

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

What is phc?

phc is a compiler for PHP that will translate PHP code directly into Linux assembly code. It can be used as a (C++) framework for developing refactoring tools, aspect weavers, script obfuscators and any other tools that operate on PHP scripts. See [Spinoffs](#) for some suggestions.

At the moment, phc gives the programmer a nice representation of a PHP script (not unlike the DOM tree representation of an XML script), provides an interface for modifying this tree, and provides a way to output this tree back to normal PHP code. For a quick idea of what phc can do for you, check out the [Demo](#). To find out what is planned for the next release, read [What's in Store](#).

Note that in particular, the current release does not yet compile PHP. It is therefore not yet useful for end-users, but can be very useful for programmers wishing to implement tools for PHP. To get an impression of how to implement such tools using phc, check out [Getting Started](#).

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

9 January 2006. Lots of minor feature enhancements. See [CHANGES](#) for details. **IMPORTANT:** this release is not downwards compatible with the previous!

1 December 2005. Bugfix. Null vectors are now unparsed properly by the dot unparser, and Windows-style linebreaks 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. Bugfix. 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.

phc -- the open source PHP compiler

\$LastChangedDate: 2006-01-09 12:20:48 +0000 (Mon, 09 Jan 2006) \$. Contents © the authors.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

Authors

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

Acknowledgements

Dr. David Abrahamson is involved in the discussions about the design of phc. He also wrote the `RemoveConcatNull` transform and suggested its use for a tutorial (now [Tutorial 3](#)).

\$LastChangedDate: 2006-01-04 16:19:56 +0000 (Wed, 04 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

Documentation

We have tried to document phc as well as we can, but if anything is still unclear, please let us know by sending an email to the [mailing list](#). The documentation for phc divides into tutorials, reference documentation and internal documentation. The tutorials get you started without necessarily going into all the details. They should be read in the order that they are listed. The reference documentation is not meant as introductory material, but is meant as the definitive reference to phc and can be used to look things up. The internal documentation describes how phc is implemented and is of interest to programmers who wish to modify phc beyond the API we provide for doing so.

Tutorials

Demo

For a quick overview of what the current phc release can do for you, check out the [Demo](#). The demo runs through a simple refactoring example (renaming a function). It shows the source and target program, and the code to do the refactoring. It also shows how phc represents programs internally.

Getting Started

[Getting Started](#) explains the basic principles behind phc, in particular behind how phc represents PHP scripts internally. It then shows how to write a program 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 `mysql_connect` by calls to `dbx_connect`.

Reference

Installation

The [Installation Instructions](#) guide you through installing phc (a simple process).

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.

The Grammar Formalism

There are various styles of grammars. The style we have adopted is non-standard, but the [grammar formalism](#) explains it in detail and explains why we have adopted it. It also explains the mapping from the grammar to the C++ class structure.

Converting PHP

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

Limitations

Known [limitations](#) of the current implementation of phc.

Tutorial 3: Restructuring the Tree

[Tutorial 3](#) shows how you can modify the structure of the tree. It works through an example that replaces all method invocations on objects (`$a->foo()`) by code that checks whether the object is instantiated first.

Tutorial 4: Using State

[Tutorial 4](#) introduces a very important concept: 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.

What's in Store?

[What's in Store?](#) gives a quick preview of what the next release of phc will have to offer. It briefly explains what type inference is, and (as an example) shows how it can be used for refactoring.

\$LastChangedDate: 2006-01-04 19:58:54 +0000 (Wed, 04 Jan 2006) \$. Contents © the [authors](#).

Internal Documentation

Implementation of the Tree Traversal API

The [Implementation of the Tree Traversal API](#) describes the details of the implementation of the tree traversal API. It is of interest only to programmers wishing to understand or modify it. You do not need to read this document to be able to use the API.

Memory Layout for Multiple and Virtual Inheritance

[This article](#) describes how multiple and virtual inheritance are implemented in C++ (specifically, in `gcc`). This article is not specific to phc, but is of interest to all advanced C++ programmers.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff projects](#) | [Mailing List](#)

System Requirements

phc needs a Unix-like environment to run (it has been tested on Solaris and Linux). To compile phc, you need at least the following tools.

- C and C++ compiler (we have only tested with `gcc`)
- `make`
- `flex`
- `bison`
- `patch`
- `grep`

However, these tools should come pre-installed on most systems. In addition, you may need two other tools (which are less commonly pre-installed): If you want to add extra command line arguments to phc, you will need [`gengetopt`](#). If you are feeling adventurous and want to modify the actual parser, you may also need [`gperf`](#) to make the lexical analyser recognize extra keywords.

Finally, phc has some graphical output in the form of trees and graphs. These graphics use the “dot” format, and you will need something like [`graphviz`](#) to view them.

phc does not need any special libraries (other than the ones included with the distribution).

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.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/src
make
```

This should compile without any warnings or errors (there might be one warning about patching `php.tab.cpp` that can safely be ignored). 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.

Testing Your Installation

If everything went smoothly, you should now have a new binary called `phc` in the current directory. Run it by typing

```
./phc
```

You should see something like

```
phc revision 316 (2005/09/30)
```

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

```
-h, --help          Print help and exit
-V, --version       Print version and exit
--dump-tokens       Perform lexical analysis only (spits out a token
                    list). Probably only useful for debugging phc.
                    (default=off)
--dump-php          Dump PHP code back immediately after parsing to
                    standard output (pretty printing). (default=off)
--dump-ast          Dump the AST from the source in dot format.
                    (default=off)
```

Running phc

In this section, we very briefly show how to run `phc`. 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
/*
 * PHP output generated by the phc unparser
 * phc revision 359 (2005/10/11)
 */

print("Hello world!");
?>
```

Finally, `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 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](#)). If all this works, congratulations! You have successfully installed `phc` :-) Read [Getting Started](#) for more information on using `phc`.

phc -- the open source PHP compiler

\$LastChangedDate: 2006-01-04 16:19:56 +0000 (Wed, 04 Jan 2006) \$. Contents © the authors.

[Home](#) | [Downloads](#) | [Documentation](#) | [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 "transform.h"

class RenameFooToBar : public TreeTransform
{
    void pre_method_name(Token_method_name* in, Token_method_name** out)
    {
        if(*in == "foo")
            *out = new Token_method_name("bar");
    }
};
```

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

```
RenameFooToBar fooToBar;
php_script->transform(&fooToBar);
```

The Result

Running phc gives

```
<?php
/*
 * PHP output generated by the phc unparser
```

phc -- the open source PHP compiler

```
* phc revision 316M (2005/09/30)
*/

function bar()
{
    return 5;
}

$foo = bar();
print("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: 2006-01-04 16:19:56 +0000 (Wed, 04 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

Getting Started

For this introductory tutorial, we assume that you have successfully downloaded and installed phc, and that you know how to run it (as described in the [Installation Instructions](#)). This tutorial gets you started with using phc to develop your own tools for PHP.

The Abstract Syntax

phc has a particular view of PHP scripts, 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 rule 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.

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:
```

phpc -- the open source PHP compiler

```
AST_expr* expr;
Vector<AST_statement*>* iftrue;
Vector<AST_statement*>* 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. Similarly, the class definitions for arrays and array elements look like

```
class AST_array
{
public:
    Vector<AST_array_elem*>* array_elems;
};

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

The full description of the grammar can be found in the [Grammar Definition](#), and 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 [Converting 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.

So let's get our hands dirty and actually implement something. Let's write a very simple program 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.

In the directory `translation/`, you will find a file called `process_ast.cpp`. This is the file you will want to modify if you want to process the tree in any way. In this file, you will find a function called `process_ast()`:

```
void process_ast(AST_php_script* php_script)
{
    /*
     * Process php_script in whatever way you want
     *
     * If the user chooses to run the unparser (to PHP or to DOT),
     * the unparser will output the modified tree
     *
     * Read the tree transformation tutorial for more information.
     */
}
```

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 [Converting 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 vectors. So, to count the number of classes, all we have to do is query the number of elements in the `class_defs` vector:

```
void process_ast(AST_php_script* php_script)
{
    printf("%d class definitions found", php_script->class_defs->size());
}
```

Once you make this modification in `translation/process_ast.cpp`, recompile phc by running `make` (make sure you run `make` in the right directory!), and try running the new phc with some sample PHP script. It should tell you how many class definitions there are in the script!

Actually...

If you actually did try the modification described in the previous section, 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 [Converting PHP](#).

What's Next?

In theory, you now know enough to start implementing your own tools for PHP. Modify `process_ast()` in any way you want, and then pass the `--dump-php` flag to phc to request that phc outputs the tree back to normal PHP code after having executed `process_ast()`.

However, you will probably find that modifying the tree, despite being well-defined and easy to understand, is actually rather difficult. In particular, it is a lot of work. The good news is that phc provides explicit 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](#)

\$LastChangedDate: 2006-01-04 16:19:56 +0000 (Wed, 04 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

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!";
?>
```

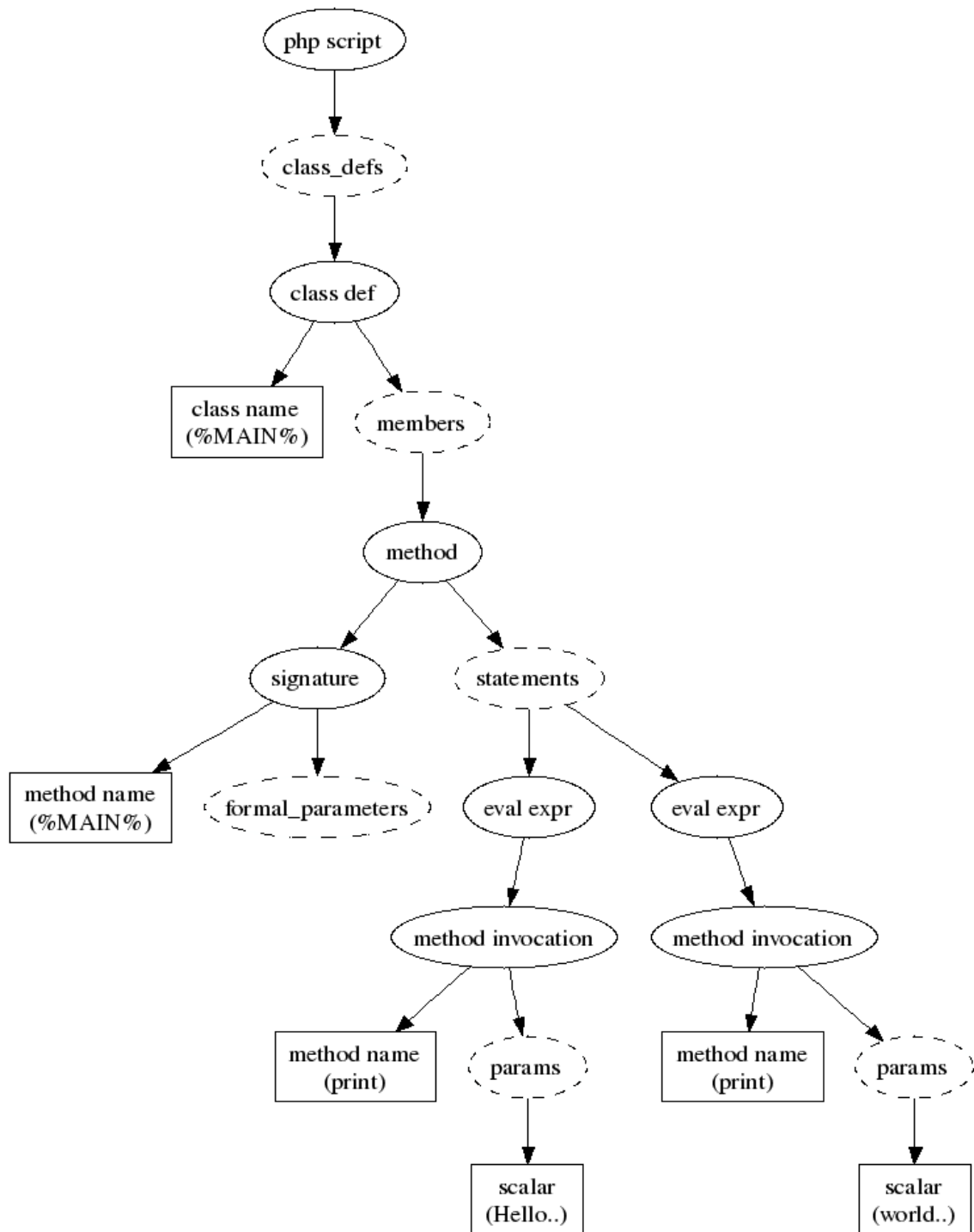
we should report two function calls.

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*
```

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

Remember that if you want to do anything with the tree, you need to add code to `process_ast` (see [Getting Started](#)). The following code 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.

```
Vector<AST_class_def*>::const_iterator ci;
Vector<AST_member*>::const_iterator mi;
Vector<AST_statement*>::const_iterator si;

AST_method* method;
AST_eval_expr* eval_expr;
AST_method_invocation* method_invocation;

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
            method_invocation = dynamic_cast<AST_method_invocation*>(eval_expr->expr);
            if(method_invocation == 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 `TreeTransform` (defined in `transform.h`). `TreeTransform` 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 "transform.h"

class CountFunctionCalls : public TreeTransform
{
private:
    int num_function_calls;

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

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

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

Save this class to a file called `CountFunctionCalls.h`. Now you need to modify `process_ast`, to make sure that `phc` actually runs our new traversal:

```
#include "CountFunctionCalls.h"

void process_ast(AST_php_script* php_script)
{
    CountFunctionCalls cfc;
    php_script->transform(&cfc);
}
```

Try to make this modification, and recompile phc. It should report the correct number of function calls.

Counting All Statements

Suppose we wanted to count all statements, rather than just function calls. We could define methods for `eval_expr`, `post_for`, `post_while`, `post_if`, etc., but there is an easier way. In addition to the type specific methods, there are also a number of more general methods for `member`, `statement`, `expr` and `refable_expr` (see the [grammar](#) for more information about these nodes).

Thus, to count the number of statements, it suffices to write something like

```
void post_statement(AST_statement* in, AST_statement** out)
{
    num_statements++;
}
```

Summarising, for a node of type `Foo`, which is an instance of a more general type `GenFoo`, phc will first call `pre_foo`, then `pre_gen_foo`, then visit all the node children, and finally call `post_foo` and `post_gen_foo` (in that order).

What's Next?

To find out how you can modify the tree, continue with [Tutorial 2](#).

\$LastChangedDate: 2006-01-04 19:58:54 +0000 (Wed, 04 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

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.)

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:expr_opt_ref*
method_name      ::= METHOD_NAME | 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 an `expr` (expression). 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 a public attribute called `value`. The type of `value` depends on the token; for `Token_int`, `value` is an integer, for `Token_real`, `value` is a double, etc. For most tokens, though, `value` will be an STL string.

Thus, we arrive at the following first attempt.

```
#include "transform.h"

class MySQL2DBX : public TreeTransform
{
public:
    void post_method_invocation(AST_method_invocation* in, AST_refable_expr** out)
    {
        Token_method_name* name;

        // Check the type of the function name
        name = dynamic_cast<Token_method_name*>(in->method_name);

        // If it is a Token_function_name (i.e., not an expression),
        // compare to "mysql_connect" and replace in case of a match
        if(name != NULL && *name->value == "mysql_connect")
        {
            in->method_name = new Token_method_name("dbx_connect");
        }
    }
};
```

```

    }
  }
};

```

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).

(If you are not familiar with the C++ `dynamic_cast` operator: `dynamic_cast<Type>(x)` tries to cast `x` to type `Type`. If that fails, the dynamic cast returns `NULL`; otherwise, it returns a pointer to `x` of type `Type`. Here we use it to check whether `method_name` is of type `Token_method_name`, as opposed to `AST_expr`.)

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 [Converting PHP](#).)

Finally, `NULL` is represented by `Token_null`. `Token_null` does not have a `value` field.

We are now ready to write our conversion function:

```

class MySQL2DBX : public TreeTransform
{
public:
    void post_method_invocation(AST_method_invocation* in, AST_refable_expr** out)
    {
        Token_method_name* name;
        Vector<AST_expr_opt_ref*>::iterator pos;
        Token_constant_name* module_name;
        AST_constant* module_constant;

        // Check the type of the function name
        name = dynamic_cast<Token_method_name*>(in->method_name);

        // If it is a Token_method_name (i.e., not an expression),
        // compare to "mysql_connect" and replace in case of a match
        if(name && *name->value == "mysql_connect")
        {

```

phc -- the open source PHP compiler

```
// Check for too many parameters
if(in->params->size() > 3)
{
    printf("Error: unable to translate call "
           "to mysql_connect on line %ld\n", in->line_number);
    return;
}

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

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

pos = in->params->begin();
in->params->insert(pos, module_constant); pos++;
/* Skip host */ pos++;
in->params->insert(pos, new Token_null());
}
};
```

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: 2006-01-04 19:58:54 +0000 (Wed, 04 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

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 `RemoveConcatNull` and can be found in `translation/removeconcatnull.cpp`. 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 `RemoveConcatNull` transform cleans this up. In this tutorial we will explain how this transform can be written.

Implementation

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). This would be a good start:

```

class RemoveConcatNull : public TreeTransform
{
public:
    void pre_bin_op(AST_bin_op* in, AST_expr** out)
    {
        // Find concat operators
        if(*in->op->value == ".")
        {
            // ...
        }
    }
}

```

What are we going to do inside the `if`? Previously we modified `in` to do the transform, but now we cannot do that: `in` is a binary operator, and we want to get rid of that binary operator altogether! In fact, if the left operand of the expression is the empty string, we want to replace the binary operator by its right operand, and vice versa for the other operand.

If you want to modify the structure of the tree, you will have to modify `*out` instead of modifying `in`. If you look closely at the signature of `pre_bin_op`, you will see that even though the type of `in` is `AST_bin_op*`, the type of `*out` is `AST_expr*`. This means that you can replace the binary operator by any other node, as long as that node is an expression (inherits from `AST_expr`).

Fortunately for us, both operands of the binary operator are expressions (the grammar says so), so we can safely assign either operand to `*out`. Here is the full transform:

```

class RemoveConcatNull : public TreeTransform
{
public:
    void pre_bin_op(AST_bin_op* in, AST_expr** out)
    {
        // Find concat operators
        if(*in->op->value == ".")
        {
            Token_string* s;

            // Replace with right operand if left operand is the empty string
            s = dynamic_cast<Token_string*>(in->left);
            if(s != NULL && *s->value == "")
                *out = in->right;
            else
            {
                // Replace with left operand if right operand is the empty string
                s = dynamic_cast<Token_string*>(in->right);
                if(s != NULL && *s->value == "")
                    *out = in->left;
            }
        }
    }
}

```

We use the dynamic cast to check whether either operand is a string (as opposed to another type of expression), and then, if the operand is indeed a string, whether it is the empty string. If so, we replace the binary operator with the other operand.

Note that `*out` gets initialised to `in` before the `pre_bin_op` is invoked, so if you do not assign to `*out`, the structure of the tree remains intact.

Modifying *out and Traversal Order

As explained in [Tutorial 1](#), for a node of type `Foo`, which is an instance of a more general type `GenFoo`, phc will first call `pre_foo`, then `pre_gen_foo`, then visit all the node children, and finally call `post_foo` and `post_gen_foo` (in that order).

However, what happens when `pre_foo` returns a brand new node (of type `Bar`, say), instead of modifying the node that's there? The answer is that instead of continuing with the children of this (new) node, phc instead starts over and calls `pre_bar` (and then `pre_gen_bar`) on the new node. This means that you are guaranteed that the transform will not skip any nodes in the tree while going down the tree.

Recursion Threshold

Calling the pre-transform again on new nodes avoids some nasty surprises, but it does have one unfortunate consequence. Consider a transform that increments all integers in a script by 1 (obviously not a useful transform, but it is a simple one and illustrates the problem). We could write the transform in two ways. Here is the first implementation:

```
class IncrementInts : public TreeTransform
{
public:
    void pre_int(Token_int* in, AST_expr** out)
    {
        in->value++;
    }
};
```

If you include this transform in `process_ast` and test it, you will find that it works as expected. Note that this transform does not assign to `*out`, but simply modifies `in`. The second transform is very similar to the first, but instead of modifying the `Token_int` node, it replaces the `Token_int` node by a brand new one:

```
class IncrementInts : public TreeTransform
{
public:
    void pre_int(Token_int* in, AST_expr** out)
    {
        *out = new Token_int(in->value + 1);
    }
};
```

Before you test this transform, try to reason out what will happen first, taking into account what we just told you about the traversal order when assigning to `*out`.

If you think about it, you will realise that this transform will loop forever! Because `pre_int` returns a new node, phc will then call `pre_int` *again* on that new node, *ad infinitum*. To avoid the compiled code crashing phc tries to detect the situation. When you run the transform, phc will exit with

```
Internal error: Transform `IncrementInts' appears to be looping on a node of type `Token_int'
```

The algorithm to detect this situation is actually very simple: every transform has a so-called *recursion threshold*. This is the maximum number of times a single node is allowed to be replaced. As soon as this threshold is exceeded, phc will exit with the above error message.

The default threshold is 10. If this is too low for one of your transforms, you can increase it by defining `get_recursion_threshold`:

```
class SomeTransform : public TreeTransform
{
public:
    long get_recursion_threshold()
    {
        // Use a recursion threshold of 500 instead of 10
        return 500;
    }

    // Other methods..
};
```

What's Next?

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

\$LastChangedDate: 2006-01-04 19:58:54 +0000 (Wed, 04 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

Tutorial 4: Using State

This tutorial does not introduce any new features of phc, but explains 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 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 a nightmare; 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`. Then we process the method, and as soon as we find a function call to a DBX function, we set

uses_dbx to true. 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. Note the use of pre_method and post_method to initialise and check use_dbx, respectively.

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

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

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

    void post_method_invocation(AST_method_invocation* in, AST_refable_expr** out)
    {
        Token_method_name* name;

        // Check for Token_method_name (versus AST_expr)
        name = dynamic_cast<Token_method_name*>(in->method_name);

        // Check for dbx_
        if(name != NULL && name->value->find("dbx_") == 0)
        {
            uses_dbx = true;
        }
    }
};
```

It's that simple!

pre Versus post Functions

Sometimes the choice between pre_xxx and post_xxx is clear; for example, pre_method and post_method have very distinct responsibilities in the above transform. In other cases, the choice is not so clear: should we use post_function_call or pre_function_call to set uses_dbx? In general, we recommend using post if the choice is unclear; the post method gets run after all children have been transformed, so the post method operates on the “final” final version of the node; the pre method might end up drawing conclusions that get invalidated by the transform of the children of the node.

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: 2006-01-04 16:19:56 +0000 (Wed, 04 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

Tutorial 5: Modifying the Traversal Order

As explained in the previous tutorials (in particular, [Tutorial 1](#)), when the TreeTransform walks over 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)
    {
        print("Do something");
    }
    else
    {
        print("Do something else");
    }
?>
```

is translated to

```
<?php
    if($expr)
    {
        /* TODO: Insert comment */
        print("Do something");
    }
    else
    {
        /* TODO: Insert comment */
        print("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* node)
{
    TRANSFORM(AST_expr*, node->expr);
    TRANSFORM_VECTOR(AST_statement*, node->iftrue);
    TRANSFORM_VECTOR(AST_statement*, node->iffalse);
}
```

TRANSFORM and TRANSFORM_VECTOR are defined in `transform.h`; the first argument is the type of the node to visit, and the second argument is the name of that node. TRANSFORM_VECTOR calls TRANSFORM for every element of the specified vector. So, 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:

```
class CommentIfs : public TreeTransform
{
private:
    bool comment;

public:
    CommentIfs()
    {
        comment = false;
    }

    void children_if(AST_if* node)
    {
        TRANSFORM(AST_expr*, node->expr);
        comment = true;
        TRANSFORM_VECTOR(AST_statement*, node->iftrue);
        comment = true;
        TRANSFORM_VECTOR(AST_statement*, node->iffalse);
        comment = false;
    }

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

        comment = false;
    }
};
```

What's Next?

This is the last tutorial in this series on using the `TreeTransform` class. Of course, there is no substitute for experimentation; to really understand how this thing 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 converted into the abstract syntax
- The definition of the C++ classes for the AST nodes in `parsing/ast.h`
- The definition of the `TreeTransform` class in `translation/transform.h`

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!

phc -- the open source PHP compiler

\$LastChangedDate: 2006-01-04 16:19:56 +0000 (Wed, 04 Jan 2006) \$. Contents © the authors.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

What's in Store?

We have planned SSA and type inference for the next release of phc. You can expect this release in a few months time.

Static Single Assignment (SSA) Form

The first thing that we will do after building the AST, is convert the tree into SSA form. SSA means that every variable only gets assigned in one location in the program. So, 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
    $x = dbx_connect(...);
    use($x);

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

should get translated to

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

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

In particular, the second `use($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

phc -- the open source PHP compiler

case, refactoring should fail (or insert code to explicitly distinguish between a few types).

So stay tuned :-)

\$LastChangedDate: 2006-01-04 16:19:56 +0000 (Wed, 04 Jan 2006) \$. Contents © the authors.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

The Abstract Grammar

This is the full and authoritative definition of the phc abstract grammar for PHP. 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 ::= static_var*
static_var ::= VARIABLE_NAME expr

unset ::= variable*

declare ::= directive+ (statement*)?
    
```

phc -- the open source PHP compiler

```
directive ::= DIRECTIVE_NAME expr

try ::= statement* 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 | refable_expr
    | INT | REAL | STRING | BOOL | NULL

refable_expr ::=
    variable | pre_op | post_op | array
    | method_invocation | new | clone

assignment ::= dst:refable_expr src:expr_opt_ref

expr_opt_ref ::= expr | refed_expr
refed_expr ::= refable_expr

list_assignment ::= list_elements expr
list_elements ::= (list_element?)*
list_element ::= refable_expr | 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 | expr

target ::= expr | CLASS_NAME

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

array ::= array_elem*
array_elem ::= key:expr? val:expr_opt_ref

method_invocation ::= target method_name params:expr_opt_ref*
method_name ::= METHOD_NAME | expr

new ::= class_name params:expr_opt_ref*
class_name ::= CLASS_NAME | expr

clone ::= expr
```

Token values

As explained in the [grammar formalism](#), a token x is represented by a class `Token_x`. This class will have an attribute called `value` representing the value of the terminal symbol. For most terminal

phc -- the open source PHP compiler

symbols, the type of `value` will be an STL string. The exceptions are listed below:

`Token_int::value` long

`Token_real::value` double

`Token_bool::value` bool

`Token_null` *does not have a value*

`$LastChangedDate: 2006-01-06 15:19:54 +0000 (Fri, 06 Jan 2006) $`. Contents © the authors.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

The Grammar Formalism

The grammar formalism we use is slightly unorthodox, but 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.

Unlike most grammar formalisms, 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`).

However, 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`. All classes representing terminal symbols have an attribute `value` that represents the value of the terminal symbol. The type of `value` depends on the symbol; for most symbols, this will be an STL string. However, in `Token_int`, for example, the type of `value` will be `long` (these types are listed in the grammar definition). 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.

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 (STL) `vector<AST_b>` (or `vector<Token_b>` for terminal symbols) 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 `Vector<AST_foo>` will be the plural form of *foo*. However, as mentioned above, this can be overridden by specifying a label.

As an example, consider the rule for `variable` in the grammar. A `variable` is a `refable_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_refable_expr
{
public:
    AST_variable_name* variable_name;
    Vector<AST_expr*>* 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 a `Vector<AST_a>`. If a list is specified, but empty, the vector will simply contain no elements. If the list is not specified at all, the vector 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 vector, but elements of the vector may be `NULL`. This is used for example to denote empty array indices (`a[]`) in the rule for `variable`.

\$LastChangedDate: 2006-01-09 12:13:18 +0000 (Mon, 09 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

Converting 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%::print("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 params:expr_opt_ref*
method_name ::= METHOD_NAME | expr
```

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 | expr
```

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.

<code>target?</code>	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., <code>\$x->y</code>) or a class
----------------------	---

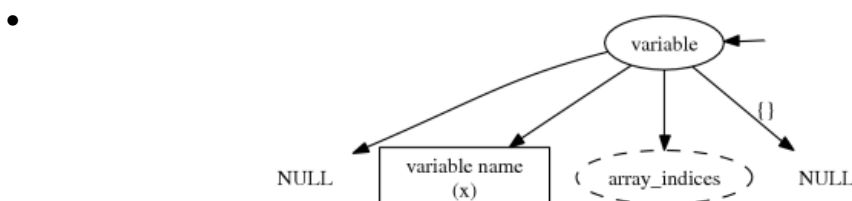
name (for a static class attribute, e.g. `FOO:: $y`). Unlike function calls however, in variables the target is optional (indicated by the question mark). If no target is specified, the variable refers to a *local* variable in a method (see [Global Variables](#) for information on how we deal with the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

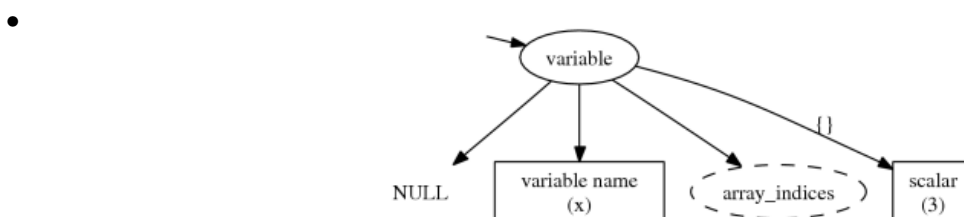
array_indices: (expr?)* A variable may have one or more array indices, for example `$x[3][5]`. The strange construct `(expr?)*` means: a list of `(*)` optional `(?)` expressions. For example, `$x[4][]` is a list of two expressions, but the second expression is not given. In PHP, this means “use the next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

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

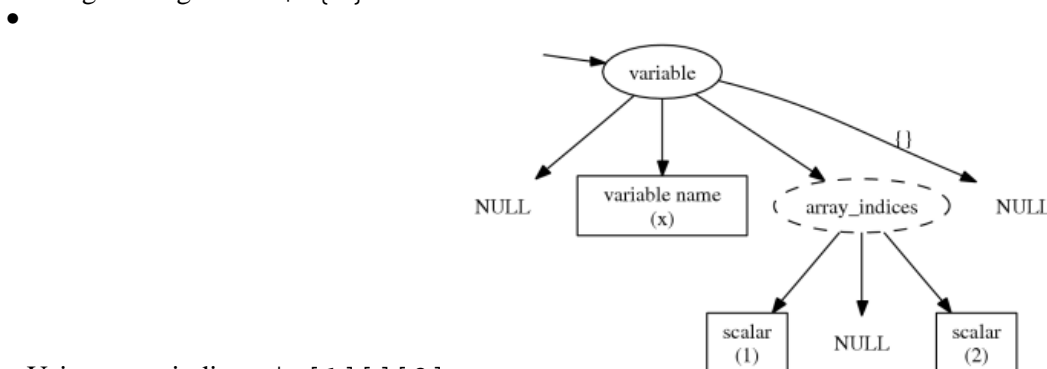
We illustrate the various possibilities using diagrams:



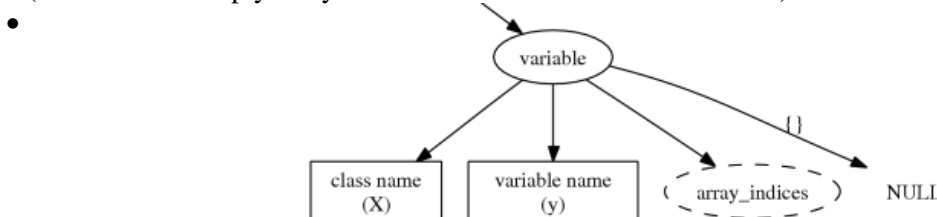
The simple case: `$x`
 Note that the name of the variable is `x`, not `$x`



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

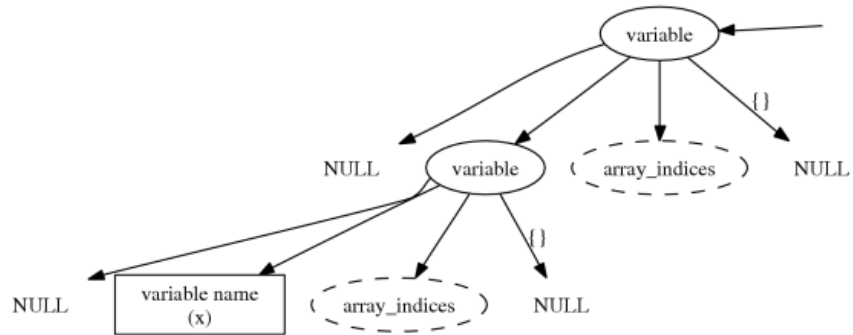


Using array indices: `$x[1][][2]`
 (Note that the empty array index means “next available” in PHP)



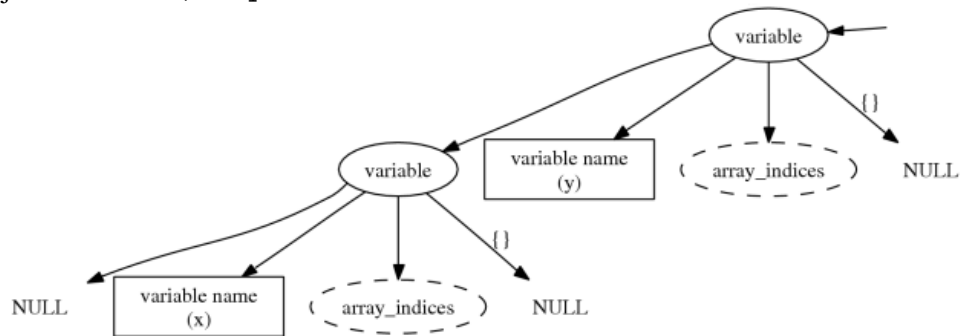
Class constants: `X::y`
 Note that here also the variable name is `y`, not `$y`. The fact that you must write `$x->y` but `X::$y` in PHP disappears in the abstract syntax.

- Variable variables: `$$x`



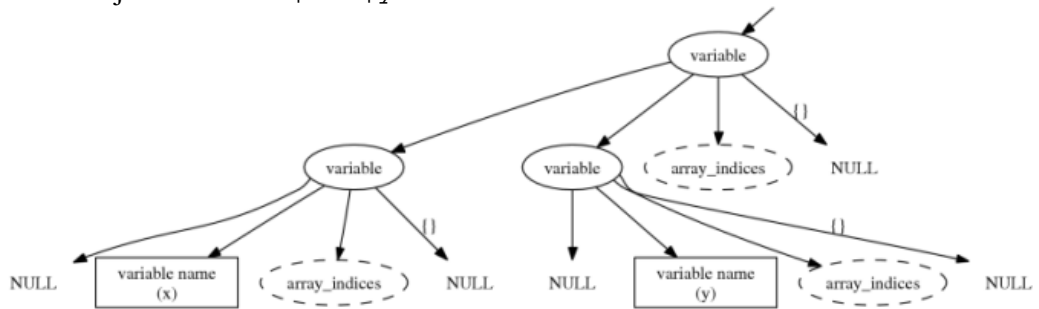
Note how the name of the variable (second component) is now given by another variable.

- 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 `Vector<char*>` 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();
```

phpc -- the open source PHP compiler

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 lose any comments. If something goes wrong with comments, please [send](#) us a sample program that shows where it goes wrong. Note that whitespace in multiline 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 `elseif`s, the greater the level of nesting. This transformation can be “hidden” by the unparser.

Miscellaneous Other Changes

- `echo` and `print` both get translated to a function call to `print`. If `echo` has multiple (comma separated) arguments, they get translated into multiple function calls (`echo a, b;` becomes `print(a); print(b);`). Fragments of inline HTML also become arguments to a function call to `print`.
- 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;`
- 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.

Finally, the `phpc` 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.

\$LastChangedDate: 2006-01-09 12:13:18 +0000 (Mon, 09 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

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 [Converting 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

Converting PHP explains how we deal with comments. All 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.

The only problem that we are aware of with our method of dealing with comments is how we deal with whitespace in multiline 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 :-)

Finally, 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).

\$LastChangedDate: 2006-01-08 00:54:58 +0000 (Sun, 08 Jan 2006) \$. Contents © the authors.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

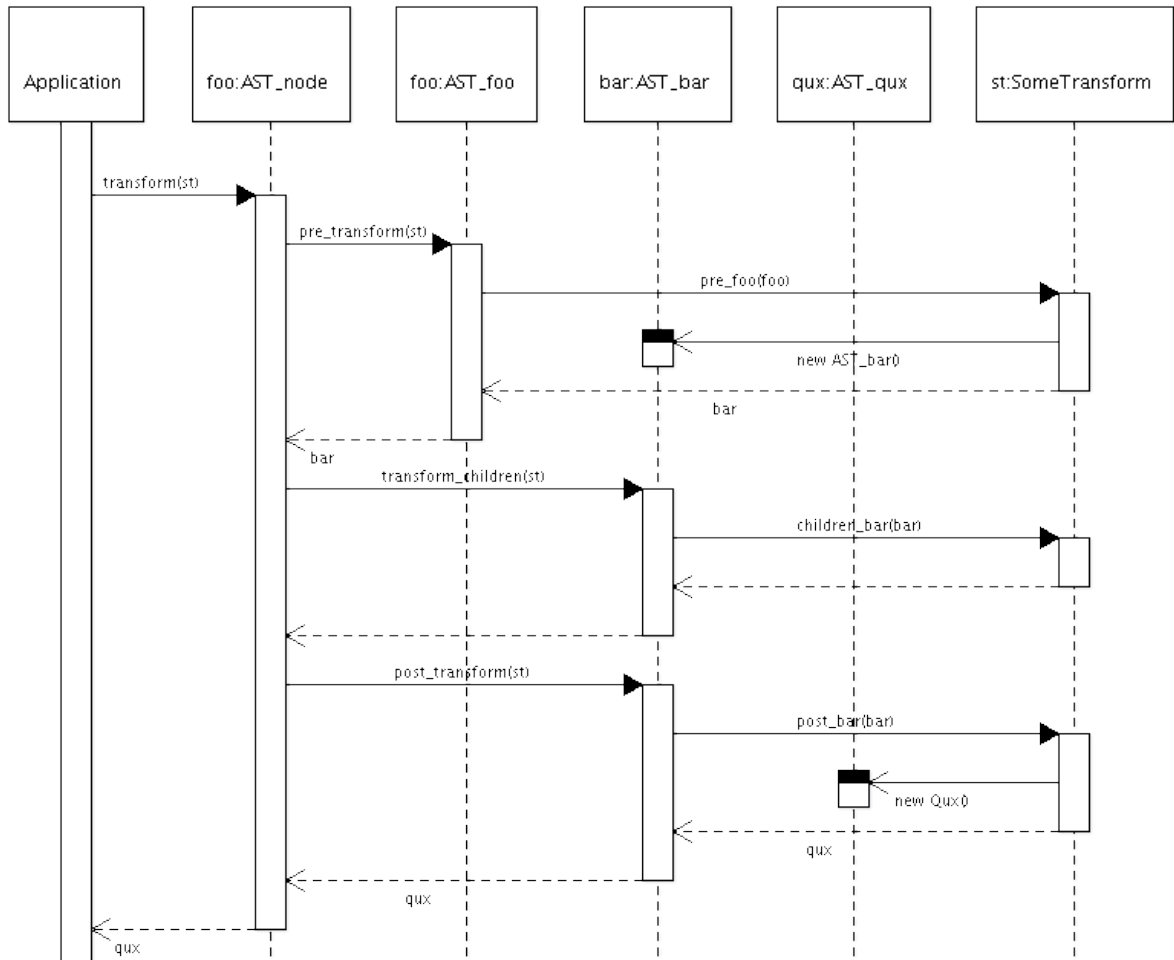
Implementation of the Tree Traversal API

This document describes the implementation of the tree traversal API and is of interest only to programmers wishing to understand or modify it. You do not need to read this document to be able to use the API (refer to the [tutorials](#) instead).

Note that if you want to *truly* understand the implementation of the tree traversal API, you should also read the article “[Memory Layout for Multiple and Virtual Inheritance](#)”.

Overview

We explain the general idea first and worry about the details later. The following sequence diagram depicts what happens when `transform` is invoked on a node `foo` of type `AST_foo`. Note that `foo` is shown twice in this diagram; once with type `AST_node`, and once with type `AST_foo`. This is meant to illustrate that some behaviour of `foo` is generically defined in `AST_node`, and some behaviour is defined in `AST_foo` itself. Obviously, both `foos` refer to the *same* object.



Actually, this diagram is a simplification, and not one hundred percent accurate, but it is large enough as it is and we said we'd worry about details later :) There are a few things to note about this diagram.

- `transform` is defined generically in `AST_node` and is not redefined in any of the subclasses.
- When a client calls `transform` on a node, the first thing `transform` does is to call `pre_transform` on *itself*.
- `pre_transform`, as well as `transform_children` and `post_transform`, are not defined generically (in fact, they are pure virtual in `AST_node`) but are defined in each and every class in the hierarchy. However, the implementation of these methods is very simple, since all they do is call the corresponding method in the `TreeTransform` object (reminiscent of the “Strategy” design pattern). `transform` does not call these methods directly, because, as mentioned, it is defined generically and thus does not know which method in the tree transform to call. We essentially use the C++ virtual function dispatcher to invoke the appropriate methods in the tree transform.
- The default implementations of `pre_xxx` and `post_xxx` in the `TreeTransform` class do nothing, and simply return the node unaltered. The default implementation of `children_xxx` calls `transform` recursively on all children of the node, and adjusts the corresponding pointers on the parent node. So, after `children_xxx`, all pointers in the parent node now point to the *transformed* children.
- Even if `pre_xxx` or `post_xxx` return new nodes, the `transform` method of the *original* node remains in charge of the transformation, until it returns the node returned by `post_xxx` to the caller.

As mentioned, this diagram is a simplification. There are two things it does not show. First, when the pre-transform returns a new node, `transform` starts from scratch with this new node (i.e., it then calls the pre-transform on the new node). Second, for many nodes, the pre-transform and post-transform methods actually call *two* methods in the tree transform; one for the node itself (for example, `AST_if`) and for the node's more general type (for example, `AST_statement`). We will see these details when we look at the code.

The Generic `transform` Method

We show a simplification of the actual implementation.

```
AST_node* AST_node::transform(TreeTransform* transform)
{
    AST_node* transformed_in = this;
    AST_node* transformed_out;

    /*
     * When a transform returns a new node, we recursively call the pre_transform
     * on that new node. If instead the transform only modifies the node, the
     * pre_transform gets invoked only once.
     */
    for(;;)
    {
        transformed_out = transformed_in->pre_transform(transform);
        if(!transformed_out || transformed_out == transformed_in)
            break;
        transformed_in = transformed_out;
    }

    if(transformed_out)
    {
        transformed_out->transform_children(transform);
        transformed_out = transformed_out->post_transform(transform);
    }
}
```

```

    return transformed_out;
}

```

(The actual implementation must keep track of the recursion threshold, as explained in [Tutorial 3](#)). The main complication in this method arises from the fact that we might have to call the pre-transform method repeatedly before continuing with the node's children. The `for` loop exits as soon as the pre-transform does not return a new node.

Note that `transform` knows nothing about the types of nodes it works with; as far as it is concerned, they are all simply `AST_nodes`. In particular, it returns an `AST_node` (as opposed to a more accurate type; for example, `AST_if::transform` could return an `AST_statement`). In a way, this is unfortunate, and could probably be avoided if we implemented `transform` specifically for every node in the AST hierarchy. However, this method is the core of the transformation API, and it nice is to have this core defined in one location. Modifying and understanding the API is difficult enough as it is :-)

The Implementation in the AST nodes

As mentioned in the overview, the implementation of `pre_transform`, `post_transform` and `transform_children` in the AST nodes is fairly straight-forward (we must be careful with the pointer types though). A typical implementation of `pre_transform` for nodes without a more general type is

```

AST_node* AST_class_def::pre_transform(TreeTransform* t)
{
    AST_class_def* out = this;
    t->pre_class_def(this, &out);
    return out;
}

```

The corresponding signature of `pre_class_def` in `TreeTransform` is

```

void pre_class_def(AST_class_def* in, AST_class_def** out);

```

This construction may seem somewhat unconventional. The alternative would be to change the signature of `pre_class` to

```

AST_class_def* pre_class_def(AST_class_def* in);

```

which is more natural. The implementation of `pre_transform` in `AST_class_def` can then be changed to

```

AST_node* AST_class_def::pre_transform(TreeTransform* t)
{
    return t->pre_class_def(this);
}

```

which is certainly simpler. However, this makes writing tree transforms more difficult. Most methods in a typical tree transform will only want to modify `in` (or even just collect some information), without modifying the structure of the tree. In this “alternative” setup, such a method would look like

```

AST_class_def* SomeTransform::pre_class_def(AST_class_def* in)
{
    // Do something..
    return in;
}

```

These “**return in**”s are then required in each and every method, which does little to improve the readability of the tree transform. The construction we have adopted, using the `**out` parameter and initialising `*out` to be equal to `*in` saves the programmer from worrying about `*out` (or the return value of the method) if this is not necessary.

For a node with a more general type, this method is only slightly more complicated:

```
AST_node* AST_if::pre_transform(TreeTransform* t)
{
    AST_statement* out = this;
    t->pre_if(this, &out);
    t->pre_statement(out, &out);
    return out;
}
```

The corresponding signatures in `TreeTransform` are

```
void pre_if(AST_if* in, AST_statement** out);
void pre_statement(AST_statement* in, AST_statement** out);
```

Note that this code is completely type-safe. You might expect the signature of `pre_transform` in `AST_if` to be this instead:

```
AST_statement* AST_if::pre_transform(TreeTransform* t)
```

(Note the different return type.) We would in fact prefer to define `pre_transform` that way; however, `pre_transform` is defined in `AST_node`:

```
virtual AST_node* pre_transform(TreeTransform* transform) = 0;
```

Since only the most recent versions of `gcc` implement covariant return types, we are forced to use the less accurate signature (or make `phc` more difficult to install and use by requiring a very recent version of `gcc`).

The implementation of `post_transform` is nearly identical:

```
AST_node* AST_if::post_transform(TreeTransform* t)
{
    AST_statement* out = this;
    t->post_if(this, &out);
    t->post_statement(out, &out);
    return out;
}
```

Note that types dictate the order in which we call `post_if` and `post_statement` (and similarly, `pre_if` and `pre_statement`). Finally, the implementation of `transform_children` is as simple as can be (and is typical of code that uses the Strategy design pattern):

```
void AST_if::transform_children(TreeTransform* transform)
{
    transform->children_if(this);
}
```

Default Implementation of `TreeTransform`

The default implementation of `pre_xxx` and `post_xxx` in the `TreeTransform` class is very simple: it does nothing. The only interesting code in the default `TreeTransform` class is in the

children_xxx methods. We show a typical implementation:

```
void TreeTransform::children_if(AST_if* node)
{
    TRANSFORM(AST_expr*, node->expr);
    TRANSFORM_VECTOR(AST_statement*, node->iftrue);
    TRANSFORM_VECTOR(AST_statement*, node->iffalse);
}
```

TRANSFORM and TRANSFORM_VECTOR are macros defined in transform.h; TRANSFORM transforms the specified attribute and TRANSFORM_VECTOR transforms every element of the vector. Both require the type of the elements they must transform. TRANSFORM is defined as follows:

```
#define TRANSFORM(type, member)
    if(member)
    {
        member = dynamic_cast<type>((member)->transform(this));
        assert(member);
    }
```

(For clarity, we have omitted the backslashes that the C preprocessor requires at the end of each line.) AST_node::transform returns an AST_node for reasons outlined above, so we need to cast this node back to the appropriate type. This cast cannot be done statically (see [Memory Layout for Multiple and Virtual Inheritance](#)), so we must use a dynamic cast. However, we know that the cast should succeed (unless there is a bug somewhere), so we assert that the result of the cast should not be NULL. Note that the reference to this in the call to transform refers to the TreeTransform object.

The implementation of TRANSFORM_VECTOR does the same thing for every element of the vector and is defined in terms of TRANSFORM:

```
#define TRANSFORM_VECTOR(type, vector)
    if(vector)
    {
        Vector<type>::iterator i;
        for(i = vector->begin(); i != vector->end(); i++)
            TRANSFORM(type, *i);
    }
```

\$LastChangedDate: 2006-01-08 00:54:58 +0000 (Sun, 08 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

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;
};

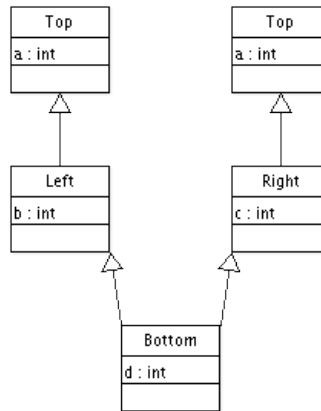
class Left : public Top
{
public:
    int b;
};

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

phc -- the open source PHP compiler



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;
};

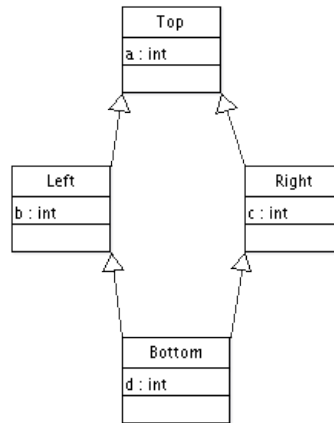
class Right : virtual public Top
{
public:
    int c;
};

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

```

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

phc -- the open source PHP compiler



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

```

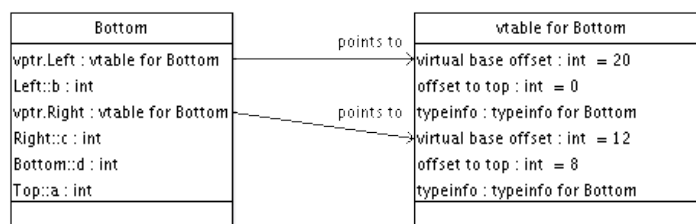
Left::Top::a
Left::b
Right::c
Bottom::d
  
```

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;
  
```

phc -- the open source PHP compiler

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

```

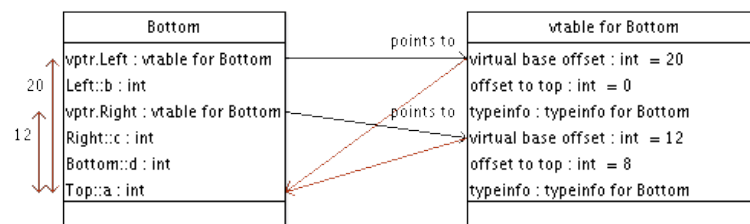
Bottom* bottom = new Bottom();
Right* right = bottom;
int p = right->a;
    
```

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

```

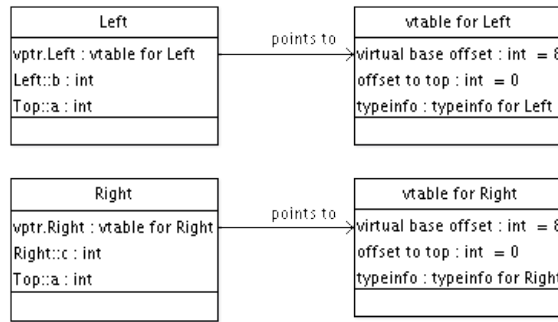
                Bottom
                vptr.Left
                Left::b
                right
                →  vptr.Right
                Right::c
                Bottom::d
                Top::a
    
```

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:

phc -- the open source PHP compiler



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).

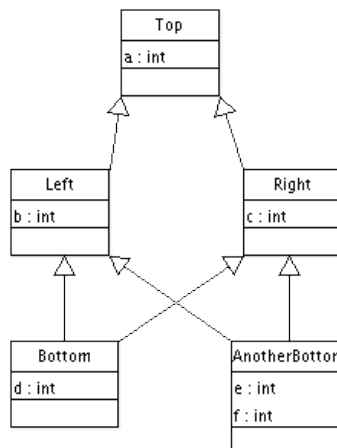
Downcasting

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

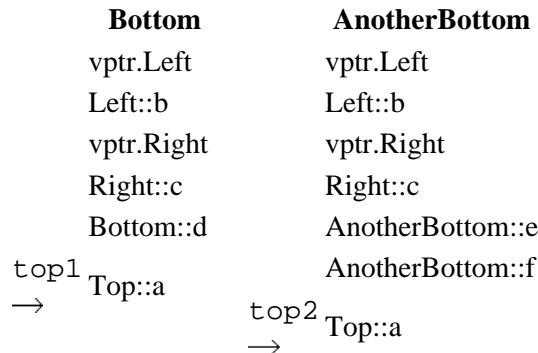


Now consider the following code.

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

phc -- the open source PHP compiler

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);
```

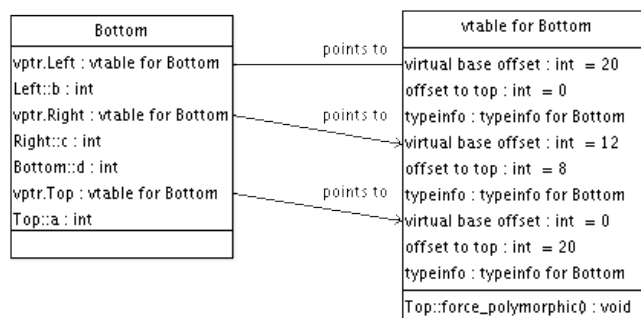
However, the compiler is still unhappy:

```
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 function to `Top`:

```
class Top
{
private:
    virtual void force_polymorphic() {}
public:
    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 tinco.cc.

Concluding Remarks

Finally, we tie a couple of loose ends.

(In)variance of Double Pointers

This is where 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;
```

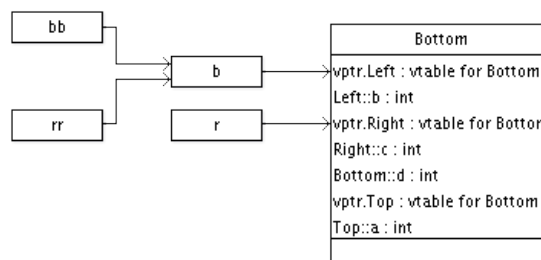
(the value of `b` gets adjusted by 8 bytes before it is assigned to `r`, so that it points to the `Right` section of the `Bottom` object). Thus, we can legally assign a `Bottom*` to a `Right*`. What about `Bottom**` and `Right**`?

```
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:

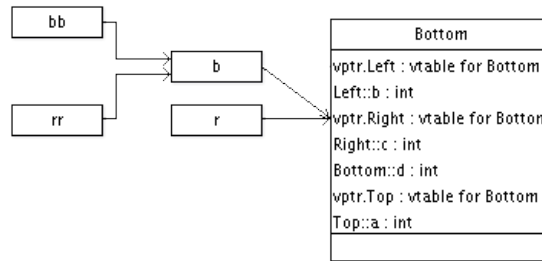


So, `bb` and `rr` both point to `b`, and `b` and `r` point to the appropriate sections of the `Bottom` object. Now consider what happens when we assign to `*rr` (note that the type of `*rr` is `Right*`, so this assignment is valid):

```
*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:

phc -- the open source PHP compiler



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.

phc -- the open source PHP compiler

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

\$LastChangedDate: 2006-01-06 15:19:54 +0000 (Fri, 06 Jan 2006) \$. Contents © the authors.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

Spinoff Projects

There are a number of projects that can be spun off from the phc compiler project. Below is a list of 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](#)
- [PHP to XML Converter](#)

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.

The next release of phc will include type inference, which will increase the usefulness of phc for refactoring greatly (see [What's in Store?](#)).

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

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.

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.

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?](#)).

PHP to XML Converter

The phc [abstract grammar](#) can easily be converted into an XML DTD, and (a particular instance of) the abstract syntax tree can easily be converted into a corresponding XML document (this conversion would in fact be very easy to implement using the `TreeTransform` API; see the tutorials). This would give an XML representation of a PHP script, which can then be processed using XSLT.

Of course, phc provides specialised support for modifying the tree too; but some people might find XSLT easier to work with. Note however that future releases of phc will also operate on a number of graphs (derived from the tree), which may be much more difficult to work with using XML tools.

\$LastChangedDate: 2006-01-08 00:54:58 +0000 (Sun, 08 Jan 2006) \$. Contents © the [authors](#).

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Mailing List](#)

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](#).

\$LastChangedDate: 2005-12-22 15:54:23 +0000 (Thu, 22 Dec 2005) \$. Contents © the [authors](#).

Join

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

Email