though of course less flexible). Such a tool can however be built on top of phc. See also the list of Spinoff Projects.

What's Next?

Tutorial 3 explains how you can modify the *structure* of the tree, as well as the tree nodes.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Tutorial 3: Restructuring the Tree

Now that we have seen in Tutorial 1 how we can traverse the tree, and in Tutorial 2 how we can modify individual nodes in the tree, in this tutorial we will look at modifying the structure of the tree itself.

The transform that we will be considering in this tutorial is one that is used in phc itself. The transform is called `Remove_concat_null` and can be found in `process_ast/Remove_concat_null.h`. The purpose of the transform is to remove string concatenation with the empty string. For example,

```
<?php
   $s = "foo" . "";
?>
```

is translated to

```
<?php
   $s = "foo";
?>
```

The reason that this transform is implemented in phc is due to how the phc parser deals with in-string syntax. For example, if you write

```
$a = "foo $b bar";
```

the corresponding tree generated by phc is

```
$a = "foo " . $b . " bar";
```

In other words, the variables are pulled out of the string, and the various components are then concatenated together. However, taken to its logical conclusion, that means that if you write

```
$a = "foo $b";
```

the parser generates

```
$a = "foo " . $b . "";
```

Obviously, the second concatenation is unnecessary, and the `Remove_concat_null` transform cleans this up. In this tutorial we will explain how this transform can be written.

Introducing the `Tree_transform` API

Concatenation is a binary operator, so we are interested in nodes of type `AST_bin_op`. If you check the grammar or, alternatively, `ast.h`, you will find that `AST_bin_op` has three attributes: a `left` and a `right` expression (of type `AST_expr`) and the operator itself (`Token_op* op`). Thus, we

are interested in nodes of type AST_bin_op whose op equals the single dot (for string concatenation).

Based on the previous two tutorials, we might try something like this:

```
class Remove_concat_null : public Tree_visitor
{
public:
   void pre_bin_op(AST_bin_op* in)
   {
      // Find concat operators
      if(*in->op->value == ".")
      {
         // ...
      }
   }
}
```

The problem is, what are we going to do inside the if? Tree visitors can only inspect and modify in; they cannot restructure the tree. In particular, we cannot replace *in by a new node. For this purpose, phc offers a separate API, the tree *transformation* API. It looks very similar to the tree visitor API, but there are two important differences. First, the pre and post methods can modify the structure of the tree by returning new nodes. Second, there are no “generic” methods in the tree transform API. So, it is not possible to define a transformation that would replace all statements by something else. (It is not clear how that would be useful, anyway.)

So, we need to write our transformation using the Tree_transform API, defined in generated/Tree_transform.h. Restructuring the class above yields

```
class Remove_concat_null : public Tree_transform
{
public:
   AST_expr* pre_bin_op(AST_bin_op* in)
   {
      // Find concat operators
      if(*in->op->value == ".")
      {
         // ...
      }
   }
}
```

The differences between the previous version have been highlighted. We inherit from a different class, and pre_bin_op now has a return value, which is the node that will replace *in. If you check the default implementation of pre_bin_op in generated/Tree_transform.cpp, you'll find:

```
AST_expr* Tree_transform::pre_bin_op(AST_bin_op* in)
{
   return in;
}
```

The **return** in; is very important; as we mentioned before, the return value of pre_bin_op will replace *in in the tree. Therefore, if we don't want to replace *in, or perhaps if we want to replace *in only if a particular condition holds, we must return in. This will replace *in by in itself.

The second thing to note is that the return type of pre_bin_op is AST_expr instead of AST_bin_op. This means that we can replace a binary operator node by another other expression node. The maketea theory explains exactly how the signatures for the pre and post methods are derived, but in most cases they are what you'd expect. The easiest way to check is to simply look them

up in <phc/Tree_transform.h>.

The Implementation

We wanted to get rid of useless concatenation operators. To be precise, if the binary operator is the concatenation operator, and the left operand is the empty string, we want to replace the node by the right operand; similarly, if the right operand is the empty string, we want to replace the operator by its left operand. Here's the full transform:

```
class Remove_concat_null : public Tree_transform
{
public:
   AST_expr* post_bin_op(AST_bin_op* in)
   {
      Token_string* empty = new Token_string(new String(""), new String(""));

      // Replace with right operand if left operand is the empty string
      if(in->match(new AST_bin_op(empty, WILDCARD, ".")))
         return in->right;

      // Replace with left operand if right operand is the empty string
      if(in->match(new AST_bin_op(WILDCARD, empty, ".")))
         return in->left;

      return in;
   }
}
```

We already explained what `match` does in the previous tutorial, but we have not yet explained the use of wildcards. If you are using a wildcard (`WILDCARD`) in a pattern passed to `match`, `match` will not take that subtree into account. Thus,

```
if(in->match(new AST_bin_op(empty, WILDCARD, ".")))
```

can be paraphrased as “is `in` a binary operator with the empty string as the left operand and `"."` as the operator (I don't care about the right operand)?“

Note that the constructor for `Token_string` has two arugments: one corresponds to the value of the string, and one corresponds to the representation of the string in the source (see also the explanation of the token classes in Tutorial 2). For most strings, both of these values are the same; however, in some cases they are different. For example, `value` might be set to `"/home/joe/myscript.php`, while `source_rep` is set to `__FILE__`.

Running Transformations

Recall from the previous two tutorials that visitors are run with a call to `visit`:

```
extern "C" void process_ast(AST_php_script* php_script)
{
    SomeVisitor visitor;
    php_script->visit(&visitor);
}
```

Likewise, transformations are run with a call to (you guessed it :) `transform`:

```
extern "C" void process_ast(AST_php_script* php_script)
{
    SomeTransform transform;
```

```
    php_script->transform(&transform);
}
```

Note however than when invoked like this, the transform should not replace the top-level node in the AST (the `AST_php_script` node itself).

A Subtlety

If you don't understand this section right now, don't worry about it; you might find it useful to read it again after having gained some experience with the transformation API.

We have implemented the transform as a *post*-transform rather than a *pre-* transform. Why? Suppose we implemented the transform as a pre-transform. Consider the following PHP expression (bracketed explicitly for emphasis:)

```
("" . $a) . ""
```

The first binary operator we encounter is the second one (get phc to print the tree if you don't see why.) So, we apply the transform and replace the operator by its left operand, which happens to be `("" . $a)`. We then continue *and transform the children of the that node*, because that is how the tree transform API is defined. But the *children* of that node are `""` and `$a`. So, that means that the other binary operator itself will never be processed!

There are two solutions to this problem. The first is the one we used above, and use a post-transform instead of a pre-transform. You should try to reason out why this works, but a rule of thumb is that unless there is a good reason to use a pre-transform, it's safer to use the post-transform, because in the post-transform the children of the node have already been transformed, so that you are looking at the “final” version of the node.

The second solution is to use a pre-transform, but explicitly tell phc to transform the new node in turn. This is the less elegant solution, but sometimes this is the only solution that will work (see for example the `Token_conversion` transform in the phc source tree). To do this, you would replace

```
return in->right;
```

by

```
return in->right->pre_transform(this);
```

What's Next?

The next tutorial in this series, Tutorial 4, introduces a very important notion in transforms: the use of *state*.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Tutorial 4: Using State

This tutorial explains an advanced feature of pattern matching, and shows an important technique in writing tree transforms: the use of state. Suppose we are continuing the refactoring tool that we began in Tutorial 2, and suppose that we have replaced all calls to database specific functions by calls to the generic DBX functions. To finish the refactoring, we want to rename any function `foo` in the script to `foo_DB`, if it makes use of the database — this clearly sets functions that use the database apart, which may make the structure of the script clearer.

So, we want to write a transform that renames all functions `foo` to `foo_DB`, if there is one or more call within that function to any `dbx_`*`something`* function. Here is a simple example:

```
<?php
   function first()
   {
      global $link;
      $error = dbx_error($link);
   }

   function second()
   {
      echo "Do something else";
   }
?>
```

After the transform, we should get

```
<?php
   function first_DB()
   {
      global $link;
      $error = dbx_error($link);
   }

   function second()
   {
      echo "Do something else";
   }
?>
```

The Implementation

Since we have to modify method (function) names, the nodes we are interested in are the nodes of type `AST_method`. However, how do we know when to modify a particular method? Should we search the method body for function calls to `dbx_`*`xxx`*? As we saw in Tutorial 1, manual searching through the tree is cumbersome; there must be a better solution.

The solution is in fact very easy. At the start of each method, we set a variable `uses_dbx` to `false`. When we process the method, we set `uses_dbx` to `true` when we find a function call to a DBX

function. Then at the end of the method, we check `uses_dbx`; if it was set to `true`, we modify the name of the method. This tactic is implement by the following transform (available as `plugins/tutorials/InsertDB.so` in the phc distribution). Note the use of `pre_method` and `post_method` to initialise and check `use_dbx`, respectively. (Because we don't need to modify the structure of the tree in this transform, we use the simpler `Tree_visitor` API instead of the `Tree_transform` API.)

```
class InsertDB : public Tree_visitor
{
private:
   int uses_dbx;

public:
   void pre_method(AST_method* in)
   {
      uses_dbx = false;
   }

   void post_method(AST_method* in)
   {
      if(uses_dbx)
         in->signature->method_name->value->append("_DB");
   }

   void post_method_invocation(AST_method_invocation* in)
   {
      Token_method_name* pattern = new Token_method_name(WILDCARD);

      // Check for dbx_
      if(in->method_name->match(pattern) &&
         pattern->value->find("dbx_") == 0)
      {
         uses_dbx = true;
      }
   }
};
```

In Tutorial 2, we simply wanted to check for a particular function name, and we used `match` to do this:

```
if(in->match(new Token_method_name("mysql_connect")))
```

Here, we need to check for method names that start with `dbx_`. We use the STL method `find` to do this, but we cannot call this directly on `in->method_name` because `in->method_name` has type `AST_method_name` (could either be a `Token_method_name` or a `AST_reflection` node). However, calling `match` on a pattern has the side effect of making the pattern equal to the tree we are matching on by replacing all wildcards with their corresponding value in the tree. So, after calling `match` in the transform, we can call `find` on the pattern (which has the right type) instead of directly on `in->method_name`. (`match` has this side effect only if the match succeeds.)

(Of course, this transform is not complete; renaming methods is not enough, we must also rename the corresponding method invocations. This is left as an exercise for the reader.)

What's Next?

Tutorial 5 explains how to change the order in which the children of a node are visited, avoid visiting some children, or how to execute a piece of code in between visiting two children.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Tutorial 5: Modifying the Traversal Order

As explained in the previous tutorials (in particular, Tutorial 1), when a `Tree_visitor` traverses a tree, it first calls `pre_xxx` for a node of type *xxx*, it then visits all the children of the node, and finally it calls `post_xxx` on the node. For many transforms, this is sufficient — but not for all. Consider the following transform. Suppose we want to add comments to the *true* and *false* branches of an *if*-statement, so that the following example

```
<?php
   if($expr)
   {
      echo "Do something";
   }
   else
   {
      echo "Do something else";
   }
?>
```

is translated to

```
<?php
   if($expr)
   {
      /* TODO: Insert comment */
      echo "Do something";
   }
   else
   {
      /* TODO: Insert comment */
      echo "Do something else";
   }
?>
```

This appears to be a simple transform. One way to do implement it would be to introduce a flag `comment` that is set to `true` when we encounter an `AST_if` (i.e., in `pre_if`). Then in `post_statement` we could check for this flag, and if it is set, we could add the required comment to the statement, and reset the flag to `false`.

However, this will only add a comment to the first statement in the *true* branch (try!). To add a comment to the first statement in the *false* branch too, we should set the flag to `true` in between visiting the children of the *true* branch and visiting the children of the *false* branch. To be able to do this, we need to modify `children_if`, as explained in the next section.

The Solution

For every AST node type *xxx*, the TreeTransform API defines a method called `children_xxx`. This method is responsible for visiting all the children of the node. The default implementation for `AST_if` is:

```
void children_if(AST_if* in)
{
   in->expr->visit(this);
   in->iftrue->visit(this);
   in->iffalse->visit(this);
}
```

(you can find this definition in `generated/Tree_visitor.cpp`). If you want to change the order in which the children of a node are visited, entirely avoid visiting some children, or simply execute a piece of code in between two children, this is the method you will need to modify.

Here is the transform that does what we need (available as `plugins/tutorials/Comment_ifs.so`):

```
class Comment_ifs : public Tree_visitor
{
private:
   bool comment;

public:
   Comment_ifs()
   {
      comment = false;
   }

   void children_if(AST_if* in)
   {
      in->expr->visit(this);
      comment = true;
      in->iftrue->visit(this);
      comment = true;
      in->iffalse->visit(this);
      comment = false;
   }

   void post_statement(AST_statement* in)
   {
      if(comment && in->get_comments()->empty())
         in->get_comments()->push_back(new String("/* TODO: Insert comment */"));

      comment = false;
   }
};
```

What's Next?

Tutorial 6 explains how to deal with transforms that can replace a single node by multiple new nodes, and shows how to call the phc parser and unparser from your plugins.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Tutorial 6: Returning Lists

In this tutorial we will develop step-by-step a transform that expands `include` statements. For example, if `b.php` is

```
<?php
   echo "Hello world";
?>
```

and `a.php` is

```
<?php
   include "b.php";
   echo "Goodbye!";
?>
```

Then running the transform on `a.php` yields

```
<?php
   echo "Hello world\n";
   echo "Goodbye\n";
?>
```

The transform we will develop in this tutorial is only a simple implementation of `includes`; a more full-featured transform is available using the option `--compile-time-includes`. The code for that transform is `process_ast/Process_includes.cpp`. The transform we will develop here is available as `plugins/tutorials/Expand_includes.so`.

Deleting Nodes

Our transform should process `include` statements. In the AST, `includes` are represented as method invocations. Thus, we might start like this:

```
class Expand_includes : public Tree_transform
{
public:
   AST_expr* pre_method_invocation(AST_method_invocation* in)
   {
      // Process includes
   }
};
```

However, this will not get us very far. The return type of `pre_method_invocation` is an `AST_expr`. That means that we can replace the method invocation (the `include` statement) only by another, single, expression. But we want to replace it by the contents of the specified file!

Recall from Tutorial 1 that to turn an expression into a statement, phc inserts an `AST_eval_expr` in the abstract syntax tree. Thus, if we want to process `include` statements, we could also look at all `eval_expr` nodes. Assuming for the moment we can make that work, does it get us any further? As

a matter of fact, it does! If you check `<phc/Tree_transform.h>`, you will see that the signature for `pre_eval_expr` is

```
void pre_eval_expr(AST_eval_expr* in, AST_statement_list* out)
```

This is different from the signatures we have seen so far. For nodes that can be replaced by a number of new nodes, the pre transform and post transform methods will not have a return value in their signature, but have an extra `AST_xxx_list` argument. This list is initialised to be empty before `pre_eval_expr` is invoked, and when `pre_eval_expr` returns, the nodes in this list will replace `*in`. If the list is empty, the node is simply deleted from the tree.

So, we will use the following plugin as our starting point. Executing this plugin deletes all `eval_expr` nodes from the tree (try it!).

```
#include <phc/Tree_transform.h>

class Expand_includes : public Tree_transform
{
public:
   void pre_eval_expr(AST_eval_expr* in, AST_statement_list* out)
   {
   }
};

extern "C" void process_ast(AST_php_script* php_script)
{
   Expand_includes einc;
   php_script->transform(&einc);
}
```

Using the XML unparser

So, we now want to do something more useful than deleting all `eval_expr` nodes from the tree. The first thing we need to be able to do is distinguish `include` statements from other `eval_expr` nodes. We can use pattern matching (see tutorials 3 and 4) to do that - but what should we match against? If you are unsure about the structure of the tree, it can be quite useful to use the XML unparser to find out what the tree looks like. We modify the plugin as follows:

```
#include <phc/Tree_transform.h>
#include <phc/process_ast/XML_unparser.h>

class Expand_includes : public Tree_transform
{
private:
   XML_unparser xml_unparser;

public:
   void pre_eval_expr(AST_eval_expr* in, AST_statement_list* out)
   {
      in->visit(&xml_unparser);
   }
}
```

The XML unparser is implemented using the `Tree_visitor` API, so it can be invoked just like you run any other visitor. There is a similar visitor called `PHP_unparser` (in `<phc/process_ast/PHP_unparser.h>`) that you can use to print (parts of the) AST to PHP syntax.

When you run this transform on `a.php`, it will print two `eval_expr` nodes (shown in XML syntax), one for the `include` and one for the `echo` . We are interested in the first, the `include`: (we have removed the `<attrs />` blocks to improve readability):

```
<AST_eval_expr>
   <AST_method_invocation>
      <Token_class_name>
         <value>%STDLIB%</value>
      </Token_class_name>
      <Token_method_name>
         <value>include</value>
      </Token_method_name>
      <AST_actual_parameter_list>
         <AST_actual_parameter>
            <bool>false</bool>
            <Token_string>
               <value>b.php</value>
               <source_rep>b.php</source_rep>
            </Token_string>
         </AST_actual_parameter>
      </AST_actual_parameter_list>
   </AST_method_invocation>
</AST_eval_expr>
```

This tells us that the `include` statement is an `eval_expr` node (that was obvious from the fact that we implemented `pre_eval_expr`). The `eval_expr` contains a `method_invocation` (we knew that too). The method invocation has target `%STDLIB%`, method name `include`, and a single parameter in the parameter list that contains the name of the file we are interested in. We can construct a pattern that matches this tree exactly:

```
class Expand_includes : public Tree_transform
{
public:
   void pre_eval_expr(AST_eval_expr* in, AST_statement_list* out)
   {
      // Pattern to match include statements
      Token_string* filename;
      AST_actual_parameter* param;
      AST_actual_parameter_list* params;
      Token_method_name* method_name;
      Token_class_name* target;
      AST_method_invocation* pattern;

      filename = new Token_string(WILDCARD, WILDCARD);
      param = new AST_actual_parameter(false, filename);
      params = new AST_actual_parameter_list();
      params->push_back(param);
      method_name = new Token_method_name(new String("include"));
      target = new Token_class_name(new String("%STDLIB%"));
      pattern = new AST_method_invocation(target, method_name, params);

      // Check we have a matching function
      if(!in->expr->match(pattern))
      {
         // No match; leave untouched
         out->push_back(in);
      }
      else
      {
         // Process the include
      }
   }
};
```

Note how the construction of the pattern follows the structure of the tree as output by the XML unparser exactly. The only difference is that we leave the actual filename a wildcard; obviously, we want to be able to match against any `include`, not just `include("a.php")`. Running this transform should remove the `include` from the file, but leave the other statements untouched (note that we need to `push_back in` to `out` to make sure a statement does not get deleted).

The Full Transform

Remember from the previous tutorials that code defined outside the scope of any class and any function becomes part of `%MAIN%::%run%` in phc's internal representation. So, to expand the include, we need to parse the specified file, and replace the `include` by all the statements in `%MAIN%::%run%` in the parsed script (we should also deal with the other functions of `%MAIN%`, and with any other classes or interfaces in the included script; this is left as an exercise for the reader). Here then is the full transform:

```
#include <phc/Tree_transform.h>
#include <phc/parse.h>

class Expand_includes : public Tree_transform
{
public:
   void pre_eval_expr(AST_eval_expr* in, AST_statement_list* out)
   {
      // Pattern to match include statements
      Token_string* filename;
      AST_actual_parameter* param;
      AST_actual_parameter_list* params;
      Token_method_name* method_name;
      Token_class_name* target;
      AST_method_invocation* pattern;

      filename = new Token_string(WILDCARD, WILDCARD);
      param = new AST_actual_parameter(false, filename);
      params = new AST_actual_parameter_list();
      params->push_back(param);
      method_name = new Token_method_name(new String("include"));
      target = new Token_class_name(new String("%STDLIB%"));
      pattern = new AST_method_invocation(target, method_name, params);

      // Check we have a matching function
      if(!in->expr->match(pattern))
      {
         // No match; leave untouched
         out->push_back(in);
      }
      else
      {
         // Try to open the file
         AST_php_script* php_script = parse(filename->value, NULL, false);
         if(php_script == NULL)
         {
            cout << "Could not parse file " << *filename->value;
            cout << " on line " << in->get_line_number() << endl;
            exit(-1);
         }

         // Replace the include by all statements in %MAIN%::%run%
         AST_class_def* main = php_script->get_class_def("%MAIN%");
         AST_method* run = main->get_method("%run%");
         out->push_back_all(run->statements);
      }
   }
```

```
};
```

What's Next?

This is the last tutorial in this series on using the `Tree_visitor` and `Tree_transform` classes. Of course, there is no substitute for experimentation: if you really want to understand how things works, you should implement your own transforms. Hopefully, the tutorials will help you do so. The following sources should also be useful:

- The grammar specification (and the definition of the grammar formalism)
- The explanation of how PHP gets represented in the abstract syntax
- The definition of the C++ classes for the AST nodes in `<phc/ast.h>`
- The definition of the `Tree_visitor` and `Tree_transform` classes in `<phc/Tree_visitor.h>` and `<phc/Tree_transform.h>` respectively

And of course, we are more than happy to answer any other questions you might still have. Just send an email to the mailing list and we'll do our best to answer you as quickly as possible! Happy coding!

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

What's in Store?

Below are some of the features we are planning to add to phc, approximately in the order we'd like to implement them. Version 0.1.6 saw the last major improvement of the tree transformation and tree visitor APIs; the focus is now on implementing an IR (see first section, below).

Intermediate Representation (IR)

The first step towards actual compilation will be to translate PHP scripts into an low-level intermediate representation. This representation looks like machine language (or Java bytecode), but is actually machine independent. The purpose of the IR is to define a very simple language with as few constructs as possible, that exposes most of the details that are implicit in PHP scripts, and over which we can define all sorts of compiler optimizations.

To give you an idea, an if-statement such as

```
if($c == 1)
  $d = $c * 2 + 1;
```

might look like

```
   $r := $c.equals 1
   $r := $r.not
   if $r goto l1
   $t := $c.mult 2
   $d := ^$t.add 1
l1:
```

in the intermediate representation. We are currently investigating what the IR should look like exactly.

Generating Machine Code

Once we have defined an IR and are able to translate PHP scripts into this IR, we can start generating machine code. We will generate Intel x86 code for Linux, although it should not be too difficult to add support for other systems, too (esp. similar systems such as FreeBSD). We might not support all features of PHP at first, preferring to make phc useful for a (smaller) class of scripts as soon as possible.

Link to the PHP Standard Library

Of course, the power of PHP lies in the PHP standard library. Having finished code generation, we will make sure that PHP scripts compiled with phc can make use of the standard library as-is.

Compile Extensions

We are planning to make phc generate PHP extensions that can be loaded into the PHP interpreter. This means that you can write PHP extensions *in PHP* and use phc to compile them. This will be very useful for people that write extensions for the purpose of speed or protecting proprietary code.

Static Analyses and Code Optimization

Once all that is done, we can start the really interesting work: optimizing scripts. Many of these optimizations require analyses that are defined over the IR. It is our intention to make the results of these analyses available at the AST level as well, so that it is possible to define transformations that operate on PHP source (e.g., refactoring), but make use of analyses defined on the IR. Here are a couple of optimizations we are planning to implement:

Static Single Assignment (SSA) Form

SSA is a transformation on the IR that guarantees that every variable only gets assigned in one location in the program. This is a very useful optimization, because it makes many other optimizations much easier to define. Consider the following PHP script

```
<?php
   $x = foo();
   echo $x;

   $x = $x + bar();
   echo $x;
?>
```

Gets translated into

```
<?php
   $x0 = foo();
   echo $x0;

   $x1 = $x0 + bar();
   echo $x1;
?>
```

This makes the relation between the assignments to variables and the use of variables more obvious. This is useful for a number of things. For example, consider a refactoring to make sure that the result of `dbx_connect` always get stored in a variable called `dbx_link`. Then, the following code

```
<?php
   $x = dbx_connect(...);
   some_function($x);

   $x = some_other_function();
   some_function($x);
?>
```

should get translated to

```
<?php
   $dbx_link = dbx_connect(...);
   some_function($dbx_link);

   $x = some_other_function();
   some_function($x);
```

```
?>
```

In particular, the second `some_function($x)` should not be modified. Converting the program to SSA form first will make this easier.

Type Inference

Type inference tries to find out the type of each variable. We will do type inference on the SSA form of the program. This means that every variable can only have a single type (because it is only assigned once). Apart from compilation, type inference is useful for numerous other tasks. As a simple example, consider implementing a semantic checker (see Spinoff Projects). For example, in the following code,

```
$x = $x + $y;
```

if we can deduce that both `$x` and `$y` have type `string`, we might issue a warning that the plus (+) should probably be a dot (`.`) (PHP will warn for this at run-time if `error_reporting` is set high enough, but it is useful to catch these errors at compile time.)

For a more complicated example, consider renaming a class method. If we rename method `foo` to `bar` in class *A* (but not in class *B*) in the following example,

```
<?php
   class A
   {
      function foo
      {
         echo "Hello ";
      }
   }

   class B
   {
      function foo
      {
         echo "world!";
      }
   }

   $a = new A();
   $b = new B();

   $a->foo();
   $b->foo();
?>
```

we should get

```
<?php
   class A
   {
      function bar
      {
         echo "Hello ";
      }
   }

   class B
   {
      function foo
      {
```

```
            echo "world!";
        }
    }

    $a = new A();
    $b = new B();

    $a->bar();
    $b->foo();
?>
```

In particular, the line `$b->foo();` should not be modified. We can only do this if we know that the type of `$a` is A, and the type of `$b` is B. Type inference cannot always be 100% accurate; in such a case, refactoring should fail (or insert code to explicitly distinguish between a few types).

So stay tuned :-)

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Home | Downloads | Documentation | Plugins | Spinoff Projects | Mailing List

The Abstract Grammar

This is the full and authoritative definition of the phc abstract grammar for PHP in maketea format (this can also be found in `generated_src/phc.tea` in the distribution). For a description of the structure of the grammar, and how it converts to C++ code, refer to the Grammar Formalism.

Overall Structure

```
php_script ::= interface_def* class_def+ ;

interface_def ::= INTERFACE_NAME extends:INTERFACE_NAME* member* ;

class_def ::= class_mod CLASS_NAME extends:CLASS_NAME?
   implements:INTERFACE_NAME* member* ;
class_mod ::= "abstract"? "final"? ;

member ::= method | attribute ;

method ::= signature statement*? ;
signature ::= method_mod is_ref:"&"? METHOD_NAME formal_parameter* ;
method_mod ::= "public"? "protected"? "private"?
   "static"? "abstract"? "final"? ;
formal_parameter ::= type is_ref:"&"? VARIABLE_NAME expr? ;
type ::= "array"? CLASS_NAME? ;

attribute ::= attr_mod VARIABLE_NAME expr? ;
attr_mod ::= "public"? "protected"? "private"? "static"? "const"?  ;
```

Statements

```
statement ::=
     if | while | do | for | foreach
   | switch | break | continue | return
   | static_declaration
   | unset | declare | try | throw | eval_expr ;

if ::= expr iftrue:statement* iffalse:statement* ;
while ::= expr statement* ;
do ::= statement* expr ;
for ::= init:expr? cond:expr? incr:expr? statement* ;
foreach ::= expr key:variable? is_ref:"&"?
   val:variable statement* ;

switch ::= expr switch_case* ;
switch_case ::= expr? statement* ;
break ::= expr? ;
continue ::= expr? ;
return ::= expr? ;

static_declaration ::= VARIABLE_NAME expr? ;
unset ::= variable ;
```

```
declare ::= directive+ statement* ;
directive ::= DIRECTIVE_NAME expr ;

try ::= statement* catches:catch* ;
catch ::= CLASS_NAME VARIABLE_NAME statement* ;
throw ::= expr ;

eval_expr ::= expr ;
```

Expressions

```
expr ::=
     assignment | list_assignment | cast | unary_op | bin_op
   | conditional_expr | ignore_errors | constant | instanceof
   | variable | pre_op | post_op | array
   | method_invocation | new | clone
   | literal ;

literal ::= INT | REAL | STRING | BOOL | NULL ;

assignment ::= variable is_ref:"&"? expr ;

list_assignment ::= list_elements expr ;
list_elements ::= list_element?* ;
list_element ::= variable | list_elements ;

cast ::= CAST expr ;
unary_op ::= OP expr ;
bin_op ::= left:expr OP right:expr ;

conditional_expr ::=
   cond:expr iftrue:expr iffalse:expr ;
ignore_errors ::= expr ;

constant ::= CLASS_NAME CONSTANT_NAME ;

instanceof ::= expr class_name ;

variable ::= target? variable_name
   array_indices:expr?* string_index:expr? ;
variable_name ::= VARIABLE_NAME | reflection ;
reflection ::= expr ;

target ::= expr | CLASS_NAME ;

pre_op ::= OP variable ;
post_op ::= variable OP ;

array ::= array_elem* ;
array_elem ::= key:expr? is_ref:"&"? val:expr ;

method_invocation ::= target method_name actual_parameter* ;
method_name ::= METHOD_NAME | reflection ;

actual_parameter ::= is_ref:"&"? expr ;

new ::= class_name actual_parameter* ;
class_name ::= CLASS_NAME | reflection ;

clone ::= expr ;
```

Additional Structure

```
node ::=
     php_script | class_mod | signature
   | method_mod | formal_parameter | type | attr_mod
   | static_var | directive | list_element | variable_name | target
   | array_elem | method_name | actual_parameter | class_name
   | commented_node | expr | identifier
   | formal_parameter* | directive* | array_elem* | actual_parameter*
   | INTERFACE_NAME* | list_element* | expr*
   ;

commented_node ::=
     member | statement | interface_def | class_def | switch_case
        | catch interface_def* | class_def* | member* | statement*
        | switch_case* | catch*
   ;

identifier ::=
     INTERFACE_NAME | CLASS_NAME | METHOD_NAME | VARIABLE_NAME
   | DIRECTIVE_NAME | CAST | OP | CONSTANT_NAME
   ;
```

Mix-in Code

The code generated based on the grammar listed above can be extended by “mix-in” code, which adds fields or methods to the class structure generated by maketea. For a full listing of the mix-in code, see `generated_src/phc.tea` in the phc distribution.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Representing PHP

Most PHP constructs can immediately be represented in terms of the phc abstract grammar. There are a few constructs that present some difficulties. This document describes how these difficulties are resolved, and it explains some of the more difficult rules in the grammar.

Top Level Grammar Structure

The major difference between our abstract grammar for PHP and the “official” grammar (distributed in source code format with the PHP distribution) is the top-level structure. Stripped-down, the top-level of the PHP grammar looks something like

```
php_script ::= statement*

statement ::= class_def | method | if | while | ... other statements ...

method ::= statement*

class_def ::= member*
member ::= method | attribute
```

Compare this to the top-level grammar structure that we have adopted:

```
php_script ::= class_def+

class_def ::= member*
member ::= method | attribute

method ::= statement*

statement ::= if | while | ... other statements ...
```

(This shows essentials only; see the grammar for the details).

This mismatch has two consequences. The first is that PHP allows scripts have methods that do not belong to any class, and statements that do not belong to any method. phc introduces a special class called `%MAIN%` for this purpose. All functions defined outside the scope of any class get added as a static method to `%MAIN%`, and all statements defined outside the scope of any method get added to a special method `%run%` (in `%MAIN%`). Thus, the following simple PHP script

```
<?php
   function hello()
   {
      echo "Hello world!";
   }

   hello();
?>
```

gets represented as

```
<?php
   class %MAIN%
   {
      static function hello()
      {
         %STDLIB%::echo("Hello world!");
      }

      static function %run%()
      {
         %MAIN%::hello();
      }
   }
?>
```

The second consequence is that PHP allows scripts to have function definitions *inside* other function definitions (or inside `if`-statements, `while`-loops, etc.). This is not correctly supported by phc; see limitations.

Method `targets`

Recall the grammar rules for method invocations:

```
method_invocation ::= target method_name actual_parameter* ;
method_name ::= METHOD_NAME | reflection ;
```

As explained above, phc thinks of a PHP script as consisting of a set of classes. That means that a function call must either be invoked on an object, or it must be a static method in some class. The grammar rule for `target` is

```
target ::= expr | CLASS_NAME ;
```

So, a `target` is either an expression (for example, in `$x->foo()`), or a class name (in `CLASS::foo()`).

When the user does not explicitly specify a target (for example, the call to `hello()` in the example above), phc will automatically insert a target. If the method that gets invoked is defined in `%MAIN%` (i.e., the user provided an implementation), the target will be set to `%MAIN%` (for example, the call to `hello()`). Otherwise, the target is set to `%STDLIB%` (for example, the call to `echo`). Like `%MAIN%`, `%STDLIB%` is a special class that collects all methods defined in the PHP standard library. (Incidentally, if PHP6 implements namespaces, namespaces will probably be represented similarly.)

Variables

The grammar rule for variables reads

```
variable ::= target? variable_name array_indices:(expr?)* string_index:expr?
variable_name ::= VARIABLE_NAME | reflection
```

This is probably one of the more difficult rules in the grammar, so it is worth explaining in a bit more detail. The following table describe each element of the first rule in detail.

`target?`	Just like function calls, variables can have a target, and just as for function calls, this target can be an expression (for an object, e.g., `$x->y`) or a class

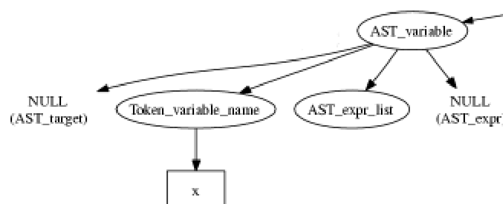
name (for a static class attribute, e.g. `FOO::$y`). Unlike function calls however, in variables the target is optional (indicated by the question mark). If no target is specified, the variable refers to a *local* variable in a method (see Global Variables for information on how we deal with the `global` statement).

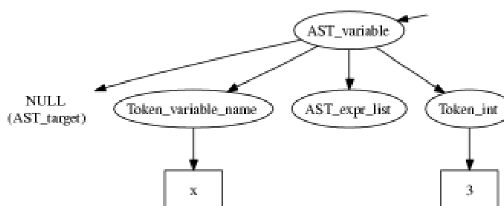
`variable_name`
: Again, as for function calls, the name of the variable may be a literal `VARIABLE_NAME` (`$x`), or be given by an expression (which is wrapped up in an `AST_reflection` node). The latter possibility is referred to as "variable variables" in the PHP manual. For example, `$$x` is the variable whose name is currently stored in (another) variable called `$x`.

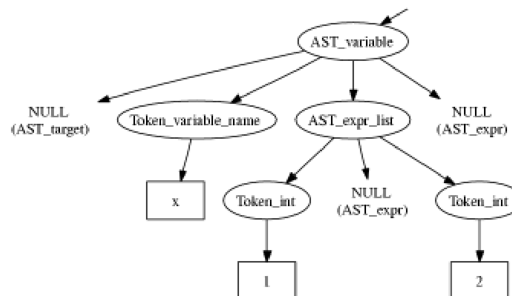
`array_indices:``(expr?)*`
: A variable may have one or more array indices, for example `$x[3][5]`. The strange construct `(expr?)*` means: a list of (`*`) optional (`?`) expressions. For example, `$x[4][]` is a list of two expressions, but the second expression is not given. In PHP, this means "use the next available index".

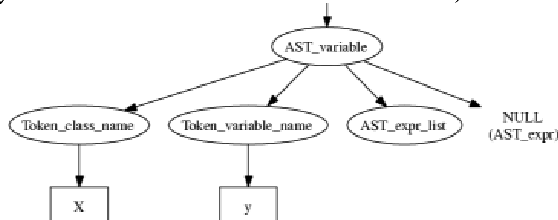
`string_index:``expr?`
: Finally, a variable may contain one string index (`$x{5}`) that accesses an individual character from a string.

We illustrate the various possibilities using diagrams:

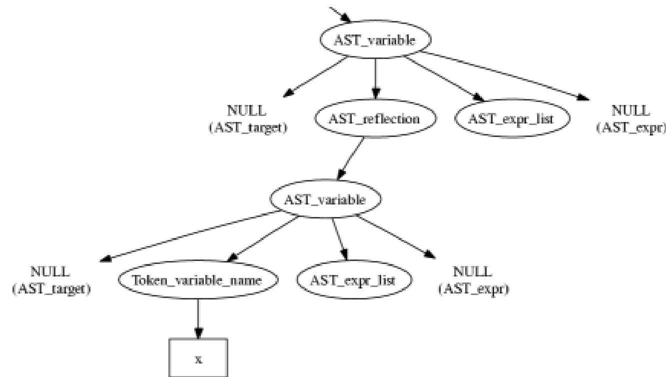
- 

  The simple case: `$x`
  Note that the name of the variable is `x`, not `$x`
- 

  Using a string index: `$x{3}`
- 

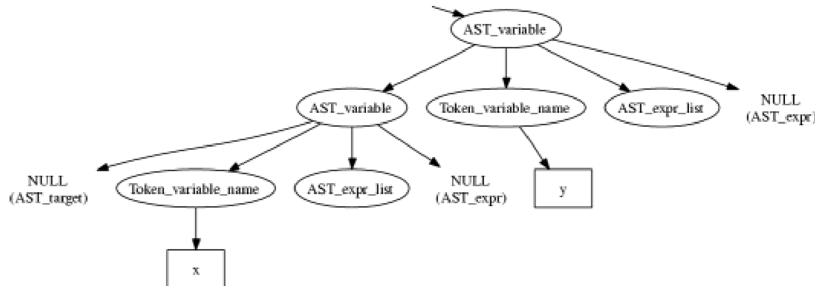
  Using array indices: `$x[1][][2]`
  (Note that the empty array index means "next available" in PHP)
- 

  Class constants: `X::$y`
  Note that here also the variable name is `y`, not `$y`. The fact that you must write `$x->y` but `X::$y` in PHP disappears in the abstract syntax.
- Variable variables: `$$x`



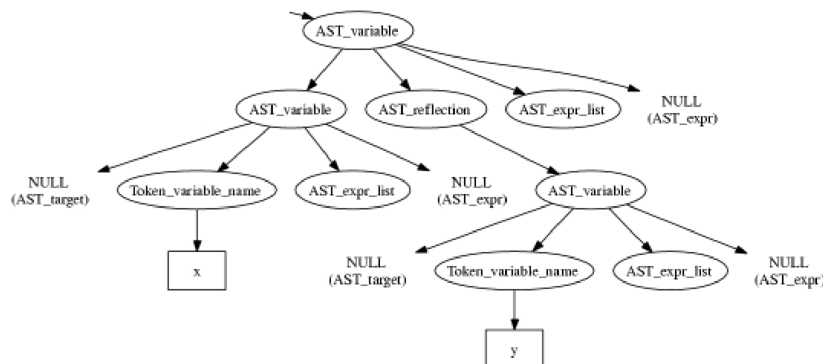
Note how the name of the variable (second component) is now given by another variable.

- Object attributes: `$x->y`



Note that the target is now given by a variable.

- Variable object attributes: `$x->$y`



Both the target and the variable name are given by (other) variables.

Comments

A number of nodes in the AST are dedicated "commented nodes". Their corresponding C++ classes inherit from `AST_commented_node`, which introduces a `List<String*>*` attribute called `comments`. The commented nodes are class members (`AST_member`), statements (`AST_statement`), interface and class definitions (`AST_interface_def`, `AST_class_def`), switch cases (`AST_switch_case`) and catches (`AST_catch`).

When the parser encounters a comment in the input, it attaches it either to the previous node in the AST, or to the next, according to a variable `attach_to_previous`. This variable is set as follows:

- It is reset to `false` at the start of each line
- It is set to `true` after seeing a semicolon, or either of the keywords `class` or `function`

Thus, in

```
foo();
// Comment
```

```
bar();
```

the comment gets attached to `bar();` (to be precise, to the corresponding `AST_eval_expr` node; the function call itself is an expression and phc does not associate comments with expressions), but in

```
foo(); // Comment
bar();
```

the comment gets attached to `foo();` instead. The same applies to multiple comments:

```
foo(); /* A */ /* B */
// C
// D
bar();
```

In this snippet, `A` and `B` get attached to `foo();`, but `C` and `D` get attached to `bar();`. Also, in the following snippet,

```
// Comment
echo /* one */ 1 + /* two */ 2;
```

all comments get attached to the same node. This should work most of the time, if not all the time. In particular, it should never loose any comments. If something goes wrong with comments, please send us a sample program that shows where it goes wrong. Note that whitespace in multi-line comments gets dealt with in a less than satisfactory way; see limitations for details for details.

String parsing

Double quoted strings and those written using the HEREDOC syntax are treated specially by PHP: it parses variables used inside these strings and automatically expands them with their value. phc handles both the simple and complex syntax defined by PHP for variables in strings. We transform a string like

```
"Total cost is: $total (includes shipping of $shipping)"
```

into:

```
"Total cost is: " . $total . " (includes shipping of " . $shipping . ")"
```

which is represented in the phc abstract syntax tree by a number of strings and expressions concatenated together. Thus, as a programmer you don't need to do anything special to process variables inside strings. Any code you write for processing variables will also appropriately handle variables inside strings.

Currently, the unparser will *not* output strings of the first form, but will always output them in the second form (using concatenation). Future releases of phc may remedy this (alternatively, dedicated more advanced pretty-printing tools for PHP could be built on top of phc; see Spinoffs).

Global Variables

Global variable declarations are not explicitly recorded in the phc AST. Instead the local variable declared global is assigned a reference to the appropriate global variable, which will be a static class attribute of %MAIN% (see above description of the Top Level Grammar Structure).

For example, the following code

```
<?php
   $x = 100;

   function foobar()
   {
      global $x;
      $x = 200;
   }

   foobar();
?>
```

is represented internally as

```
<?php
   class %MAIN%
   {
      static $x = 100;

      static function foobar()
      {
         $x =& %MAIN%::$x;
         $x = 200;
      }

      static function %run%()
      {
         %MAIN%::foobar();
      }
   }
?>
```

Obviously, the phc unparser will output code using global declarations.

Note that local variables in `%run%` are really global variables; for that reason, any “local” variable in `%run%` get assigned a target of `%MAIN%` (if no target was specified in the program).

elseif

The abstract grammar does not have a construct for `elseif`. The following PHP code

```
<?php
   if($x)
      c1();
   elseif($y)
      c2();
   else
      c3();
?>
```

gets interpreted as

```
<?php
   if($x)
      c1();
   else
   {
      if($y)
         c2();
      else
         c3();
   }
```

```
?>
```

The higher the number of `elseifs`, the greater the level of nesting. This transformation is “hidden” by the unparser.

Miscellaneous Other Changes

- If `echo` has multiple (comma separated) arguments, they get translated into multiple function calls (`echo a, b;` becomes `echo a; echo b;`). Fragments of inline HTML also become arguments to a function call to `echo`.
- The keywords `use`, `require`, `require_once`, `include`, `include_once`, `isset` and `empty` all get translated into a function call to a function with the same name as the keyword
- `exit` also becomes a call to the function `exit`; `exit;` and `exit();` are interpreted as `exit(0)`
- Class attribute declarations can only declare a single attribute per declaration in the abstract syntax; thus, `var x, y;` becomes `var x; var y;`. A similar comment applies to `static_var`
- We do not support the += style of operators; `a += 2;` gets translated into `a = a + 2;`. It should be possible to reverse this translation in the unparser, but this is not currently implemented.

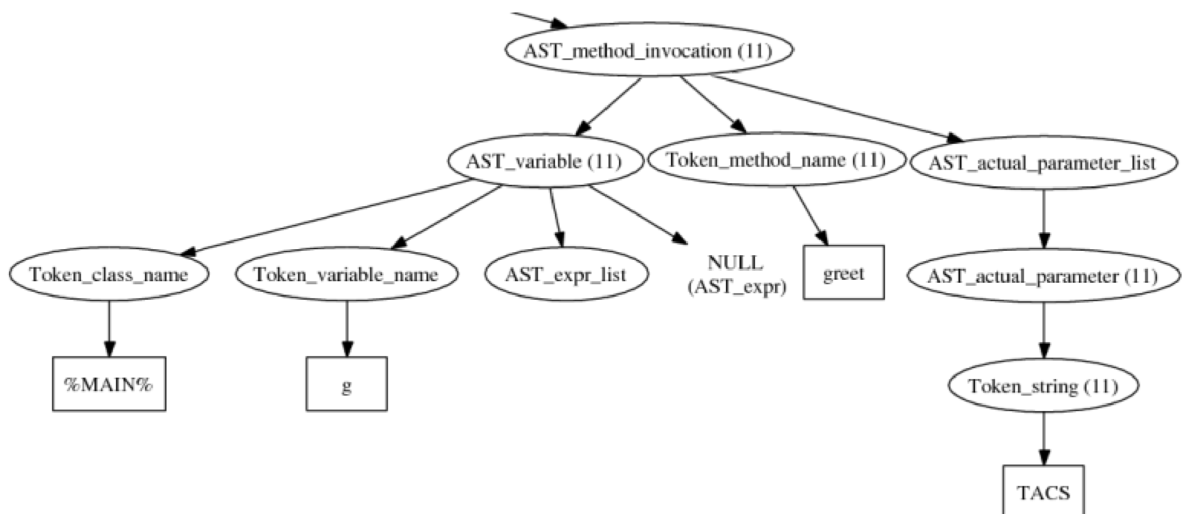
Comparison to the PHP grammar

Finally, the phc grammar is much simpler than the official grammar, and as a consequence more general. The class of programs that are valid according to the abstract grammar is larger than the class of programs actually accepted by the PHP parser. In other words, it is possible to represent a program in the abstract syntax that does not have a valid PHP equivalent. The advantage of our grammar is that is much, *much* easier to work with.

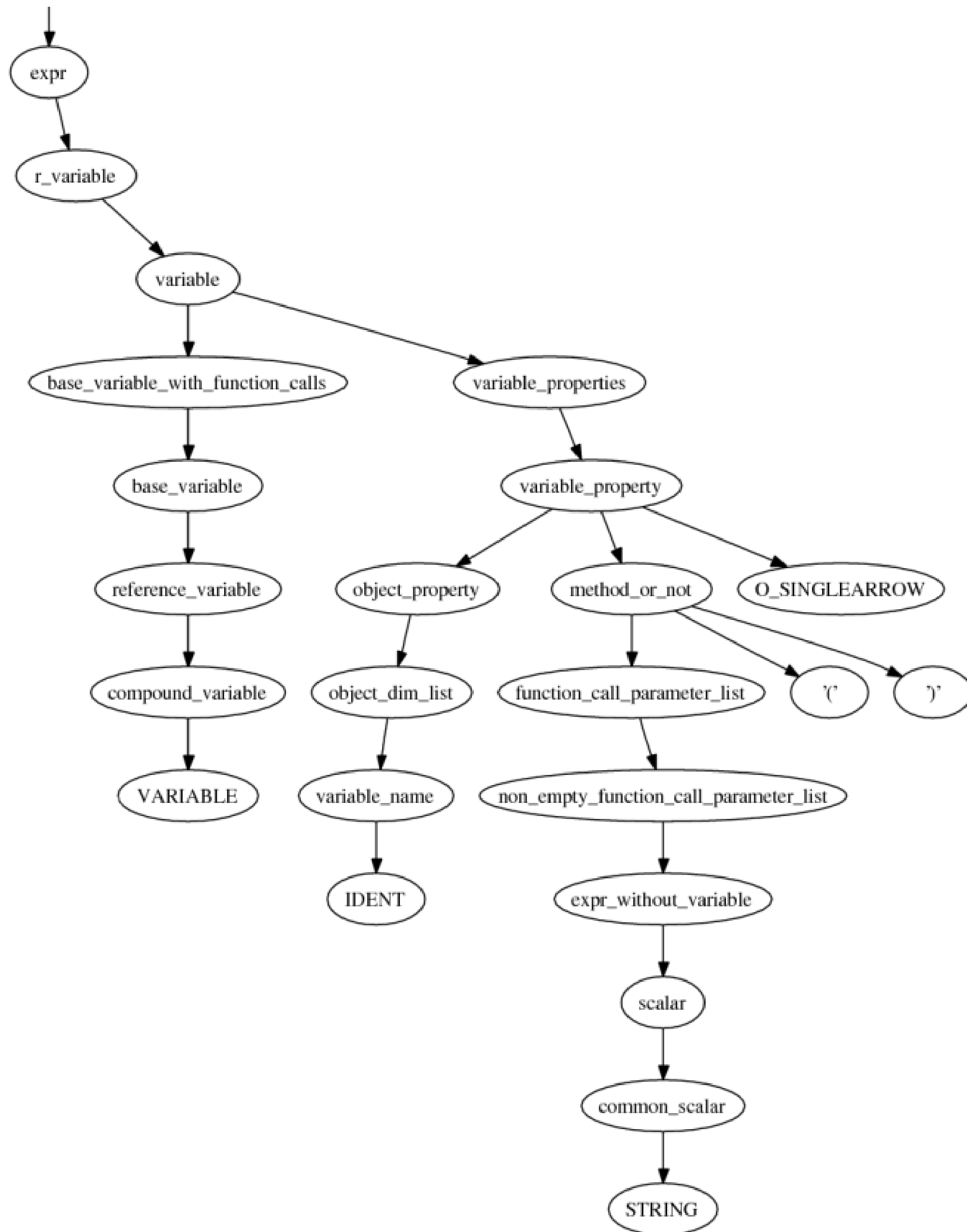
To compare, consider the tree for

```
$g->greet("TACS");
```

Using the phc abstract syntax, this looks like



However, in the official PHP grammar, the tree would look like



Not only is the number of concepts used in the tree much larger (`base_variable_with_function_calls`, `reference_variable`, `variable_property`, etc. etc.), the concepts used in the phc tree map directly to constructs in the PHP language; that does not hold true for the PHP tree. Moreover, the fact that this expression is a method invocation (function call) is immediately obvious from the root of the expression in the phc tree; the root of the PHP tree says that the expression is a variable, and only deeper down the tree does it become apparent that the expression is in fact a function call.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Overview of the AST classes and transformation API

This document explains the code for the AST classes, tree visitor API and tree transformation API. All this code is generated by a tool called maketea. It does not explain how this code is derived from the phc grammar; some of the details of this process are explained in the maketea theory.

The AST classes

There are two main kinds of AST classes: classes that correspond to non-terminals in the grammar, and classes that correspond to terminals in the grammar. These two kinds are called `AST_xxx` and `Token_xxx` respectively. Examples are `AST_while`, `AST_expr`, `Token_method_name` and `Token_int`.

The main difference is that terminal classes have one additional field (and sometimes two). Every token class gets an attribute called `value`. The type of this attribute depends on the token; for most tokens it is `String*` (this is the default); however, if the grammar explicitely specifies a type for the value (in angular brackets, for example `REAL<double>`), this overrides the default. If the default value is overridden, the token class gets an additional attribute `source_rep`, which corresponds to the value of the token in the source value. The type of `source_rep` is always `String*`. For example, the real number `5E-1` might have `value` set to the `double` 0.5, but `source_rep` set to `"5E-1"`. Similarly, a string `__FILE__` might have `value` set to `/home/joe/myscript.php`, but `source_rep` set to `__FILE__`. If the type of the `value` attribute it set to be empty, the token class does not get a value but (but it will get a `source_rep` field). This is the case for `Token_null<>` in the phc grammar.

In addition, all the tokens classes have a method called `get_value_as_string()` and a method `get_source_rep` when applicable. This is useful for programs that operate on general `AST_identifier` objects (such as `Token_method_name` or `Token_class_name`) or `AST_literal` (such as `Token_real` or `Token_int`). Note that the value returned by `get_value_as_string()` and `get_source_rep()` may be different; for example, `get_source_rep()` might return `0.5E-1`, while `get_value_as_string()` might return `0.5`.

All (non-terminal and terminal) then provide the following methods for deep equality, pattern matching, cloning, calling a tree visitor and calling a tree transformer. These methods are explained separately in sections below.

Deep Equality

Deep equality is implemented by `bool deep_equals(Object* other)`. It takes into account the entire tree structure generated by maketea, including any fields that are specified in the mixin code in the grammar. Thus, `deep_equals` also compares line numbers, comments, etc.

Cloning

Cloning is implemented by deep_clone. Cloning makes a (deep) copy of a tree, so the set of all pointers in the new tree is completely distinct from the set of pointers in the old tree. The only exception to this rule is that cloning the WILDCARD objects (see pattern matching, below) returns the WILDCARD object itself.

Pattern Matching

Pattern matching is implemented by bool match(Object* pattern). Pattern matching differs from deep equality in two ways. First, it does not take into account any fields added by the mixin code; for example, it does not compare line numbers or comments.

Second, match supports the use of wildcards. maketea generates a special class called Wildcard. You should never instantiate this class directly; in <phc/ast.h>, you will find the following declaration:

```
extern Wildcard* WILDCARD;
```

This WILDCARD is the sole instance of Wildcard. When match encounters a reference to this object in a pattern, it does two things: it skips that field in the comparison (so it acts as a “don't care”), and it replaces the value of the field in the pattern by the value in the tree. For example, in the body of the if in

```
Token_class_name* name = new Token_class_name(new String("SOME_CLASS"));
Token_class_name* pattern = new Token_class_name(WILDCARD);

if(name->match(pattern))
{
   // ...
}
```

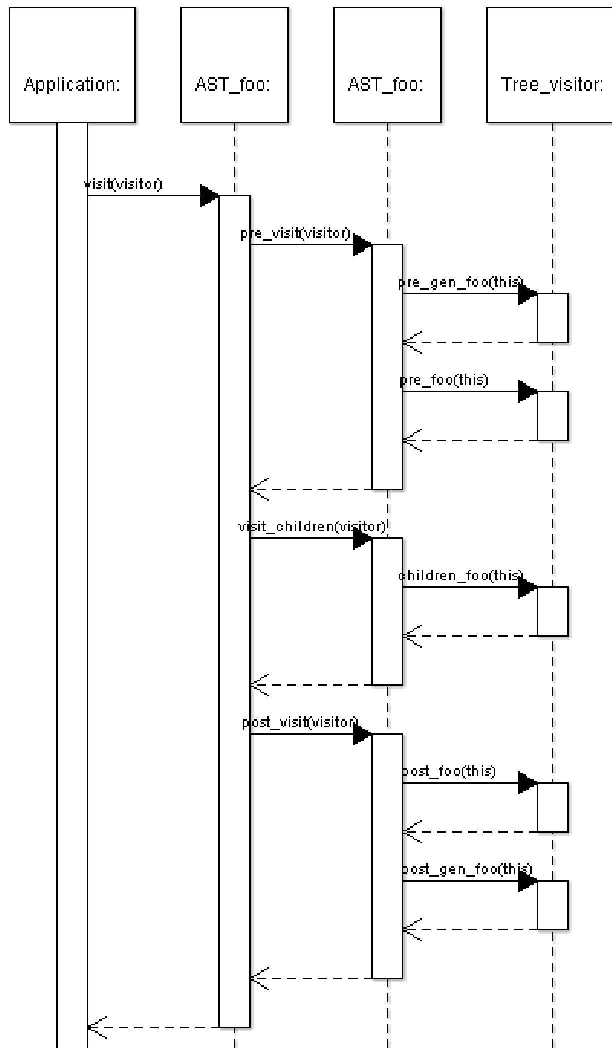
pattern->value will be set to the corresponding value in name. Tutorials 3 and 4 include examples of the use of wildcards.

Calling any methods on the WILDCARD object other than deep_clone will lead to a runtime error.

The Tree Visitor API

Every AST class provides four methods to support the tree visitor API: void visit(Tree_visitor*), void pre_visit(Tree_visitor*), void visit_children(Tree_visitor*) and void post_visit(Tree_visitor*). The implementation of each of these methods is very simple.

visit simply calls pre_visit, visit_children and post_visit in order. It could have been implemented once and for all in the AST_node class (but is not, for no particular reason).

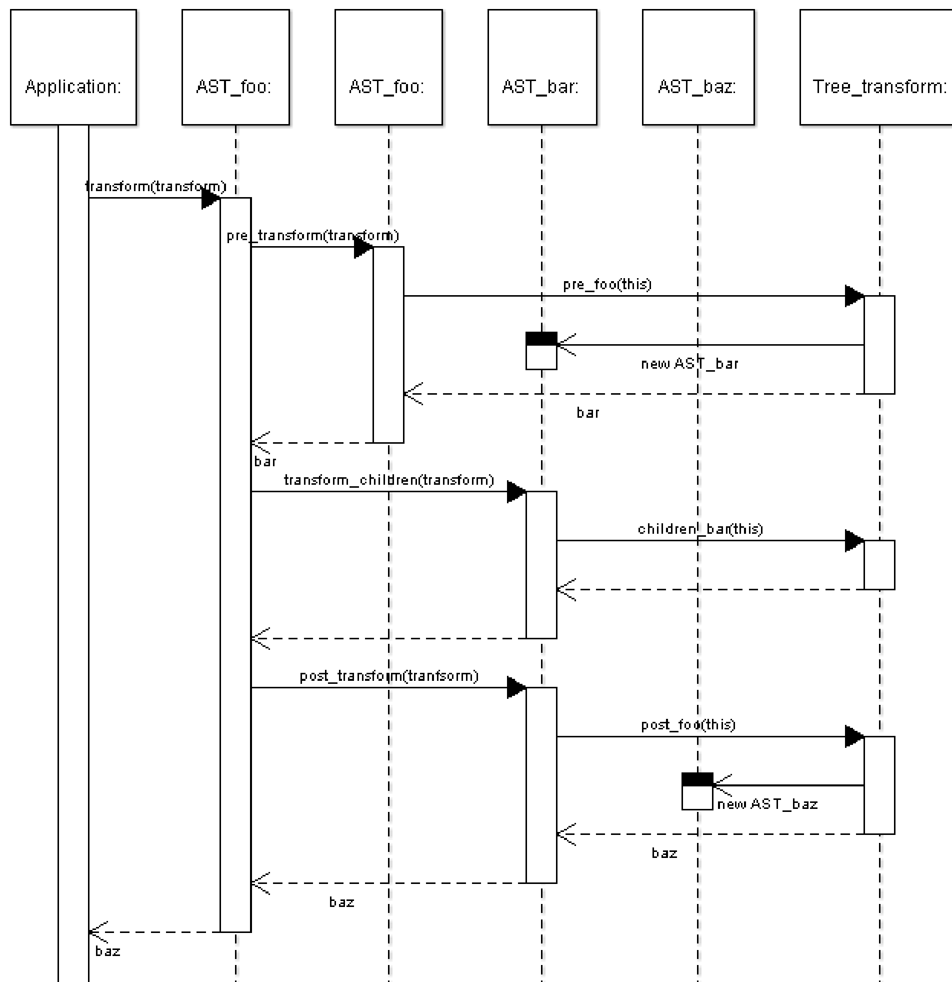


For a node `x0`, which inherits from `x1`, which inherits from `x2`, which in turn inherits from `x3`, etc., `x0::pre_visit` calls `pre_x3`, `pre_x2`, `pre_x1` and `pre_x0`, in that order, on the tree visitor object, passing itself as an argument. If `x0` inherits from multiple classes, all of the appropriate visitor methods will be invoked. However, if `x0` inherits from both `x1a` and `x1b`, the programmer should not rely on the relative order of `pre_x1a` and `pre_x1b`.

`x0::visit_children` simply calls `children_x0`.

`x0::post_visit` will call `post_x0`, `post_x1`, etc. Again, if `x0` inherits from both `x1a` and `x1b`, the programmer should not rely on the relative order of `post_x1a` and `post_x1b`. The only guarantee made by maketea is that the order of the pre-methods will be the exact reverse of the order of the post-methods.

The Tree Transform API



Every AST class `AST_foo`, which inherits from `AST_gen_foo` provides four methods to support the tree visitor API: `AST_gen_foo* transform(Tree_transformer*)`, `AST_gen_foo* pre_transform(Tree_transformer*)`, `void transform_children(Tree_transformer*)` and `AST_gen_foo* post_transform(Tree_transformer*)`. It is not entirely as straightforward as this; if `AST_foo` inherits from more than one class, the return type would probably be `AST_foo`; in some cases, `transform` might return a `AST_foo_list` instead. See the section on context resolution in the grammar formalism for details; here we consider the programmer's perspective only. The exact signatures for a particular class can always be found in `generated/ast.h`.

As with the tree visitor API, `transform` calls `pre_transform`, `transform_children` and `post_transform`. However, while `transform` calls `pre_transform` on itself, it calls `transform_children` and `post_transform` on the node returned by `pre_transform`. If `pre_transform` returns a vector, `transform` calls `transform_children` and `post_transform` on every element in that vector, assembling all the results.

`pre_transform` and `post_transform` simply call the appropriate method in the `Tree_transform` object. However, if `pre_transform` (or `post_transform`) returns a list of nodes, the corresponding method in the tree transform object will expect two arguments: the node to be transformed, and an empty list of nodes that will be the return value of `pre_transform`. In that case, `pre_transform` will first create a new empty list, pass that in as the second argument to the corresponding method in the tree transform object, and then return that list.

`transform_children` just calls the corresponding method in the tree transform object.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Limitations

This document describes the known limitations of the current phc implementation. These limitations are things that we are aware of but that are not high on our priority list of things to deal with at the moment. However, if any of them are bothering you, let us know and we might look into it.

Nested Function Definitions

As described in Representing PHP, we cannot deal with nested function definitions. phc will break on the following PHP code:

```
<?php
   if($x)
   {
      function f()
      {
         echo "First f";
      }
   }
   else
   {
      function f()
      {
         echo "Second f";
      }
   }

   f();
?>
```

Currently phc will generate the following AST for this:

```
<?php
   function f()
   {
      echo "First f";
   }

   function f()
   {
      echo "Second f";
   }

   if($x)
   {
   }
   else
   {
   }

   f();
?>
```

Comments

Representing PHP explains how we deal with comments. Most comments in a PHP script should get attached to the right token in the tree, and no comments should ever be lost. If that is not true, please send us a sample program that demonstrates where it breaks. There are a few problems that we are aware of, and there are probably others too.

Dealing with comments in a completely satisfactory way is a difficult task! The first problem with our method of dealing with comments is how we deal with whitespace in multi-line comments. Consider the following example.

```
<?php
   /*
    * Some comment with
    * multiple lines
    */
   foo();
?>
```

For clarity, the contents of the comment are underlined. In particular, notice that the whitespace at the start of the line is included in the comment. This means that when the unparser outputs the comment, it outputs something like

```
<?php
   /*
    * Some comment with
       * multiple lines
       */
   foo();
?>
```

It is unclear how to solve this problem nicely. Suggestions are welcome :-)

Second, it is not currently possible to associate a comment with the `else`-clause of an `if`-statement. Thus, in

```
<?php
   // Comment 1
   if($c)
   {
      foo();
   }
   // Comment 2
   else
   {
      bar();
   }
?>
```

Comment 2 will be associated with the call to `bar` (but Comment 1 will be associated with the `if`-statement itself). A similar problem occurs with comments for `elseif` statements.

Finally, if a scope ends on a comment, that comment will be associated with the wrong node. For example, in

```
<?php
   if($c)
   {
      echo "Hi";
```

```
    }
    else
    {
       // Do nothing
    }

    echo "World";
?>
```

the comment will be associated with the `echo "World"` statement. A similar problem occurs when a script ends on a comment; that comment will not be lost, but will be associated with the last node in the script.

Numbers

PHP accepts invalid octal numbers such as `01090`; the “incorrect tail” is silently ignored (so, this number should evaluate to 8 decimal). The phc lexical analyser will generate an “invalid token” instead which will result in a syntax error.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

maketea Theory

maketea is a tool bundled with phc which, based on a grammar definition of a language, generates a C++ hierarchy for the corresponding abstract syntax tree, a tree transformation and visitor API, and deep cloning, deep equality and pattern matching on the AST. In this document we describe the grammar formalism used by maketea, how a C++ class structure is derived from such a grammar, and explains how the tree transformation API is generated. The generated code itself is explained in another document.

The Grammar Formalism

The style of grammar formalism used by maketea is sometimes referred to as an “object oriented” context free grammar. It facilitates a trivial and reliable mapping between the abstract grammar, and the actual (C++) abstract syntax tree (AST) that is generated by the phc parser.

We make a distinction between three types of symbols: *non-terminal* symbols, *terminal symbols* and *markers*. Non-terminal symbols have the same function in our formalism as in the usual BNF formalism, and will not be further explained. We denote non-terminal symbols in lower case in the grammar (e.g., `expr`).

The distinction between terminal symbols and markers is non-standard. Markers have no semantic value other than their presence; an example is `"abstract"`. Thus, the semantic value of a marker is a boolean value; it is either there, or it is not (note that this is different from a symbol such as the semi-colon, which has *no* semantic value whatsoever, and thus does not need to be included in an abstract syntax tree). Conversely, the semantic value of a *terminal symbol* is an arbitrary value; an example is `CLASS_NAME` (the structure of a terminal symbol may be defined by a regular expression; this is irrelevant as far as the abstract grammar is concerned). We denote markers in quotes (`"abstract"`), and terminal symbols in capitals (`CLASS_NAME`).

Each non-terminal symbol `a` will have a single production in the grammar. Instances of `a` in the AST will be represented by a class called `AST_a`. The attributes of `AST_a` will depend on the production for `a` (see below).

A terminal symbol `x` will be represented by a class `Token_x`. Every token class gets an attribute called `value`. The type of this attribute depends on the token; for most tokens it is `String*` (this is the default); however, if the grammar explicitely specifies a type for the value (in angular brackets, for example `REAL<double>`), this overrides the default. If the default value is overridden, the token class gets an additional attribute `source_rep`, which corresponds to the value of the token in the source value. The type of `source_rep` is always `String*`. If the type of the `value` attribute it set to be empty, the token class does not get a value but (but it will get a `source_rep` field).

Finally, a marker will not be represented by a specialised class. Instead, a marker `"foo"` may **only** appear as an optional symbol in a production rule (`a ::= ... "foo"? ...`), and will appear as a boolean attribute `is_foo` in the class representing `a` (`AST_a`).

There are only two types of rules in the grammar. The first is the simplest, and list a number of alternatives for a non-terminal symbol `a`:

```
a ::= b | c | ... | z
```

Here, each of `b`, `c`, ..., `z` must be a single non-terminal symbol. This rule results in a (usually) empty `class AST_a {}`, which acts as a superclass for the classes for `b`, `c`, ..., `z`. This reflects the semantics of the rule (a `b` *is* an `a`); if there are multiple rules `a ::= c|...`, `b ::= c|...`, class `AST_c` will inherit from both `AST_a` and `AST_b`. This type of rule is exemplified by the production for `statement` in the grammar. There is one additional requirement for disjunction rules, which will be explained in the section on context resolution, below.

The second type is the most common:

```
a ::= b c ... z
```

In this rule, each of the `b`, `c`, ..., `z` is an arbitrary symbol (non-terminal, terminal or marker), which may be optional (`b?`) or repeated (`b*` or `b+`). This type of rule must not include any disjunctions (`a|b`), and only single symbols can be repeated (no grouping). If a symbol `b` can be repeated, it will be represented by a specialised list class `AST_list_b` (which inherits from the STL `list` class) in the tree. In addition, the symbols may be labeled (`label:symbol`). This does not add to the grammar structure, but explains the purpose of the symbol in the rule, and will be used for the name of the attribute of the corresponding class. The default name for each class attribute depends on the corresponding type: an attribute of type `AST_variable_name` (corresponding to a non-terminal `variable_name`) will be called `variable_name`. The default name for an attribute of type `AST_foo_list` will be *foos*. However, as mentioned above, this can be overridden by specifying a label.

As an example, consider the rule for `variable` in the grammar.

```
expr ::= ... | variable | ... ;
variable ::= target? variable_name array_indices:expr?* string_index:expr? ;
```

A `variable` is an `expr`, so that `variable` is represented by the class shown below. The optionality of `string_index` is not reflected directly in the class definition, but simply means that the `string_index` field in the class may be NULL.

```
class AST_variable : virtual public AST_expr
{
public:
   AST_target* target;
   AST_variable_name* variable_name;
   AST_expr_list* array_indices;
   AST_expr* string_index;
}
```

A final note on combining `*` and `?`. The construct `(a*)?` denotes an optional list of `a`s. Thus, it will be represented by an `AST_a_list`. If a list is specified, but empty, the list will simply contain no elements. If the list is not specified at all, the list will be NULL. This is used, for example, to distinguish between methods that contain no statements and abstract methods. Similarly, `(a?)*` is a (non-optional) list of optional `a`s. Thus, this is a list, but elements of the list may be NULL. This is used for example to denote empty array indices (`a[]`) in the rule for `variable`.

Context Resolution

We also derive the tree visitor API and tree transformation API from the grammar. The tree visitor API is very simple to derive; see the overview of the generated code for an explanation. The tree transformation API however is slightly more difficult to derive. The problem is to decide the signatures for the transform methods, or in other words, what can transform into what? For example, in the phc grammar for PHP, the transform for an if-statement should be allowed return a list of statements of any kind (because it is safe to replace an if-statement by a list of statements). Similarly, a binary operator should be allowed return any other expression (but not a list of them). For reasons that will become clear very soon, we call the process of deciding these signatures "context resolution".

Contexts

A context is essentially a use of a symbol somewhere in a (concrete) rule in the grammar. There are four possibilities. Consider:

```
concrete1 ::= ...
concrete2 ::= ...
concrete3 ::= ...
concrete4 ::= ...
concrete5 ::= ...
concrete6 ::= ...
abstract1 ::= concrete3 | concrete4
abstract2 ::= concrete5 | concrete6

some_concrete_rule ::= concrete1 concrete2* abstract1 abstract2*
```

then, based on the rule for some_concrete_rule, concrete1 occurs in the context (concrete1,concrete1,Single) - i.e., as a single instance of itself, concrete2 occurs in the context (concrete2,concrete2,List), i.e. as a list of instances of itself. The use of the abstract1 class leads to a number of contexts:

```
(abstract1,abstract1,Single)
(concrete3,abstract1,Single)
(concrete4,abstract1,Single)
```

And finally, the use of abstract2* yields to the contexts

```
(abstract2,abstract2,List)
(concrete5,abstract2,List)
(concrete6,abstract2,List)
```

These contexts essentially mean that an instance of concrete5 can be replaced by any number of any (concrete) instance of "abstract2".

Reducing Contexts

If there are two or more conflicting contexts for a single symbol, we must resolve the contexts to their most specific (restrictive) form. For instance, for the phc grammar, this yields

```
(if,statement,List)
(CLASS_NAME,CLASS_NAME,Single)
(INTERFACE_NAME,INTERFACE_NAME,Single)
```

So, a context is a triplet (symbol,symbol,multiplicity), where the symbols are terminal or non-terminal symbols, and the multiplicity is either Single, Optional, List, OptionalList or ListOptional (list of optionals). When reducing two contexts (a,b,c) (`a'`,`b'`,`c'`), we take the meet of `b` and `b'` (that is,

the most general common subclass of `b` and `b'`, where more general means higher up in the inheritance hierarchy), and opt for the most restrictive Multiplicity (Single over Optional, Single over List, etc.). The general idea is that we want the most permissive context for a non-terminal that is still safe: if it is safe to replace an `a` by a list of `b`s *everywhere* in a tree, the context we want for `a` is (`a`, `b`, list).

To see the reason for taking the meet, consider this fragment of the phc grammar:

```
expr ::= ... | BOOL
cast ::= CAST expr
method_invocation ::= target ...
target ::= expr | CLASS_NAME
```

The use of "expr" in the rule for cast leads to the context (BOOL,expr,Single) The use of "target" in the rule for method_invocation leads to the context (BOOL,target,Single). By taking the meet of "expr" and "target", this gives the context (BOOL,expr,Single). This means that it is always safe to replace a boolean by any other expression (but it is not always safe to replace a boolean by any other *target*).

In the case of CLASS_NAME, we have the contexts

```
(CLASS_NAME,class_name,Single)
(CLASS_NAME,target,Single)
```

The meet of class_name and target does not exist; hence this gives the context

```
(CLASS_NAME,CLASS_NAME,Single)
```

That is, the only safe transformation for CLASS_NAME is from CLASS_NAME to CLASS_NAME.

To be precise about the “most specific” multiplicity, here is a Haskell definition that returns the meet of two multiplicities:

```
meet_mult :: Multiplicity -> Multiplicity -> Multiplicity
meet_mult a b | a == b = a
meet_mult Single _ = Single
meet_mult List Optional = Single
meet_mult List OptList = List
meet_mult List ListOpt = List
meet_mult Optional OptList = Single
meet_mult Optional ListOpt = Optional
meet_mult OptList ListOpt = List
meet_mult a b = meet_mult b a  -- meet is commutative
```

Resolution for Disjunctions

We cannot deal with this situation:

```
s ::= a
a ::= b | c
d ::= b
e ::= c*
```

This grammar leads to the following contexts:

```
(a,a,Single)
(b,a,Single)
(b,b,Single)
```

```
(c,a,Single)
(c,c,List)
```

Resolving these contexts lead to

```
(a,a,Single)
(b,b,Single)
(c,c,List)
```

However, this is incorrect, because this indicates that an `a` will only be replaced by another, single, `a`; but a `c` (which is an `a`) will in fact return a list of `c`s. The problem is that the non-terminals in the rule for `a` have a different multiplicity in their contexts (single for `b`, list for `c`). maketea disallows this; if this happens in a grammar, maketea will exit with a "cannot deal with mixed multiplicity in disjunction" error.

Otherwise, for a rule `a ::= b1 | b2 | ...`, if the multiplicity of `a` is list, and the multiplicities of all the `b`s are lists, the multiplicity for `a` will be list; if the multiplicity of all the `b`s is single, the multiplicity for `a` will be set to single (independent of the original multiplicity for `a`).

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Memory Layout for Multiple and Virtual Inheritance (By Edsko de Vries, January 2006)

Warning. This article is rather technical and assumes a good knowledge of C++ and some assembly language.
In this article we explain the object layout implemented by `gcc` for multiple and virtual inheritance. Although in an ideal world C++ programmers should not need to know these details of the compiler internals, unfortunately the way multiple (and especially virtual) inheritance is implemented has various non-obvious consequences for writing C++ code (in particular, for downcasting pointers, using pointers to pointers, and the invocation order of constructors for virtual bases). If you understand how multiple inheritance is implemented, you will be able anticipate these consequences and deal with them in your code. Also, it is useful to understand the cost of using virtual inheritance if you care about efficiency. Finally, it is interesting :-)

Multiple Inheritance

First we consider the relatively simple case of (non-virtual) multiple inheritance. Consider the following C++ class hierarchy.

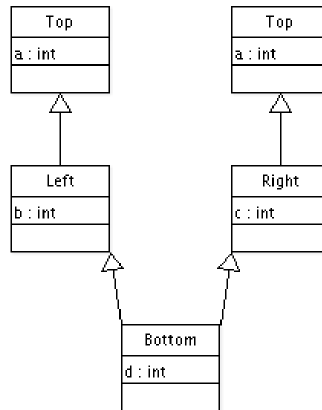
```
class Top
{
public:
   int a;
};

class Left : public Top
{
public:
   int b;
};

class Right : public Top
{
public:
   int c;
};

class Bottom : public Left, public Right
{
public:
   int d;
};
```

Using a UML diagram, we can represent this hierarchy as



Note that `Top` is inherited from *twice* (this is known as *repeated inheritance* in Eiffel). This means that an object `bottom` of type `Bottom` will have *two* attributes called `a` (accessed as `bottom.Left::a` and `bottom.Right::a`).

How are `Left`, `Right` and `Bottom` laid out in memory? We show the simplest case first. `Left` and `Right` have the following structure:

Left	**Right**
Top::a	Top::a
Left::b	Right::c

Note that the first attribute is the attribute inherited from `Top`. This means that after the following two assignments

```
Left* left = new Left();
Top* top = left;
```

`left` and `top` can point to the exact same address, and we can treat the `Left` object as if it were a `Top` object (and obviously a similar thing happens for `Right`). What about `Bottom`? `gcc` suggests

Bottom
Left::Top::a
Left::b
Right::Top::a
Right::c
Bottom::d

Now what happens when we upcast a `Bottom` pointer?

```
Bottom* bottom = new Bottom();
Left* left = bottom;
```

This works out nicely. Because of the memory layout, we can treat an object of type `Bottom` as if it were an object of type `Left`, because the memory layout of both classes coincide. However, what happens when we upcast to `Right`?

```
Right* right = bottom;
```

For this to work, we have to adjust the pointer value to make it point to the corresponding section of the `Bottom` layout:

	Bottom
	Left::Top::a
	Left::b
`right` ⟶	Right::Top::a
	Right::c
	Bottom::d

After this adjustment, we can access `bottom` through the `right` pointer as a normal `Right` object; however, `bottom` and `right` now point to *different* memory locations. For completeness' sake, consider what would happen when we do

```
Top* top = bottom;
```

Right, nothing at all. This statement is ambiguous: the compiler will complain

```
error: `Top' is an ambiguous base of `Bottom'
```

The two possibilities can be disambiguated using

```
Top* topL = (Left*) bottom;
Top* topR = (Right*) bottom;
```

After these two assignments, `topL` and `left` will point to the same address, as will `topR` and `right`.

Virtual Inheritance

To avoid the repeated inheritance of `Top`, we must inherit *virtually* from `Top`:

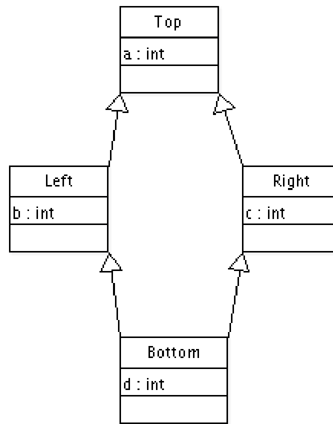
```
class Top
{
public:
   int a;
};

class Left : virtual public Top
{
public:
   int b;
};

class Right : virtual public Top
{
public:
   int c;
};

class Bottom : public Left, public Right
{
public:
   int d;
};
```

This yields the following hierarchy (which is perhaps what you expected in the first place)



while this may seem more obvious and simpler from a programmer's point of view, from the compiler's point of view, this is vastly more complicated. Consider the layout of `Bottom` again. One (non) possibility is

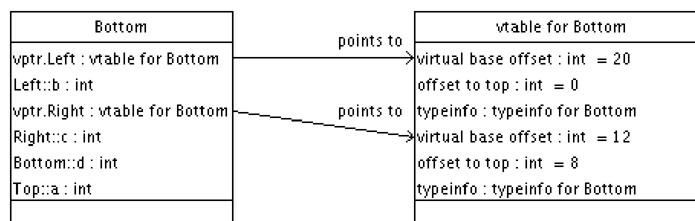
Bottom
Left::Top::a
Left::b
Right::c
Bottom::d

The advantage of this layout is that the first part of the layout collides with the layout of `Left`, and we can thus access a `Bottom` easily through a `Left` pointer. However, what are we going to do with

```
Right* right = bottom;
```

Which address do we assign to `right`? After this assignment, we should be able to use `right` as if it were pointing to a regular `Right` object. However, this is impossible! The memory layout of `Right` itself is completely different, and we can thus no longer access a “real” `Right` object in the same way as an upcasted `Bottom` object. Moreover, no other (simple) layout for `Bottom` will work.

The solution is non-trivial. We will show the solution first and then explain it.



You should note two things in this diagram. First, the order of the fields is completely different (in fact, it is approximately the reverse). Second, there are these new `vptr` pointers. These attributes are automatically inserted by the compiler when necessary (when using virtual inheritance, or when using virtual functions). The compiler also inserts code into the constructor to initialise these pointers.

The `vptrs` (virtual pointers) index a “virtual table”. There is a `vptr` for every virtual base of the class. To see how the virtual table (**vtable**) is used, consider the following C++ code.

```
Bottom* bottom = new Bottom();
Left* left = bottom;
int p = left->a;
```

The second assignment makes `left` point to the *same address* as `bottom` (i.e., it points to the “top” of the `Bottom` object). We consider the compilation of the last assignment (slightly simplified):

```
movl   left, %eax         # %eax = left
movl   (%eax), %eax       # %eax = left.vptr.Left
movl   (%eax), %eax       # %eax = virtual base offset
addl   left, %eax         # %eax = left + virtual base offset
movl   (%eax), %eax       # %eax = left.a
movl   %eax, p            # p = left.a
```

In words, we use `left` to index the virtual table and obtain the “virtual base offset” (**vbase**). This offset is then added to `left`, which is then used to index the `Top` section of the `Bottom` object. From the diagram, you can see that the virtual base offset for `Left` is 20; if you assume that all the fields in `Bottom` are 4 bytes, you will see that adding 20 bytes to `left` will indeed point to the `a` field.

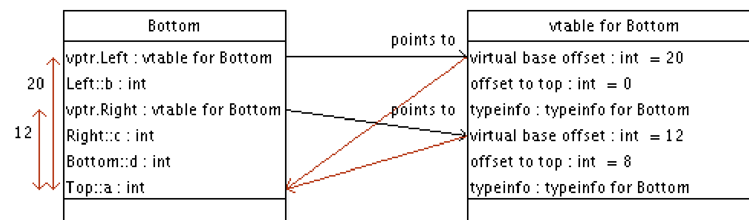
With this setup, we can access the `Right` part the same way. After

```
Bottom* bottom = new Bottom();
Right* right = bottom;
int p = right->a;
```

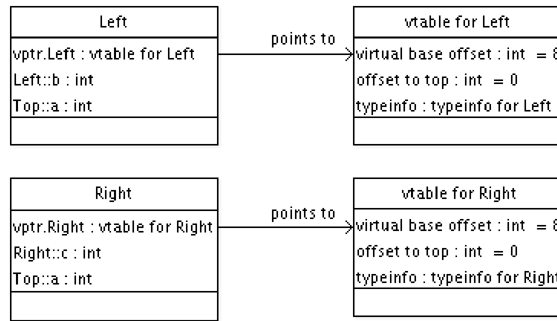
`right` will point to the appropriate part of the `Bottom` object:

	Bottom
	vptr.Left
	Left::b
`right` →	vptr.Right
	Right::c
	Bottom::d
	Top::a

The assignment to `p` can now be compiled in the *exact same way* as we did previously for `Left`. The only difference is that the `vptr` we access now points to a different part of the virtual table: the virtual base offset we obtain is 12, which is correct (verify!). We can summarise this visually:



Of course, the point of the exercise was to be able to access real `Right` objects the same way as upcasted `Bottom` objects. So, we have to introduce `vptrs` in the layout of `Right` (and `Left`) too:



Now we can access a `Bottom` object through a `Right` pointer without further difficulty. However, this has come at rather large expense: we needed to introduce virtual tables, classes needed to be extended with one or more virtual pointers, and a simple attribute lookup in an object now needs two indirections through the virtual table (although compiler optimizations can reduce that cost somewhat).

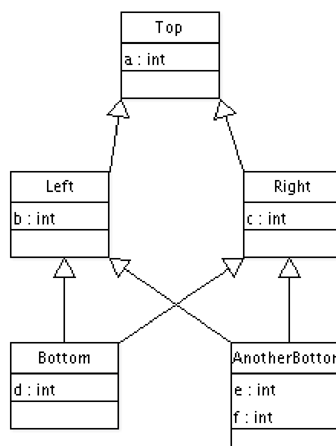
Downcasting

As we have seen, casting a pointer of type `DerivedClass` to a pointer of type `SuperClass` (in other words, upcasting) may involve adding an offset to the pointer. One might be tempted to think that downcasting (going the other way) can then simply be implemented by subtracting the same offset. And indeed, this is the case for non-virtual inheritance. However, virtual inheritance (unsurprisingly!) introduces another complication.

Suppose we extend our inheritance hierarchy with the following class.

```
class AnotherBottom : public Left, public Right
{
public:
   int e;
   int f;
};
```

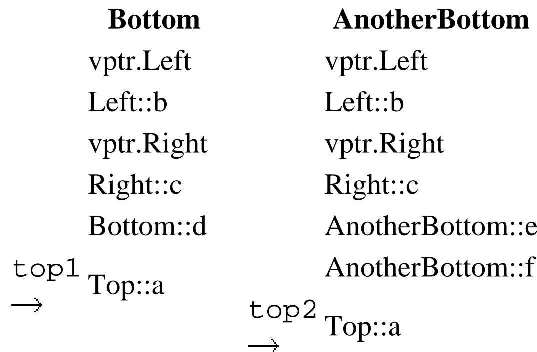
The hierarchy now looks like



Now consider the following code.

```
Bottom* bottom1 = new Bottom();
AnotherBottom* bottom2 = new AnotherBottom();
Top* top1 = bottom1;
Top* top2 = bottom2;
Left* left = static_cast<Left*>(top1);
```

The following diagram shows the layout of `Bottom` and `AnotherBottom`, and shows where `top` is pointing after the last assignment.



Now consider how to implement the *static* cast from `top1` to `left`, while taking into account that we do not know whether `top1` is pointing to an object of type `Bottom` or an object of type `AnotherBottom`. It can't be done! The necessary offset depends on the runtime type of `top1` (20 for `Bottom` and 24 for `AnotherBottom`). The compiler will complain:

```
error: cannot convert from base `Top' to derived type `Left'
via virtual base `Top'
```

Since we need runtime information, we need to use a dynamic cast instead:

```
Left* left = dynamic_cast<Left*>(top1);
```

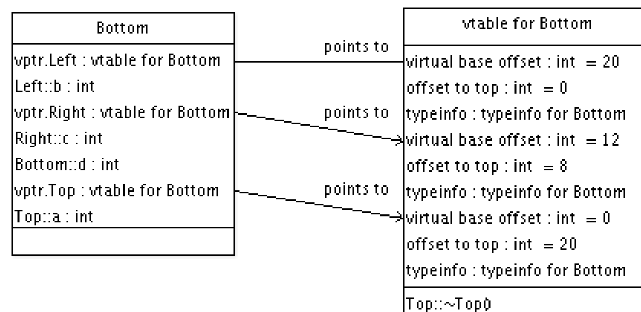
However, the compiler is still unhappy:

```
error: cannot dynamic_cast `top' (of type `class Top*') to type
   `class Left*' (source type is not polymorphic)
```

The problem is that a dynamic cast (as well as use of `typeid`) needs runtime type information about the object pointed to by `top1`. However, if you look at the diagram, you will see that all we have at the location pointed to by `top1` is an integer (`a`). The compiler did not include a `vptr.Top` because it did not think that was necessary. To force the compiler to include this `vptr`, we can add a virtual destructor to `Top`:

```
class Top
{
public:
   virtual ~Top() {}
   int a;
};
```

This change necessitates a `vptr` for `Top`. The new layout for `Bottom` is



(Of course, the other classes get a similar new `vptr.Top` attribute). The compiler now inserts a library call for the dynamic cast:

```
left = __dynamic_cast(top1, typeinfo_for_Top, typeinfo_for_Left, -1);
```

This function `__dynamic_cast` is defined in `libstdc++` (the corresponding header file is `cxxabi.h`); armed with the type information for `Top`, `Left` and `Bottom` (through `vptr.Top`), the cast can be executed. (The -1 parameter indicates that the relationship between `Left` and `Top` is presently unknown). For details, refer to the implementation in tinfo.cc.

Concluding Remarks

Finally, we tie a couple of loose ends.

(In)variance of Double Pointers

This is were it gets slightly confusing, although it is rather obvious when you give it some thought. We consider an example. Assume the class hierarchy presented in the last section (Downcasting). We have seen previously what the effect is of

```
Bottom* b = new Bottom();
Right* r = b;
```

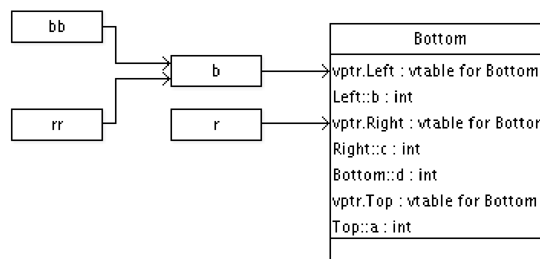
(the value of `b` gets adjusted by 8 bytes before it is assigned to `r`, so that it points to the `Right` section of the `Bottom` object). Thus, we can legally assign a `Bottom*` to a `Right*`. What about `Bottom**` and `Right**`?

```
Bottom** bb = &b;
Right** rr = bb;
```

Should the compiler accept this? A quick test will show that the compiler will complain:

```
error: invalid conversion from `Bottom**' to `Right**'
```

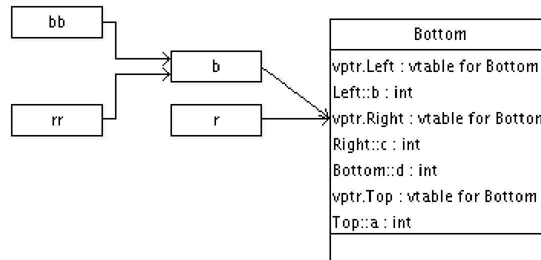
Why? Suppose the compiler would accept the assignment of `bb` to `rr`. We can visualise the result as:



So, `bb` and `rr` both point to `b`, and `b` and `r` point to the appropriate sections of the `Bottom` object. Now consider what happens when we assign to `*rr` (note that the type of `*rr` is `Right*`, so this assignment is valid):

```
*rr = b;
```

This is essentially the same assignment as the assignment to `r` above. Thus, the compiler will implement it the same way! In particular, it will adjust the value of `b` by 8 bytes before it assigns it to `*rr`. But `*rr` pointed to `b`! If we visualise the result again:



This is correct as long as we access the `Bottom` object through `*rr`, but as soon as we access it through `b` itself, all memory references will be off by 8 bytes — obviously a very undesirable situation.

So, in summary, even if `*a` and `*b` are related by some subtyping relation, `**a` and `**b` are *not*.

Constructors of Virtual Bases

The compiler must guarantee that all virtual pointers of an object are properly initialised. In particular, it guarantees that the constructor for all virtual bases of a class get invoked, and get invoked only once. If you don't explicitly call the constructors of your virtual superclasses (independent of how far up the tree they are), the compiler will automatically insert a call to their default constructors.

This can lead to some unexpected results. Consider the same class hierarchy again we have been considering so far, extended with constructors:

```
class Top
{
public:
   Top() { a = -1; }
   Top(int _a) { a = _a; }
   int a;
};

class Left : public Top
{
public:
   Left() { b = -2; }
   Left(int _a, int _b) : Top(_a) { b = _b; }
   int b;
};

class Right : public Top
{
public:
   Right() { c = -3; }
   Right(int _a, int _c) : Top(_a) { c = _c; }
   int c;
};

class Bottom : public Left, public Right
{
public:
   Bottom() { d = -4; }
   Bottom(int _a, int _b, int _c, int _d) : Left(_a, _b), Right(_a, _c)
        {
      d = _d;
        }
   int d;
};
```

(We consider the non-virtual case first.) What would you expect this to output:

```
Bottom bottom(1,2,3,4);
printf("%d %d %d %d %d\n", bottom.Left::a, bottom.Right::a,
   bottom.b, bottom.c, bottom.d);
```

You would probably expect (and get)

```
1 1 2 3 4
```

However, now consider the virtual case (where we inherit virtually from `Top`). If we make that single change, and run the program again, we instead get

```
-1 -1 2 3 4
```

Why? If you trace the execution of the constructors, you will find

```
Top::Top()
Left::Left(1,2)
Right::Right(1,3)
Bottom::Bottom(1,2,3,4)
```

As explained above, the compiler has inserted a call to the default constructor in `Bottom`, before the execution of the other constructors. Then when `Left` tries to call its superconstructor (`Top`), we find that `Top` has already been initialised and the constructor does not get invoked.

To avoid this situation, you should explicitly call the constructor of your virtual base(s):

```
Bottom(int _a, int _b, int _c, int _d): Top(_a), Left(_a,_b), Right(_a,_c)
{
   d = _d;
}
```

Pointer Equivalence

Once again assuming the same (virtual) class hierarchy, would you expect this to print “Equal”?

```
Bottom* b = new Bottom();
Right* r = b;

if(r == b)
   printf("Equal!\n");
```

Bear in mind that the two addresses are not *actually* equal (`r` is off by 8 bytes). However, that should be completely transparent to the user; so, the compiler actually subtracts the 8 bytes from `r` before comparing it to `b`; thus, the two addresses are considered equal.

Casting to `void*`

Finally, we consider what happens we can cast an object to `void*`. The compiler must guarantee that a pointer cast to `void*` points to the “top” of the object. Using the vtable, this is actually very easy to implement. You may have been wondering what the *offset to top* field is. It is the offset from the `vptr` to the top of the object. So, a cast to `void*` can be implemented using a single lookup in the vtable.

References

[1] CodeSourcery, in particular the C++ ABI Summary, the Itanium C++ ABI (despite the name, this document is referenced in a platform-independent context; in particular, the structure of the vtables is detailed here). The `libstdc++` implementation of dynamic casts, as well RTTI and name unmangling/demangling, is defined in tinfo.cc.

[2] The libstdc++ website, in particular the section on the C++ Standard Library API.

[3] C++: Under the Hood by Jan Gray.

[4] Chapter 9, “Multiple Inheritance” of *Thinking in C++ (volume 2)* by Bruce Eckel. The author has made this book available for download.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Writing a Reentrant Parser with Flex and Bison (By Edsko de Vries, August 2006)

This article explains how to create a reentrant parser with Flex and Bison, and how to include more than one parser in the same application. It is *not* suitable as an introduction to either Flex or Bison; we assume the reader is familiar with both tools, as well as with C and C++.

The problem we will set ourselves is to write a processor for the language **ABCD**. **ABCD** is a small toy language designed specifically for this tutorial. It is made up of two sub-languages, **AB** and **CD**; here is a simple example in language **AB**:

```
abbab
```

Each “program” in **ABCD** evaluates to an integer. The first occurrence of an `a` has value 1, the second occurrence has value 2, etc., and similarly for `b`s. Thus; the example above has value 9. In addition, you can “escape” to language **CD** using square brackets `[...]`:

```
a[cdd]bb[c]a
```

The value of a program in language `CD` is calculated analogously to the value of a program in language `AB`: the first `c` has value 1, the second `c` has value 2, etc., and similarly for `d`. Moreover, values are always calculated with respect to the enclosing square brackets (so, the value of the second example is 11).

Finally, in language **CD**, you can escape back to language **AB**, so you can nest either language in the other, creating arbitrarily complex nested strings. For example,

```
a[cd[a[d]]d]b[c[[cd]]]
```

also has value 11. The point of the exercise is that languages **AB** and **CD** will each get their own parser, so we will need to combine two parsers into one application. The parsers will need global state (in addition to the parser's internal bookkeeping) to be able to calculate the value of each character. Moreover, since it is possible to have an **AB** string inside another **AB** string (by escaping twice), we may have to instantiate a new **AB** parser while the “old” one is still active. Therefore it is important that the parsers are reentrant.

The code for the application we will develop in this tutorial can be found in reentrantparser.tar.gz.

High Level Overview

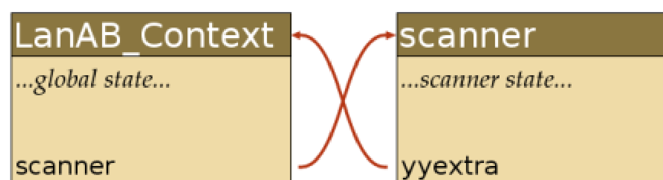
Before we start looking at the details, we will give a high level overview of the solution first. There is more than one way to solve this problem; the method I will present here is the one I believe is the least messy, but that is of course a matter of opinion; one alternative (using a C++ lexer) is discussed briefly at the end of this tutorial.

Unfortunately, although the solutions offered by Flex and Bison are very similar, they are slightly different in the details, so it will be important to remember which tool we are talking about.

When Flex generates a reentrant scanner, the function `yylex` will get an additional argument `scanner`. This argument is a pointer to a data structure that represents the state of the scanner. Before we start parsing, we must initialise this state, and then pass the state in to `yylex` every time it is invoked. The scanner state has a data field, called `yyextra`, of a user-specified type, that can be used for additional state. We will use `yyextra` to determine the semantic value of each character in the input (according to the rules explained in the introduction).

The Bison generated parser `yyparse` also gets an additional argument, but this argument represents the user-defined state only. The Bison internal global state is stored in local variables inside `yyparse`, and is completely invisible to the user.

We will create a class `LanAB_Context` (for “language **AB** context”) to hold the global (user-defined) state. We will pass in an object of type `LanAB_Context` to `yyparse`. Since `yyparse` needs to call `yylex`, we will find it useful to store a reference to the `scanner` object inside `LanAB_Context`. However, since we only have access to the `scanner` object from within `yylex`, we will use `yyextra` (mentioned above) inside the `scanner` object to point back to the `LanAB_Context` object. Graphically:



The Parser Context

The parser context will be represented by the following class:

```
#ifndef LANAB_CONTEXT
#define LANAB_CONTEXT

#include <iostream>
using namespace std;

class LanAB_Context
{
public:
   void* scanner;    // the scanner state
   int result;       // result of the program
   int a;            // value of the next a
   int b;            // value of the next b
   istream* is;      // input stream
   int esc_depth;    // escaping depth

public:
   LanAB_Context(istream* is = &cin)
   {
      init_scanner();
      this->is = is;
      a = 1;
      b = 1;
   }

   virtual ~LanAB_Context()
   {
```

```
      destroy_scanner();
   }

// Defined in LanAB.l
protected:
   void init_scanner();
   void destroy_scanner();
};

int LanAB_parse(LanAB_Context*);

#endif // LANAB_CONTEXT
```

The first section of the class lists the variables that make up the user state of the parser. Most of the variables will be self-explanatory, with the exception perhaps of `is` and `esc_depth`, which we will explain when we discuss the lexical analyser.

The constructor of `LanAB_Context` initialises some of the parser state, and calls `init_scanner`. The bodies for `init_scanner` and `destroy_scanner` will be provided in the Flex file, and will call `yylex_init` and `yylex_destroy` to initialise and free the scanner state, respectively.

The Lexer

We will explain the code for the lexer bit by bit. First of all, we need to tell Flex to create a reentrant parser:

```
%option reentrant
```

Since we will need two lexers in our application, they cannot both be called `yylex`. Hence, we set the prefix to “LanAB_” so that the scanner will be called `LanAB_lex`:

```
%option prefix="LanAB_"
```

The next two options tell Flex that we are interfacing with a Bison generated parser; `bison-bridge` adds an argument `yylval` to `yylex`, and `bison-locations` adds an argument code `yylloc` for location tracking.

```
%option bison-bridge
%option bison-locations
```

There is a bug in Flex in the code for `yywrap`, but since we don't need `yywrap` at all, we simply disable it.

```
%option noyywrap
```

We will use Flex's built-in support for line numbers:

```
%option yylineno
```

Next we need a bit of C code. First, we need to include the header file that defines the parser context, and the header file generated by Bison (for the token identifiers).

```
%{
   #include "LanAB_Context.h"
   #include "LanAB.tab.h"
```

As mentioned in the overview, the scanner state will include a field called `yyextra` that can be used for user-defined state. The type of this field is specified by `YY_EXTRA_TYPE`:

```
#define YY_EXTRA_TYPE LanAB_Context*
```

To set line numbers, we set `yyloc->first_line` to `yylineno` each time a token is recognised:

```
#define YY_USER_ACTION yylloc->first_line = yylineno;
```

Finally, because we need to parse strings as well as files, we redefine `YY_INPUT` and give it a C++ flavour. It will use the `istream` from the parser context to read the next character. The parser context defaults `is` to `cin`, but we will set it to an `istringstream` later on to parse a nested program.

```
   #define YY_INPUT(buf,result,max_size)   \
   {                                       \
      char c;                              \
      (*yyextra->is) >> c;                 \
      if(yyextra->is->eof())               \
         result = YY_NULL;                 \
      else {                               \
         buf[0] = c;                       \
         result = 1;                       \
      }                                    \
   }
%}
```

We define an exclusive scanner state `ESC` to deal with escaping:

```
%x ESC
```

We can now define the parser proper. When we see an “a” we use the parser state to determine its semantic value (see the discussion of the parser, below, for a description of `yylval`), and return token `A` to the parser (and similarly for “b”). When we see an open square bracket, we set the ”escape depth” to 1 and set the scanner state to `ESC`.

```
%%

"a"         yylval->integer = yyextra->a++; return A;
"b"         yylval->integer = yyextra->b++; return B;
"["         yyextra->esc_depth = 1; BEGIN(ESC);
.           return ERR;
\n          /* ignore */
```

In `ESC`, we increase the escape depth for every open square bracket we see, and decrease it for every close bracket. When the depth reaches 0, we return an `ESCAPE` token to the parser with the appropriate semantic value, and reset the scanner state.

```
<ESC>"]"    %{
               yyextra->esc_depth--;
               if(yyextra->esc_depth == 0)
               {
                  yylval->cptr = strndup(yytext, yyleng-1);
                  BEGIN(INITIAL);
                  return ESCAPE;
               }
               else
               {
                  yymore();
               }
```

```
            %}
<ESC>"["    yymore(); yyextra->esc_depth++;;
<ESC>.      yymore();
```

In the scanner epilogue we provide the bodies for `init_scanner` and `destroy_scanner`. Note the call to `yyset_extra` to initialise the `yyextra` field.

```
%%

void LanAB_Context::init_scanner()
{
   yylex_init(&scanner);
   yyset_extra(this, scanner);
}

void LanAB_Context::destroy_scanner()
{
   yylex_destroy(scanner);
}
```

The Parser

The start of the definition of the parser is nearly identical to the start of the lexical analyser. We tell Bison that we need a reentrant parser, and that the prefix should be changed from `yy` to `LanCD_`:

```
%pure-parser
%name-prefix="LanAB_"
```

Next we need to set a few configuration options to enable location tracking, the generation of a header file, and verbose error messages:

```
%locations
%defines
%error-verbose
```

We tell Bison that `yyparse` should take an extra parameter `context`, and that `yylex` (`LanAB_lex`) takes an additional argument `scanner` (see below for an explanation of how Bison knows which value to use for `scanner`).

```
%parse-param { LanAB_Context* context }
%lex-param { void* scanner  }
```

We use Bison's `%union` construct to define a semantic value type for integers and character pointers (strings). The terminal symbols `A` and `B`, as well as the rule `lanab` all have type `integer`; only the `ESCAPE` token has type `cptr` (string):

```
%union
{
   int integer;
   char* cptr;
}

%token <integer> A
%token <integer> B
%token <cptr> ESCAPE
%token ERR

%type <integer> lanab
```

As in the lexer, we need a bit of C code in the prologue. Note that this C code is defined *after* the definition of the `%union`, which means that this code will go into the C++ file (`LanAB.tab.c`) instead of the header file (`LanAB.tab.h`). We need a few system headers, and we need to include the parser context headers `LanAB_Context` and `LanCD_Context` (we have not shown `LanCD_Context` above but it is virtually the same as `LanAB_Context`; the full code can be found in the source archive):

```
%{
   #include <iostream>
   #include <sstream>
   #include "LanAB_Context.h"
   #include "LanCD_Context.h"

   using namespace std;
```

We need to declare the type of the lexer:

```
   int LanAB_lex(YYSTYPE* lvalp, YYLTYPE* llocp, void* scanner);
```

and define the error handler. Note that the error handler is passed the parser context and location information; these parameters are automatically added to the error handler when creating a pure (reentrant) parser.

```
   void LanAB_error(YYLTYPE* locp, LanAB_Context* context, const char* err)
   {
      cout << locp->first_line << ":" << err << endl;
   }
```

We told Bison above that `yylex` takes an additional argument `scanner` of type `void*`, but we haven't yet told it which value to use for `scanner`. The way this works is that Bison will use the name of the argument also as the value of the argument; in other words, it will call `yylex` as `yylex(..., scanner)`. Therefore, we provide a macro `scanner` that extracts the scanner state from the parser state:

```
   #define scanner context->scanner
%}
```

The definition of the grammar itself is reasonably straightforward. We will show the entire grammar and then explain some details:

```
%%

start:
     lanab
        { context->result = $1; }
   ;

lanab:
     A lanab
        { $$ = $1 + $2; }
   | B lanab
       { $$ = $1 + $2; }
   | ESCAPE lanab
       {
          {
             istringstream* is = new istringstream($1);
             LanCD_Context context(is);
             LanCD_parse(&context);
             $$ = context.result + $2;
          }
```

```
      }
   | /* empty */
      { $$ = 0; }
   ;
```

The only rule, really, that needs an explanation in this definition is the rule that deals with escapes. When we get an escape string, we set up a brand new parser context for the *other* parser (for the **CD** language). We create a new `istringstream` based on the escape string to act as the input for the new parser (this is why we redefined `YY_INPUT` in the lexer) and invoke the parser. The result of the parser (as extracted from its context) is used as the semantic value of the escape string.

The Main Application

The main application is very straightforward:

```
#include <iostream>
#include "LanAB_Context.h"

using namespace std;

int main()
{
   LanAB_Context context;

   if(!LanAB_parse(&context))
   {
      cout << context.result << endl;
   }
}
```

Of course, for the application to be complete, we also need to define the parser context `LanCD_Context.h`, lexer `LanCD.l` and scanner `LanCD.y` for the **CD** language, but they are very similar to their `AB` counterparts. You can find the full application in reentrantparser.tar.gz.

Alternative: A C++ Lexer

As an alternative to `%option reentrant`, one can also specify `%option c++` in the Flex definition. This encapsulates the generated scanner in a class called `yyFlexLexer`, which makes it automatically reentrant. However, in my opinion this leads to inelegant code.

`yylex` can no longer be redefined to take additional parameters. To give the scanner access to a (user-defined) state, you need to make `yyFlexLexer` inherit from the parser context. To be more precise, you need to define a new class `Lexer` that inherits from both `yyFlexLexer` and `LanAB_Context`, then use the `%option yyclass` to tell Flex to add the `yylex` method to your `Lexer` class instead of to `yyFlexLexer`. You must then also add some glue code to add `yylval` and `yylloc` to the parser context in such a way that Bison knows how to access them. Not only is this rather messy, it also means that the scanner has a completely different method of accessing the context (through inheritance) than the parser (through additional parameters), which does little to improve the readability of the code.

Things really get messy when you need more than one (C++) scanner in the one application. It can be done, but it isn't very easy and certainly not very obvious. Moreover, even the authors of Flex aren't very confident; the regenerated C++ code contains the following comment (Flex version 2.5.33):

```
/* The c++ scanner is a mess. The FlexLexer.h header file relies on the
 * following macro. This is required in order to pass the
 * c++-multiple-scanners test in the regression suite. We get reports
```

```
 * that it breaks inheritance.  We will address this in a future release
 * of flex, or omit the C++ scanner altogether.
 */
#define yyFlexLexer yyFlexLexer
```

which doesn't inspire much confidence. All in all, I believe the solution as presented in this tutorial is better, but if you think otherwise, please let me know.

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Plugins

The plugins on this page are “3rdparty” plugins contributed by others. They are distributed in source form, and must be compiled using the `phc_compile_plugin` script distributed with phc. If you want your own plugin to be added to this list, please contact us on the mailing list.

`compact_echo`
Daniel Barreiro

download

A simple example of a tree transformation plugin

This PHP-compiler plugin joins together several consecutive echos into a single one and if any of the parameters to an echo is a concatenation of strings it splits them and outputs them separately. This is much faster than first concatenating strings and then outputing the whole thing. For example, this:

```
<?php
   echo "1" . (1 . $a ), 'ab';
   echo 'c' ,2;
   if ($a) {
      echo 2;
      //$a=1;
      echo $a;
   }
?>
```

is turned into this:

```
<?php
   echo "1", 1, $a, "abc", 2;
   if($a)
   {
      echo 2, $a;
   }
?>
```

See comments in the source.

License: whichever applies to the PHP Compiler

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Spinoff Projects

We are delighted to announce the first actual spin-off from phc. Written by Daniel Barreiro, `PHT` embeds HTML/XML into PHP to ensure that the XML or HTML output by PHP scripts is well-formed and (although this has not yet been implemented) valid. Details and source can be found at http://www.satyam.com.ar/pht/.

Suggestions

Below is a list of other projects that we'd love to see built. All of these tools are very difficult to implement starting from scratch. With phc as a base from which to work, the burden of developing any of these tools should be greatly reduced.

- Refactoring Tool
- Style Checker
- Aspect Weaver
- Script Obfuscator
- Script Optimizer
- Pretty Printers
- Language Translation
- Semantic Checker

Refactoring Tool

According to www.refactoring.com, refactoring is *a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior*. A simple example of refactoring is renaming a variable name or a function name. More complex refactoring may involve finding repeated blocks of code inside a script and moving them to a function.

While simple refactorings can be coded directly in phc (see for example the Demo), a dedicated refactoring tool based on phc would make this process much easier for the end-user. It could even provide a graphical user interface offering common refactoring tasks, so that the programmer does not need to write any code at all to refactor their code.

We are planning to include type inference in phc at some point, which should increase the usefulness of phc for refactoring greatly (see What's in Store?).

Make Yourself Known!

If you are interested in starting on any of these (or other) projects, and you need help, do not hesitate to send questions to the mailing list.

Also, if your tool has reached some level of usability, and you would like to see it listed on this website, please let us know. In fact, if your tool is real good, we could even integrate it into phc itself.

Either way, if you are building software based on phc, we would really love to know about it — so please keep us posted!

Donations

Finally, if you do write a tool based on phc, and you're making buckets of money, we'd appreciate a donation :-) Contact us at donations@phpcompiler.org.

See also Tutorial 2 for an example of a useful refactoring for PHP.

Style Checker

Software companies often have a series of “programming style guidelines” that dictate how classes, methods and variables should be named, if and where there are compulsory comments, etc. This is useful to make sure that code written by different programmers looks the same. This is also useful in open source projects, with lots of programmers working on the same codebase.

A style checker is a tool that knows about these guidelines and is able to check whether a particular program adheres to them. It is possible to write a style checker for a particular set of guidelines directly. However, a dedicated tool for style checking should be able to read in a set of guidelines (either in text form or using a GUI) and verify programs according to them.

Note that phc records line number and comments in the generated tree, so that they can be used in the verification process too.

Aspect Weaver

Aspect oriented programming is a relatively new programming paradigm. The standard example to explain what AOP does for you is the following. Suppose you have a script with a series of functions. Say you want to start and end every function with a call to some logging function:

```
<?php
   function some_function
   {
      log("some_function: begin");

      /* do whatever the function should do */

      log("some_function: end");
   }
?>
```

And say you want to do that in each and every function, for example for debugging purposes. It's a lot of work to do that manually for every function. And when you no longer need it, it is a lot of work to remove it again. This is what's known as a *cross-cutting concern*: something you need to implement (in this example, logging), that “cross-cuts“ (affects) a lot of code. With AOP, you can write this concern as a single “aspect”. You then run your program through an “aspect weaver”, which will insert the necessary code at the start and end of each function, thus saving you a lot of work.

For more information, check out AspectJ, an aspect weaver for Java (www.eclipse.org/aspectj), or read for example *AspectJ in Action* by Ramnivas Laddad. You can do some very slick things with aspect oriented programming :-)

phc would be a good foundation upon which to build an aspect weaver for PHP.

Script Obfuscator

Most PHP scripts are distributed in source form. But what if you don't want people to modify your code? Or what if you want to make your code as difficult to understand as possible, so that it becomes harder to analyse it for possible attacks? A script obfuscator takes a PHP script and outputs another PHP script that does the same thing, but is completely unreadable. There are a few obfuscators available for PHP, but a script obfuscator based on phc can make use of a lot of structural information about the script, opening new avenues for great obfuscation :-)

Script Optimizer

Similar to a script obfuscator, a script optimizer takes a PHP script and outputs another PHP script that does the same thing, but runs much faster. As a simple example, consider

```
<?php
   echo "Text $with variables.";
?>
```

This could be changed to

```
<?php
   echo 'Text ', $with, ' variables.';
?>
```

which should run faster. Many other optimizations are possible.

Pretty Printers

phc includes a standard pretty printer that outputs the AST (phc's internal representation of a script) into normal PHP, but it is rather limited. For example, it cannot be configured. Moreover, it can probably be improved in a number of places. For example, it really should deal better with brackets in expressions (at the moment, it simply copies the brackets from the user).

A pretty printer would take an AST and output it in some way. For example, it could output the AST to normal PHP code, but in a different layout style than the standard unparser does. But you could also think of other unparsers. For example, it might output to HTML or Latex, so that you can include PHP examples in HTML websites or in Latex documents.

Since version 0.1.6., the phc unparser is defined as a tree visitor, so it is possible to define new unparsers by inheriting from the `PHP_unparser` class, and redefining only those features that need to be improved.

Language Translation

Tools that translate PHP into other languages (for example, a PHP to ASP translator). Note that this is a much more difficult task than the other projects mentioned, and such a tool would benefit greatly from future additions to phc itself (which is, after all, a compiler).

Semantic Checker

Like a style checker, a semantic checker does not actually modify a PHP script, but checks it instead for “bugs”. For example, it could issue a warning in the following code

```
<?php
   $a = "some text " + $b;
?>
```

(The + should probably have been a `.`). This project will also benefit from future releases of phc (see What's in Store?).

$LastChangedDate: 2006-09-08 12:34:31 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.

Mailing List

We want phc we to be a useful project, so we have tried to write good documentation. However, there are bound to be numerous questions that we have left unanswered. If you have something you want to ask, or a bug to report, a comment to make, or just want to tell us something, please join the mailing list using the form on the right. We'll try to answer you as quickly as we can. If you're implementing a tool based on phc we'd love to know about it!

When you hit *Subscribe* you will be sent an email asking you to confirm the subscription request. If you confirm it (instructions are in the email), you will be sent a welcome email that contains a password and a link to a page that allows you change your options (for example your password). It will also tell you how to unsubscribe.

The archives of the mailing list are public and can be read online.

Join

Please use the form below to join the phc mailing list.

Email

$LastChangedDate: 2006-09-08 12:24:58 +0100 (Fri, 08 Sep 2006) $. Contents © the authors.