

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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

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.

\$LastChangedDate: 2005-11-02 16:43:55 +0000 (Wed, 02 Nov 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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 and references. The tutorials try to get you started without necessarily going into all the details. They should be read in the order that they are listed here. 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.

Tutorials

Demo

For a quick overview of what the current phcrelease 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`.

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: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff projects](#) | [Contact us](#)

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
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: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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_function_name(Token_function_name* in, Token_function_name** out)
    {
        if(*in == "foo")
            *out = new Token_function_name("bar");
    }
};
```

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

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

The Result

Running phc gives

```
<?php
/*
```

phc — the open source PHP compiler

```
* PHP output generated by the phc unparser
* 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: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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:
```

```

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](#)

phc -- the open source PHP compiler

\$LastChangedDate: 2005-10-27 09:52:51 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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!";
?>
```

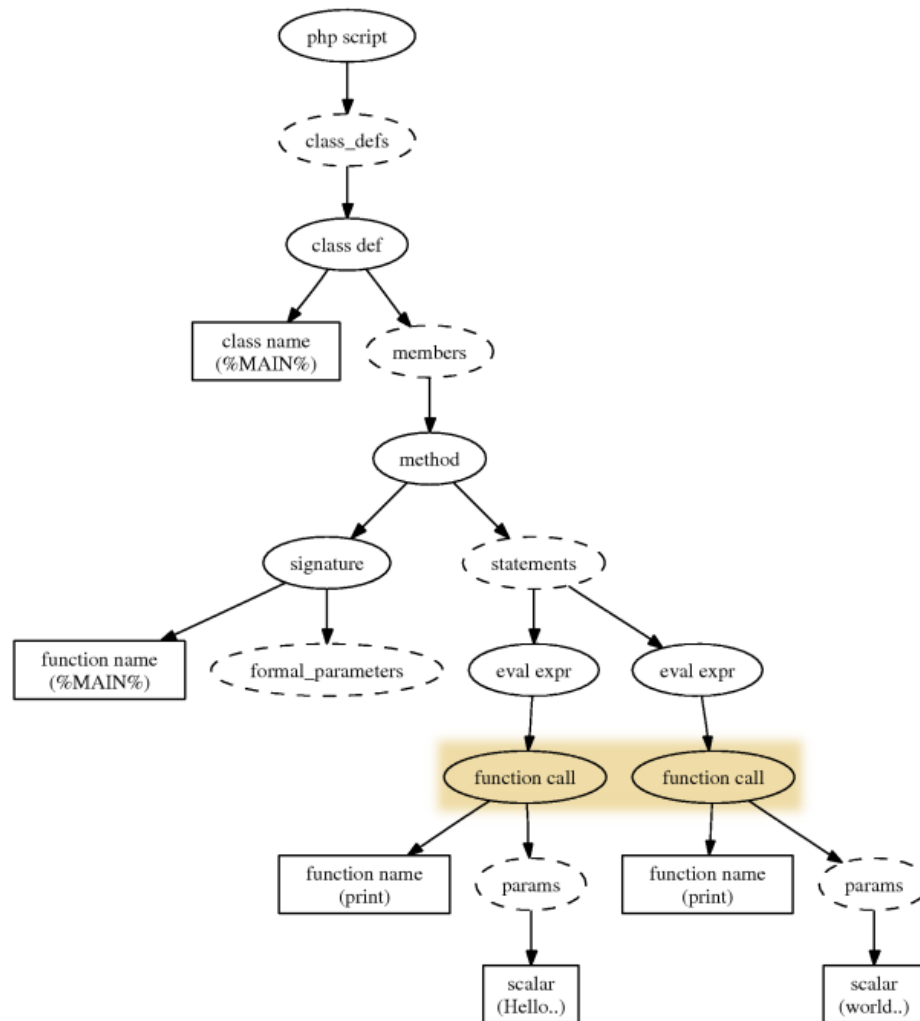
we should report two function calls.

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 ::= FUNCTION_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 highlighted. The `eval_expr` nodes just above them probably need some explanation. There are many different types of statement 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_function_call* function_call;

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
            function_call = dynamic_cast<AST_function_call*>(eval_expr->expr);
            if(function_call == 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 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_function_call(AST_function_call* 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: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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](#):

```
function_call ::= target function_name params:expr_opt_ref*
function_name ::= FUNCTION_NAME | expr
```

(The `target` of a function call 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 `function_name`. The grammar tells us that a `function_name` is either a `FUNCTION_NAME` or an `expr`. If a symbol is written in ALL_CAPS in the grammar, that means it refers to a literal. In this case, to an actual function name (such as `mysql_connect`). In PHP, it is also possible to call a function 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 function calls 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` inherit from the `STL string` class and can be treated as such.

Thus, we arrive at the following first attempt.

```
#include "transform.h"

class MySQL2DBX : public TreeTransform
{
public:
    void post_function_call(AST_function_call* in, AST_refable_expr** out)
    {
        Token_function_name* name;

        // Check the type of the function name
        name = dynamic_cast<Token_function_name*>(in->function_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 == "mysql_connect")
        {
            in->function_name = new Token_function_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 `function_name` is of type `Token_function_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 a `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 a `Token_scalar`. Although most `Token_something` classes have a very simple constructor that takes a string only (such as `Token_function_name`, above), `Token_scalar` is a special case because there are many different types of scalars (ints, floats, booleans, etc.). Therefore, `Token_scalar` has a number of specialised constructors; the one we need is `new_NULL`. For a full list, see the definition of the `Token_scalar` class in the file `parsing/ast.h`.

We are now ready to write our conversion function:

```
class MySQL2DBX : public TreeTransform
{
public:
    void post_function_call(AST_function_call* in, AST_refable_expr** out)
    {
        Token_function_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_function_name*>(in->function_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 && *name == "mysql_connect")
{
    // Check for too many parameters
    if(in->params->size() > 3)
    {
        printf("Error: unable to translate call "
            "to mysql_connect on line %d\n", in->line_number);
        return;
    }

    // Modify name
    in->function_name = new Token_function_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, Token_scalar::new_NULL());
}
}
};

```

If we apply this transformation to

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

We get

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

Error Messages

Notice how we print the warning in the transformation above:

```

printf("Error: unable to translate call "
    "to mysql_connect on line %d\n", in->line_number);

```

Every node in the tree has an associated `line_number`. In this case, this is the number of the line where the call to `mysql_connect` originated. This line number can be used to give more helpful error and warning messages to the user.

If an error is serious enough that phc should not continue after calling `process_ast` (for example, if it should not output the tree back to normal PHP, even if the user requested `--dump-phc`, you should modify the global variable `num_errors`. If this is greater than 0 after calling `process_ast`, phc will exit at that point. So, you could modify the above procedure to be:

```

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

```

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: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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 nodes in the tree, in this tutorial we will look at modifying (parts of) the tree itself.

Function targets

Recall the grammar rules for function calls:

```
function_call ::= target function_name params:expr_opt_ref*
function_name ::= FUNCTION_NAME | expr
```

As we mentioned before, 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. As we explained before, functions that are declared outside the scope of a class will be added as methods to a special class called %MAIN%. Code that is defined outside the scope of any function, gets added to a special method in %MAIN% called %run%. So, phc interprets the following script

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

    $x = foo();
?>
```

as

```
<?php
class %MAIN%
{
    static function foo()
    {
        return 5;
    }

    static function %run%()
    {
        $x = %MAIN%::foo();
    }
}

%MAIN%::%run%();
?>
```

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()`).

Variables

The grammar rule for variables is a bit complicated and requires some explanation. Here it is:

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

We will deal with the various components of the rule one at a time:

<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. <code>FOO::y</code>). 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 <i>local</i> variable in a method.
<code>variable_name</code>	Again, as for function calls, the name of the variable may be a literal <code>VARIABLE_NAME</code> (<code>\$x</code>), or be given by an expression. The latter possibility is referred to as <i>variable variables</i> in the PHP manual. For example, <code>\$\$x</code> is the variable whose name is currently stored in (another) variable called <code>\$x</code> .
<code>array_indices:(expr?)*</code>	A variable may have one or more array indices, for example <code>\$x[3][5]</code> . The strange construct <code>(expr?)*</code> means: a list of (*) optional (?) expressions. For example, <code>\$x[4][]</code> is a list of two expressions, but the second expression is not given. In PHP, this means use the next available index.
<code>string_index:expr?</code>	Finally, a variable may contain one string index (<code>\$x{5}</code>) that accesses an individual character from a string.

[Converting PHP](#) gives a series of examples of PHP (variable) expressions and their corresponding trees. For now, we are interested only in the `target`.

Accessing Attributes in Non-Objects

Consider the following PHP script

```
<?php
    $x = $y->foo;
?>
```

In this script, `$y` never gets initialised, so it cannot possibly be an object trying to access attribute `foo` should fail. Yet, PHP will accept this script and assign `NULL` to `$x`. If you set `error_reporting` to `E_NOTICE`, PHP *will* give a warning, but continue nevertheless. Suppose you consider this to be a much more serious problem. We could write a transform that replaces all such attributes by

```
<?php
    $x = is_object($y) ? $y->foo : die("Attribute of non-object requested");
?>
```

This is the transform we will write in this tutorial.

Refable Expressions

We are interested in nodes in the tree of type `AST_variable` with a target of type `AST_expr`. Consider the default implementation of the `post_variable` method in the `TreeTransform` class (defined in `translation/transform.h`):

```
void post_variable(AST_variable* in, AST_refable_expr** out)
{
}
```

In other words, the default implementation does nothing. In the [previous tutorial](#) (when we modified function calls), we simply modified some properties of `in`. However, if you want, you can create an entirely new object, and assign that to `*out`, disregarding the previous tree node altogether (`*out` gets assigned to `*in` before the method gets called).

Have a close look at the signature of `post_variable`. Notice that the type of `in` is an `AST_variable` (as expected), but the type of `*out` is actually a more general type, `AST_refable_expr`. A refable expression is an expression that you can create a reference to. Obviously, you can create a reference to a variable (`&$x`), so a variable is a refable expression. But there are more refable expressions; for example, function calls are refable (it makes sense to do `&foo()`). The signature of `post_variable` means that you can assign any refable expression at all to `*out`; it does not necessarily have to be a variable. The reason for this is that wherever you can use a variable in a PHP script, you can also use any other expression, as long as it is refable.

According to the grammar, the construct `x?y:z` is a `conditional_expr`. Unfortunately, a `conditional_expr` is not refable! The reason is not hard to see: what would it mean to create a reference to `($\$x$? 1 : 2)`, for example? So, that means we cannot assign an `AST_conditional_expr` to `*out` in `post_variable`.

So what can we do? Fortunately, function calls *are* refable. So, we could write a (PHP) function such as

```
function ensure_object($obj, $attr)
{
    if(is_object($obj))
        return $attr;
    else
        die("Attribute of non-object requested");
}
```

Then we can replace `$y->foo` by

```
 $\$x$  = ensure_object( $\$y$ ,  $\$y$ ->foo);
```

The Transform

The following code does the transformation we need.

```
#include "transform.h"

class AttributesInNonObjects : public TreeTransform
{
public:
    void post_variable(AST_variable* in, AST_refable_expr** out)
    {
        AST_expr* object;
        Token_class_name* main;
```

```

Token_function_name* fn;
Vector<AST_expr_opt_ref*>* params;

// We are interested in variables with a target only
if(in->target == NULL)
    return;

// And the target should be an expression (not a CLASS_NAME)
object = dynamic_cast<AST_expr*>(in->target);
if(object == NULL)
    return;

// Create a call to %MAIN%::ensure_object()
main = new Token_class_name("%MAIN%");
fn = new Token_function_name("ensure_object");
params = new Vector<AST_expr_opt_ref*>;
params->push_back(object);
params->push_back(in);

*out = new AST_function_call(fn, params, main);
    }
};

```

This code should be fairly self-explanatory. Note that the precise definitions of all the constructors for the AST_something classes can be found in ast.h (in the directory parsing/).

Adding ensure_object

Of course, the transform should also add the ensure_object method itself. The code to do this is mainly an exercise in using the constructors for the AST nodes. It is not difficult, but it is laborious. Future releases of phpc might provide better support for doing this type of thing.

```

void post_php_script(AST_php_script* in, AST_php_script** out)
{
    AST_method* method;
    AST_signature* signature;
    AST_method_mod* method_mod;
    Token_function_name* function_name;
    Vector<AST_formal_parameter*>* params;
    Vector<AST_statement*>* body;
    AST_if* the_if;
    AST_expr* cond;
    AST_statement* iftrue;
    AST_statement* iffalse;
    AST_variable* obj;
    Token_scalar* error_msg;

    // Create method signature
    method_mod = AST_method_mod::new_STATIC();
    function_name = new Token_function_name("ensure_object");
    params = new Vector<AST_formal_parameter*>;
    params->push_back(new AST_formal_parameter("obj"));
    params->push_back(new AST_formal_parameter("attr"));

    signature = new AST_signature(method_mod, function_name, params, false);

    // Create method body
    body = new Vector<AST_statement*>;

    // Condition to the if
    obj = new AST_variable(new Token_variable_name("obj"));
    cond = new AST_function_call("is_object", obj);

```

```
// Body of the if (true branch)
iftrue = new AST_return(obj);

// Body of the if (false branch)
error_msg = Token_scalar::new_STRING("Attribute of non-object requested");
iffalse = new AST_eval_expr(new AST_function_call("die", error_msg));

// The if-statement itself
the_if = new AST_if(cond, iftrue, iffalse);

// Create method and add it to %MAIN%
method = new AST_method(signature, body);
method->statements->push_back(the_if);
in->main->add_member(method);
}
```

Code Readability

The transform in [Tutorial 2](#) was intended to generate a script that could subsequently be edited -- the result should still be a readable script. This is typical of refactoring transforms.

The transform in this tutorial generates unreadable code (with lots of calls to `ensure_object`), but this is irrelevant: the generated script is not meant to be edited any further: it should be run, used or tested; when modifications are to be made to the script, they are made in the *original* script. When the modifications have been made, this transform can be re-applied before the script is run.

What's Next?

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

\$LastChangedDate: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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 implemented 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->function_name->append("_DB");
    }

    void post_function_call(AST_function_call* in, AST_refable_expr** out)
    {
        Token_function_name* name;

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

        // Check for dbx_
        if(name != NULL && name->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?

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`

phc -- the open source PHP compiler

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

\$LastChangedDate: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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:"FUNCTION_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
    | global_declaration | static_declaration
    | unset | declare | try | throw | eval_expr

if ::= expr iftrue:statement* iffalses: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?

global_declaration ::= variable_name*
static_declaration ::= static_var*
static_var ::= VARIABLE_NAME expr

unset ::= variable*
```

```

declare ::= directive+ (statement*)?
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 | SCALAR | constant | instanceof | refable_expr

refable_expr ::=
    variable | pre_op | post_op | array
    | function_call | 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

function_call ::= target function_name params:expr_opt_ref*
function_name ::= FUNCTION_NAME | expr

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

clone ::= expr

```

\$LastChangedDate: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

The Grammar Formalism

The grammar formalism we use is perhaps 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 string; 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`, which will inherit from the STL `string` class (representing the semantic value of the symbol). 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+`). In particular, this type of rule should 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 `as`. 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 `as`. 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: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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%`). [Tutorial 3](#) shows an example.

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

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 undone by the unparser.

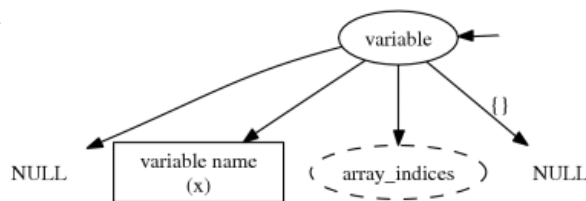
Variables

The grammar rule for variables reads

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

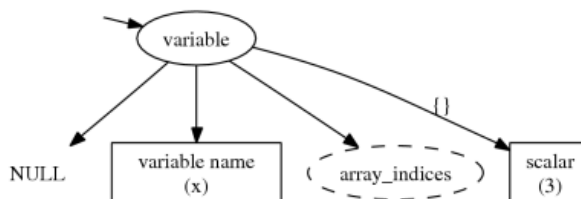
This is probably one of the more difficult rules in the grammar, so it is worth explaining in a bit more detail. [Tutorial 3](#) gives a textual description of the various components of the rule. Here we will give a few diagrams that show examples.

- The simple case: `$x`

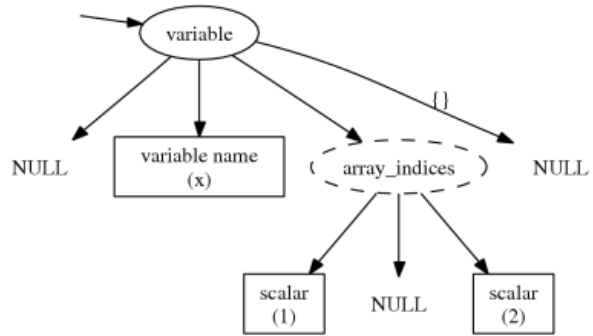


Note that the name of the variable is `x`, not `$x`

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

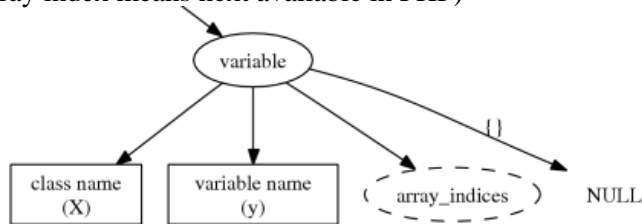


- Using array indices: `$x[1][][2]`



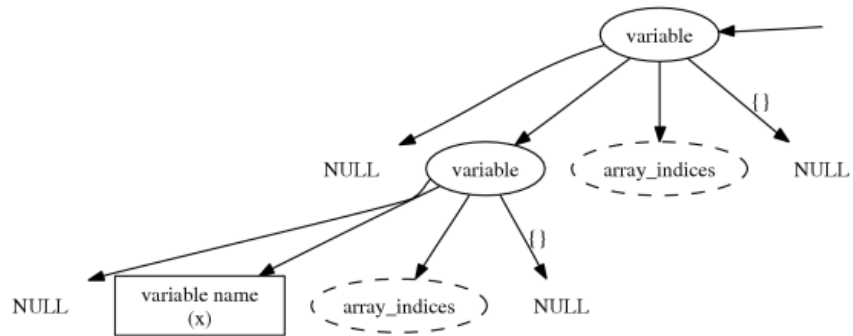
(Note that the empty array index means next available in PHP)

- Class constants: `X : :y`



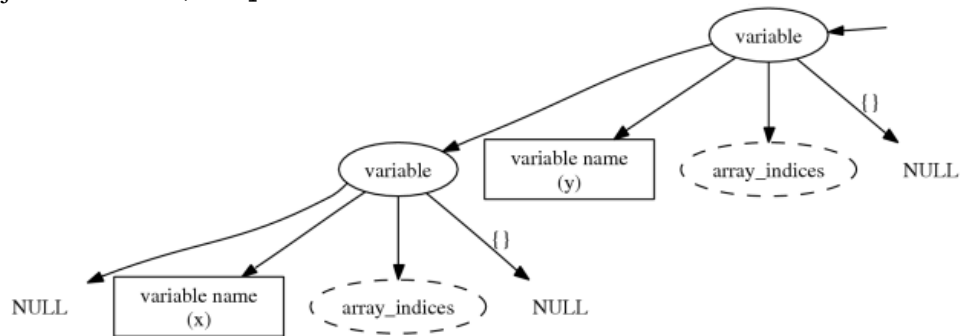
Note that here also the variable name is `y`, not `$y`. The fact that you must write `$x->y` but `X : :$y` in PHP disappears in the abstract syntax.

- Variable variables: `$$x`



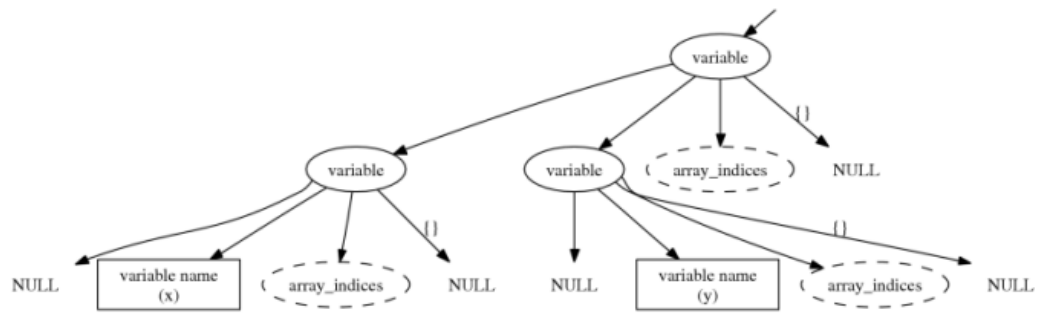
Note how the name of the variable (second component) is now given by another variable.

- 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();
```

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 less than satisfactory; 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](#)).

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

\$LastChangedDate: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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 g()
    {
      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: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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

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 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

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.

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 unparsers 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.

Note that although the current unparsers do not use the phc tree transformation API (see any of the [tutorials](#)), it should in fact be possible, and indeed easier, to implement an unparsers as a tree transform.

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: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

[Home](#) | [Downloads](#) | [Documentation](#) | [Spinoff Projects](#) | [Contact Us](#)

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: 2005-10-27 09:55:57 +0100 (Thu, 27 Oct 2005) \$. Contents © Edsko de Vries and John Gilbert.

Join

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

Email