Design and Implementation of a PHP Compiler Front-end

Edsko de Vries* and John Gilbert
{devriese, gilberj}@cs.tcd.ie

Department of Computer Science
School of Computer Science and Statistics
Trinity College Dublin, Ireland

September 21, 2007

Abstract

This technical report describes the design and implementation of a front-end for `phc`, the open source PHP compiler. This front-end provides an excellent basis for developing tools which process PHP source code, and consists of a well-defined Abstract Syntax Tree (AST) specification for the PHP language, a lexical analyser and parser which construct the AST for a given script, and a visitor and transformation API for manipulating these ASTs.

1 Introduction

PHP [1] is a dynamically typed general purpose scripting language which can be embedded in HTML pages. It was designed in 1995 for implementing dynamic web pages, its name initially standing for *Personal Home Pages*. PHP now stands for the recursive acronym *PHP: Hypertext Preprocessor*. While predominantly used for server-side scripting PHP can be used for writing command line scripts and client-side GUI applications. It may also be embedded into host applications to provide them with scripting functionality.

The main implementation of PHP is free open source software. This provides the *de facto* definition of the syntax and semantics for the language since there is no formal specification. While the end user interface and extensions API for the PHP interpreter are well documented [2, 10], scant documentation exists for the internals of the parser and the Zend engine (which interprets scripts). The lexical analyser is defined using a Flex description of 2000 lines, and the parser a Bison specification of 900 lines. It is difficult to extract any formal specification from either source file because of: unnecessary redundancy and poor structuring of the grammars; convoluted rules which attempt to enforce static checks; embedded semantic actions trying to ensure associativity is appropriately maintained at certain points during parsing; and the requirement for tight coupling between the lexical analyser and parser due to in-string syntax which is not amenable to processing using the traditional lexical analysis and parser interface.

With its widespread deployment PHP requires tools for refactoring, optimizing, obfuscating, checking style and semantics, aspect weaving, and for translation to other languages such as ASP. Many users who have attempted to write software for processing and transforming scripts have resorted to using elaborate regular expressions for pattern matching and other ad hoc approaches due to the difficulty of adapting the PHP front-end to their requirements. Those that have braved the internals of the PHP implementation have generally used the parse trees which represent the concrete syntax productions used in the Bison file to construct a

*Supported by the Irish Research Council for Science, Engineering and Technology.

tree representation of the input [5]. We argue later in this technical report that the parse tree representation of a script which corresponds to that of the PHP language is an entirely inappropriate structure for developers to work with. Without a usable representation of a PHP script's source code the implementation of any of the tools we mentioned above is a serious undertaking.

The open source PHP compiler `phc` [7] is an ongoing project which aims to compile PHP scripts to x86 Intel assembly language. This technical report describes the design and implementation of the front-end of `phc` which consists of a well defined Abstract Syntax Tree (AST) specification for the PHP language, a lexical analyser and parser which construct the AST for a given script and a visitor/transformation API for manipulating these ASTs. It provides an excellent basis for developing compilers as well as any other tools which process PHP source code.

The remainder of this technical report is structured as follows: first we introduce our Abstract Grammar, which defines the structure of PHP scripts as seen by `phc`. Next an overview is given of the visitor and transformation API we provide for manipulating Abstract Syntax Trees whose structure is defined by the Abstract Grammar. We then outline a number of projects which could benefit from our existing front-end, and which would be rather difficult to develop entirely from scratch. Finally we finish with a description of plans for future work and our conclusions.

2 Abstract Grammar

In this section we discuss the abstract grammar used by `phc` to represent PHP scripts. We explain the decisions faced while designing the grammar, highlight some the difficulties with parsing and conclude with a few comments on unparsing abstract syntax back trees to PHP syntax.

2.1 Motivation

Consider the following simple PHP program.

```
<?php
   printf("Hi");
?>
```

This program gets represented by `phc` as the AST shown in Figure 1. The interpretation of this structure should be self-explanatory and can be read directly from the figure: we have a PHP script containing a single statement which evaluates an expression; the expression is a method invocation; the name of the method is `printf`; and we pass in one parameter, the string "`Hi`".

While this may seem like an obvious representation, no such representation exists in the PHP implementation, nor does this representation correspond to any official definition of PHP. As mentioned in the introduction, no official grammar for PHP exists. The only "grammar" that is available is the PHP parser, written in Bison [9]. While the Bison grammar defines a representation of sorts, we claim that this representation is not particularly useful as a language definition or as an internal representation. For comparison, Figure 2 shows the parse tree that would result from the same program if constructed using the Bison grammar from PHP. Not only is the tree much larger, but it also contains many nodes that do not correspond to any conceptual notions (for example, what exactly is an *unticked_statement* or an *r_variable*?).

Moreover, the structure of the parse tree is rather convoluted. Consider the subtree for *expr*. The root node of this subtree (*expr*) corresponds to the *AST_eval_expr* node in the AST, but the node immediately below the *expr* is an *r_variable*. Only further down the parse tree does it become apparent that the expression is not a variable at all, but a function call. In the AST the root node of the expression is of type *AST_method_invocation*, which seems a more logical choice. The parse tree also does not always show the correct associativity of operators—we consider such an example in Section 2.3.

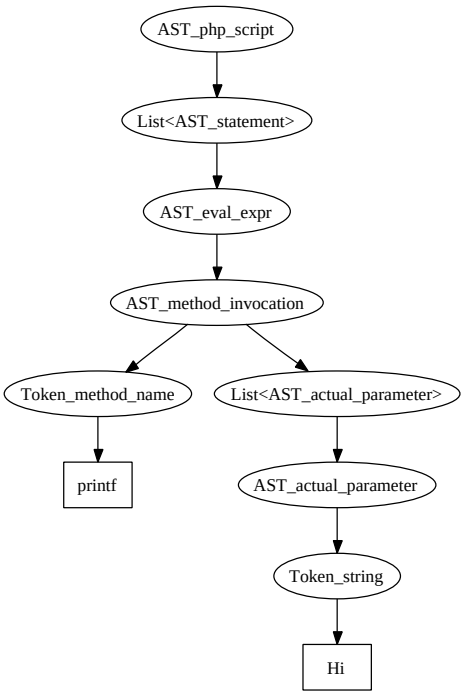

Figure 1: Abstract syntax tree for `<?php printf("Hi"); ?>`

Some may feel that the parse tree suffices as a representation [5]. However, since the choice of language representation has a profound impact on a large part of the compiler, we feel that the parse tree is too complicated and illogical a structure to work with. For that reason, we spent a lot of time devising an intuitive and clear definition of PHP in the form of an AST.

2.2 Design

The *abstract grammar* which defines the structure of our ASTs is shown in full in Appendix A. In contrast to the definition of the Bison grammar, which is driven by implementation concerns, the design goals for the abstract syntax were conciseness and clarity. We tried to keep the number of concepts to a minimum, and felt it was imperative to make every concept correspond to a conceptual notion familiar to a PHP programmer.

The design of the abstract grammar was also constrained by the underlying formalism. Briefly, every rule must be either a sequence of terminal and non-terminal symbols (such as the rule for *method*), or a disjunction of terminal and non-terminal symbols (such as the rule for *member*). Combinations of sequencing and disjunction of symbols are not allowed. This guarantees a straightforward mapping from the abstract grammar to a C++ data structure [6].

The top-level structure of the grammar (*php_script*) is defined to be a list of statements. In other words, a PHP script is considered to be a list of statements. While this corresponds to the top-level structure of the PHP Bison definition, it was not our first choice. In our initial design, the top-level structure of the PHP script was considered to be a list of class definitions. Function definitions that were not originally part of any class were treated as static functions in a special class called `MAIN`, and code that was not part of any function definition was collected in a static function named `run` in `MAIN`. Since the language specification had to include the notion of a class definition (including static definitions), this reduced the number of concepts in the abstract grammar. However, while this potentially simplifies the design of the code generator, not all scripts can be transformed into this format. For example, consider

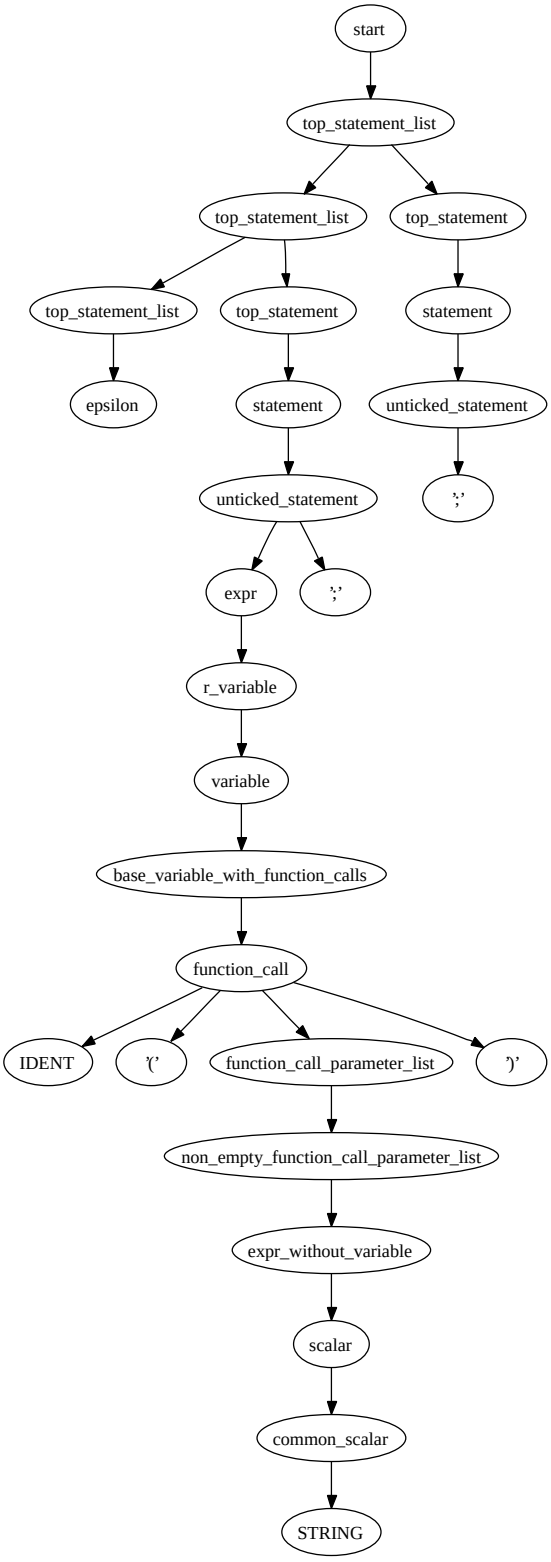

Figure 2: Parse tree for `<?php printf("Hi"); ?>`

```
<?php
   include "common.php";

   class D extends C // class C defined in common.php
   {
      // ...
   }
?>
```

If this script is changed so that the `include` statement becomes part of a function `run` in `MAIN`, class *C* will no longer be defined (included) before class *D* is defined, and the program becomes invalid (the interpreter will report an error *"Cannot inherit from undefined class C"*).

In a similar vein, our initial design did not include an explicit notion corresponding to the `global` statement of PHP. The scoping rules of PHP are very simple. There are only two scopes: the global scope; and the local scope within a function (compare to the programming language C, where every block of statements enclosed in curly brackets introduces a new scope). By default, a variable used in a function is considered to live in the local scope. For example,

```
<?php
   function f()
   {
      $x = 2;
   }

   $x = 1;
   f();
   echo $x;
?>
```

will print 1 (not 2) , because the `$x` used inside `f` is local to `f`. The `global` keyword can be used to "import" a variable from the global scope into the local scope:

```
<?php
   function f()
   {
      global $x;
      $x = 2;
   }

   $x = 1;
   f();
   echo $x;
?>
```

will print 2. PHP provides an alternative way to access global variables from a local scope: the `$GLOBALS` array. Every global variable is considered to be an element of `$GLOBALS`, and `$GLOBALS` itself is a so-called "superglobal": it is available in every scope. Therefore, we can rewrite the previous example as

```
<?php
   function f()
   {
      $GLOBALS["x"] = 2;
   }

   $x = 1;
   f();
   echo $x;
?>
```

To reduce the number of concepts in the grammar, we thought we could represent

```
global $x;
```

as

```
$x =& $GLOBALS["x"];
```

which makes the local `$x` a reference to the global `$x`. While this corresponds very closely to the "official" semantics of `global` (insofar as there is one), we learnt that this translation is in fact incorrect: while `$GLOBALS` is a superglobal and therefore available in every scope, it is possible to remove `$GLOBALS` from a scope if desired. For example,

```
function f()
{
   unset($GLOBALS);
   $GLOBALS["x"] = 2;
}
```

first removes `$GLOBALS` from the local scope, *making any subsequent uses of `$GLOBALS` refer to an ordinary local array which happens to be called `$GLOBALS`*; thus, calling `f` has no effect on the global variable `$x`. However, the following function `g` *does*:

```
function g()
{
   unset($GLOBALS);
   global $x;
   $x = 2;
}
```

Even though the code generator still implements `global` by making `$x` a reference to the global `$x`, this example clearly shows that the translation using `$GLOBALS` is invalid: `$GLOBALS` may not be in scope. The upshot of this and the previous example is that it dangerous to try to reduce the number of concepts in the grammar by representing some PHP constructs in terms of others.

Nevertheless, the number of concepts in our abstract grammar is significantly smaller than the number of concepts in the PHP Bison definition. There are various reasons for this. For example, the grammar formalism used by Bison does not allow the direct definition of lists of anything (such as a list of statements), which can only be encoded using recursion (this is clearly shown in Figure 2: *start* does not have two children, corresponding to two statements, but contains a *top_statement_list* which, in a left-recursive fashion, encodes a list of statements).

A more important reason is that the Bison grammar is implementation driven as mentioned above. For example, the Bison grammar distinguishes between an *r_variable* and a *w_variable*. The only difference between the two (both correspond to variables or function calls) is their use: as an lvalue (to the left of an assignment operator) or as an rvalue (to the right of an assignment) respectively. While this distinction is useful for some applications, it complicates others. For example, a refactoring tool that renames variables does not necessarily want to make a distinction between variables on the left or right of an assignment operator, and thus would benefit if the grammar uses the same concept to represent both.

Finally, the Bison grammar imposes some semantic limitations. For example, function arguments can have default values, but these values must be static. So,

```
function f($x = 1) { ... }
```

is a valid definition, but

```
function f($x = $a) { ... }
```

is not, because `$a` is not a static value. This restriction is enforced in the grammar definition—there are separate grammar rules for expressions and static expressions. The definition for static expressions is not completely trivial because a static expression could be an array:

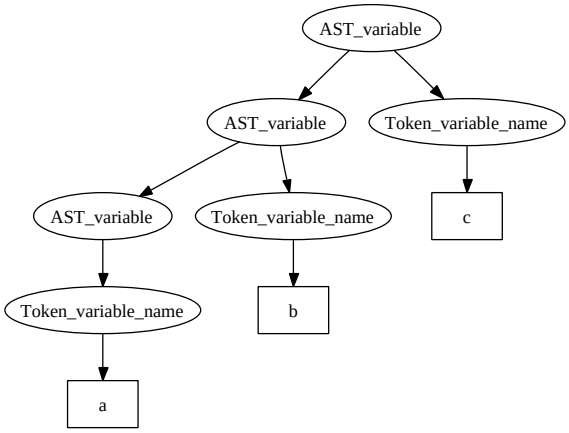

Figure 3: Abstract syntax tree for the expression `$a->b->c`

```
function f($x = array(1,2)) { ... }
```

which means that separate concepts are needed for "static arrays" (arrays containing only static elements, which could in turn be static arrays) and regular arrays. In the design of our AST we do not enforce these semantic checks: we have only one definition of expressions, including only one definition of arrays since most semantic checks cannot be enforced by a context free grammar anyway (the grammar formalism simply is not powerful enough). It seems arbitrary to encode some semantics checks but not others (in `phc` semantic checking is a separate phase). Moreover, since `phc` is a compiler rather than an interpreter, we can apply optimization techniques such as constant folding and partial evaluation to identify complex expressions with constant values: thus, we can potentially accept more programs.

2.3 Parsing

Once the AST had been defined we had two options: write a new parser to build the AST from scratch; or reuse the existing PHP parser. The first option is a non-trivial undertaking. The abstract grammar by its very nature is not suitable for use as a parser definition. It is highly ambiguous and leaves out all syntactic sugar (semicolons, arrows, etc). Moreover, it would be next to impossible to verify that our new parser accepted the same language as the original PHP parser.

Thus, the only realistic option was to reuse the existing PHP parser stripped of all its semantic actions and modified to output a representation of its input corresponding to our abstract grammar. This task was relatively simple if tedious, but there were a few gotchas along the way. The first of these was that the recursive structure of the Bison definition does not always correspond to the required associativity of operators. For example, consider the expression

```
$a->b->c
```

The correct reading of this expression is

```
($a->b)->c
```

as can be readily verified by writing a small test script. In words, this expression accesses field `c` of `$a->b`, as opposed to field `b->c` of variable `$a`. The arrow operator is left associative, as reflected in the AST corresponding to this expression (Figure 3). But the Bison grammar definition (and, correspondingly, the parse tree) for this expression's structure is right recursive. Indeed, the parse tree for the same expression (Figure 4) suggests that the arrow operator is right associative rather than left associative.

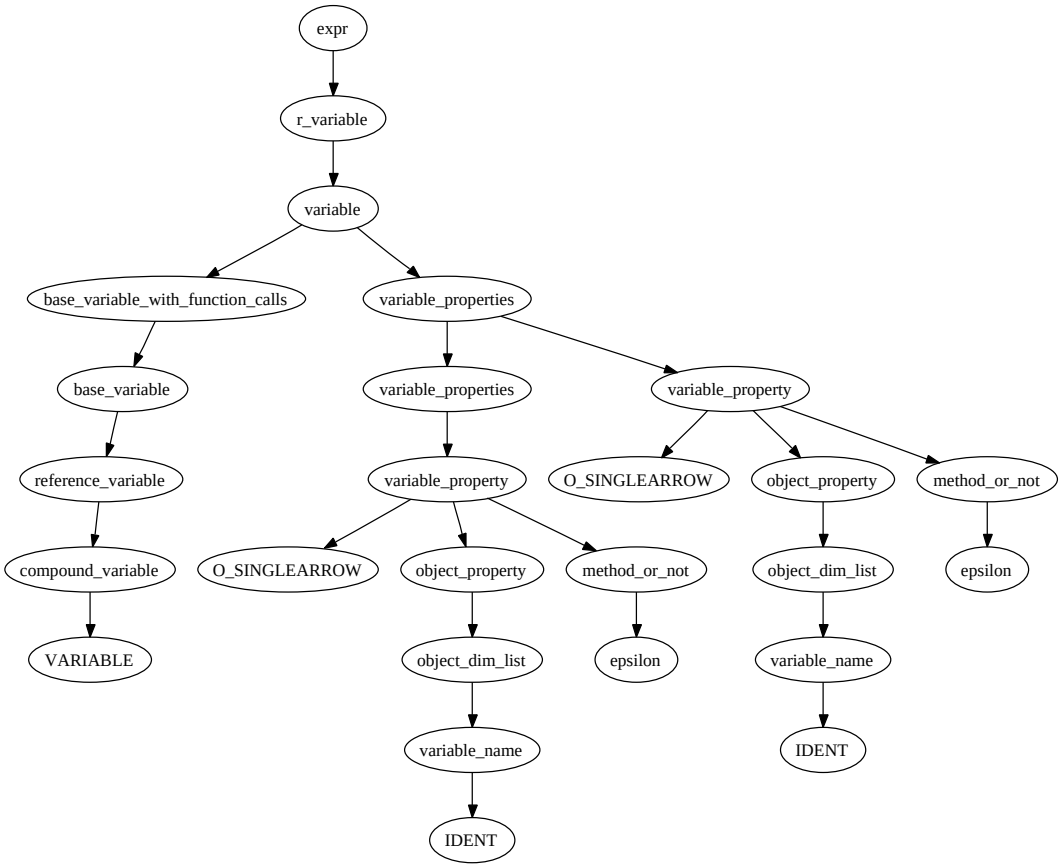

Figure 4: Parse tree for the expression `$a->b->c`

Another problem was mentioned in section 2.1: the grammar rule for `r_variable` must synthesize either an *AST_variable* or an *AST_method_invocation* depending on the AST subtrees already built by the children of `r_variable`. We solved this problem by associating an optional list of function arguments with every variable. When a variable's node has a non-NULL list of function arguments, it is transformed from an `AST_variable` into an `AST_method_invocation`. Naturally, this action is hidden in the parser and by the time the AST has been built, variables are represented by `AST_variable` nodes and method invocations by `AST_method_invocation` nodes.

However, the most difficult problem encountered when processing PHP is lexical analysis. Here complications arise due to PHP's "in-string" syntax, where the contents of a string are interpreted. Consider

```
<?php
   $a = 5;
   echo "a is $a\n";
?>
```

Even though the reference to `$a` appears within the string it will be interpreted, hence this script will print "`a is 5`". PHP distinguishes between two kinds of in-string syntax: simple; and complex. Simple in-string syntax consists of a small number of constructs that can be used directly inside strings. The script shown above is an example of such simple in-string syntax. Other examples include `$a[1]` (array indexing) and `$a->b` (object field access). Complex in-string syntax is denoted by {`$`...}, and essentially provides a way of escaping temporarily to regular PHP syntax. As suggested by their names, supporting simple syntax is relatively straightforward (it suffices to add regular expressions for each of the provided constructs to the lexical analyser), while complex syntax is more difficult.

The general approach we take is to treat in-string syntax as an implicit concatenation operation—the `echo` statement in the previous example is implicitly treated as

```
echo "a is " . $a . "\n";
```

Thus, neither our abstract grammar nor the parser has a notion of in-string syntax. In this example, when a user writes

```
echo "a is $a\n";
```

the string of tokens returned to the parser by the lexical analyser is

```
IDENT(echo), STRING("a is "), OP(.), VAR(a), OP(.), STRING("\n"), OP(;)
```

This is not completely trivial to implement. When the lexical analyser recognizes a simple in-string pattern, it pushes a number of tokens onto a local stack; in this case, the initial segment of the string "`a is `", a concatenation operator, the variable `a`, and another concatenation operator. It then moves into a state `RET_MULTI`. Subsequent calls to the lexical analyser while it is in this state will return the next token from the local stack without matching any input. Once the stack is exhausted, the lexical analyser returns to its previous state (in this example, the state recognizing strings).

This technique can be used for all simple patterns, but unfortunately does not scale to complex in-string syntax. For example, consider

```
echo "{double index: {$a{$b{1}}}}\n";
```

While this example is perhaps slightly contrived, it clearly demonstrates the problem—the outer braces are part of the string itself; the braces around the variable delimit the complex in-string pattern, and the remaining braces delimit an array index. Ideally the lexical analyser should recognize that the string continues after the final closing brace, but to know which brace is the last one we need to be able to match braces: the canonical example of a non-regular problem [12].

There are two possible solutions to this problem: we could embed a small syntax analyser within the lexical analyser; or we could couple the existing syntax analyser and the lexical analyser more closely.

Both of these solutions are slightly inelegant—they break the clean separation between lexical and syntax analysis. A more "principled" solution would be to use a scannerless parser and do away with the separation completely. This is the approach taken in PHP-front [4], which uses the advanced SGLR parser from the ASF+SDF framework [8]. The downside with that solution is, of course, that the original PHP Bison file cannot be used, making it hard to be sure the parser accepts the same language as the "official" parser. Moreover, verifying the impact of minor changes in the "official" grammar which occur when new versions of PHP are released means that maintaining a parser developed from scratch would be an ongoing, time consuming, burden. In fact the PHP-front developers report various difficulties trying to be completely compatible with the official parser [Eric Bouwers, personal communication].

PHP itself chooses the first option, and embeds a small hand-written parser into the lexical analyser which only accepts a small subset of PHP. Although this solution avoids coupling the syntax and lexical analysers, it increases the complexity of the lexical analyser. We chose the second option instead. When the lexical analyser recognizes the start of a complex construct ({$), it returns the initial segment of the string to the parser followed by a "special" concatenation operator (`O_MAGIC_CONCAT`). It then pushes the dollar sign ($) back into the input and reverts from the "parsing a string" state to the "parsing PHP" state. As far as the lexical analyser is concerned, we have finished parsing the string—it is up to the syntax analyser to instruct the lexical analyser to return to the "parsing a string" state.

In the parser we have two grammar rules for concatenation: one dealing with the standard concatenation operator; and one dealing with `O_MAGIC_CONCAT`. The rule for standard concatenation simply builds part of the AST:

```
| expr '.' expr
    {
      $$ = NEW(AST_bin_op, ($1, $2, $3));
    }
```

while the rule for `O_MAGIC_CONCAT` builds the same syntax tree (semantically both operators are equivalent), its implementation is slightly different:

```
| expr O_MAGIC_CONCAT expr '}'
    {
      $$ = NEW(AST_bin_op, ($1, ".", $3));
      context->return_to_complex_syntax();
    }
```

Compared to the previous rule, there are two differences. The rule includes an additional closing brace, which—though it may look out of place—is the closing brace that terminates a complex in-string construct. By adding the closing brace here, the parser matches the braces for us, hence we know when to instruct the lexical analyser to return to parsing a string—established by the call to `return_to_complex_syntax()`.

This solution keeps both the lexical analyser and the syntax analyser simple, and even though it couples both analysers, the interaction is limited to a single call from the syntax analyser to the lexical analyser.

2.4 Unparsing

Unparsing is the problem of converting the AST back into PHP syntax. Writing an unparser that outputs valid PHP syntax of some form (even pretty-printed) is straightforward, but writing an unparser suitable for use in a refactoring tool is very difficult. Ideally, such an unparser should reproduce a program in its original form with the layout and comments (including whitespace) positioned in exactly the same way as in the programmer's original code. We make an effort at positioning comments, but have not yet been able to come up with a satisfactory solution to dealing with layout.

The `phc` unparser essentially re-does the layout from scratch, mostly ignoring the user's original layout. It makes an attempt at producing readable output, and will position the user's comments (approximately) in

the right place. The parser records a few attributes with each node. For example, any part of an expression can be marked as `starts_line`. This attribute is taken into account by the unparser: if a programmer writes

```
$x =
   "SELECT " . $a .
   "FROM " . $b .
   "GROUPBY " . $c;
```

it will be unparsed as

```
$x =
     "SELECT " . $a .
     "FROM " . $b .
     "GROUPBY " . $c;
```

While layout is not preserved identically (the whitespace varies), at least the general structure is preserved. Attributes are also used to unparse some PHP constructs not explicitly represented in the AST in their original form. For example, the abstract grammar does not have a construct for `elseif` (which gets represented as a nested `if`), but the unparser will output a nested `if` as an `elseif`, if it is has the attribute `is_elseif` set.

The lexical analyser uses a local variable `attach_to_previous` to indicate whether a comment should be associated with the next or the previous token. Initially `attach_to_previous` is set to false, and it is reset to false again after every line break in the input. It is set to true when the lexical analyser encounters a semicolon. This means that in

```
<?php
   // First comment
   f(); // Second comment

   // Third comment
   g();
?>
```

the first and second comment will be associated with the call to f, while the third comment will be associated with g. In fact, for this example, the unparser will be able to reproduce the exact layout, including the blank line in between the call to f and the comment before g. How? Keeping track of whitespace in general and associating whitespace with the relevant nodes in the AST is quite a difficult problem. However, blank lines provide the programmer with a useful device to indicate logical units in a program's text, and it is important that the unparser is able to reproduce them. The mechanism for doing this has already been described: we simply treat a blank line as a comment! Thus the call to g actually has *two* comments associated with it—the blank line and the "third" comment[1]. Thus, the only edge case we have to deal with are comments at the end of a block, for example:

```
if($cond)
{
  echo "yes";
}
else
{
  // do nothing
}
```

Here, there is no node with which to associate the last comment. To solve this problem we introduce a `NOP` (do-nothing) statement at the end of every block (if there are left-over comments), and the comment

[1]This approach was suggested to us by Tim Van Holder on the `bison-help` mailing list.

is associated with the NOP. This does not technically violate our design principle that the abstract grammar contains only concepts familiar to the PHP programmer, because PHP has a NOP statement (represented by a single semi-colon), and the concept of a NOP statement is useful in other situations.

3 Traversal API

Although representing the PHP script using our abstract grammar is an important step towards providing a usable framework for processing PHP, it remains a long way from being easy to use. Without additional tool support, traversing the AST would involve a large amount of boilerplate code, and this would make even the simplest transformations long and cumbersome.

To solve this problem phc provides two APIs: a tree visitor API and a tree transformation API. Both take the form of a C++ class which is to be inherited from for particular transformations. There are two important differences between the visitor and transformation APIs.

1. The visitor API supports "generic" methods: methods that get applied to every node in the tree or to a subset of nodes, such as all statements. As an extreme example, phc provides an XML unparser (which outputs the AST in XML format). The XML unparser is defined in terms of the tree visitor API, but only defines a few methods. In essence, for every node of type AST_x in the AST, it outputs <AST_x> before the children of the node and </AST_x> after the children have been unparsed.

2. The tree visitor API is limited in the way in which it can modify the AST: it can modify individual nodes in the tree but it cannot modify the structure of the tree in any way. However, using the tree transformation API it is possible to change the tree structure (for example, remove nodes from the tree or replace a single statement by a list of statements). The transformation API is designed in such a way that, barring the use of C++ type casts, it is impossible to write transformations that invalidate the integrity of the tree with respect to the abstract grammar. For example, an expression can only be replaced by another expression but not by a statement. However, the tree transformation API does not provide generic methods, so for example it is not possible to define a transform that replaces every node in script (that is to say, it is possible, but you would have to explicitly replace every type of node)[2].

Due to the way we implement in-string syntax (Section 2.3), the AST may have many extraneous concatenation operations. For example, starting with

```
echo "$a";
```

we will get an AST which represents

```
echo "" . $a . "";
```

As an example of what can be done, Figure 5 shows a transform that cleans up these structures so that the above code would be replaced by

```
echo $a;
```

after the transformation has been completed. The transform overrides only one method from the parent class (AST_transform): post_bin_op, a method that is applied to all nodes of type AST_bin_op (binary operators) after each node's children have been processed. In the body of the transform we also use some other infrastructure provided by phc: pattern matching on (parts of) the AST. First we check if the left

[2] In a select few cases it is possible to write generic transforms. For example, it is possible to write a transform that transforms every statement without having to write explicit transforms for if-statements, while-statements, etc. This is however due to a technical implementation detail of the API and only works for a few types.

```
AST_expr* Remove_concat_null::post_bin_op(AST_bin_op* in)
{
   Token_string* empty = new Token_string(new String(""), new String(""));
   Wildcard<AST_expr>* wildcard = new Wildcard<AST_expr>;

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

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

   return in;
}
```

Figure 5: Removing extraneous concatenations

operand of the concatenation operator is the empty string, and if it is we replace the binary operator by its right operand. If not, we check if the right operand is the empty string and if so we replace the binary operator by its left operand. If neither pattern matches we leave the binary operator unchanged.

One interesting but slightly subtle point of this transform is the use of `post_bin_op` as opposed to `pre_bin_op` which gets applied to all binary operators *before* their children have been processed. What would happen if we used `pre_bin_op` instead of `post_bin_op`? Suppose we start with the same example as depicted in figure 6. The root of the tree is the right-most `AST_bin_op` node. Using `pre_bin_op` the transform gets invoked on that node before its children have been processed. This node matches the second pattern (right operand is the empty string), so we replace the node by its left operand (the second `AST_bin_op`). Finally, we transform *each of the children* of the node. But the node has already been replaced by the second (left) binop, so we are now transforming the *children* of the second binary operator rather than the binary operator itself. Hence the second concatenation operation will not be removed.

When we initially encountered this problem, we changed the API so that the pre-transformation was invoked *again* if the first pre-transformation returned a new node (where "new" was defined in terms of pointer equality). In the particular example of the `Remove_concat_null` transform this is (arguably) a better solution because it means we can implement the transform using a pre-transform or a post-transform

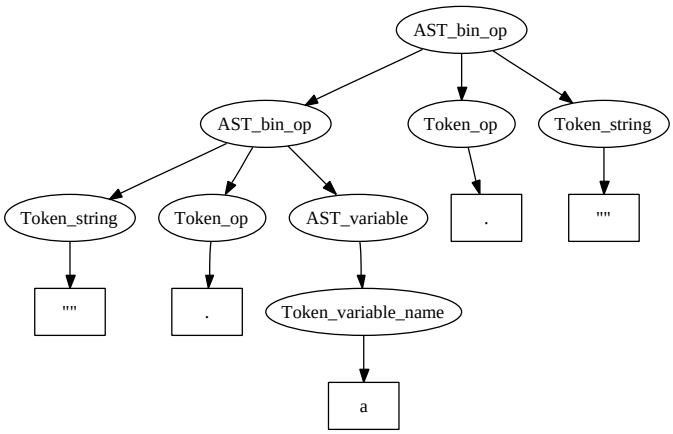

Figure 6: Tree for `"$a"`

with the same result—which is perhaps what one would expect. After all, the problem described in the previous paragraph is rather subtle. However, we found that this created more problems than it solved (for example, it can easily result in non-terminating transforms), so the newer versions of `phc` no longer have this behaviour (although programmers can easily simulate it by manually invoking the pre-transform again on a transformed node).

As a second example of a transform, consider the code listing shown in Figure 7. This transformation replaces a statement of the form

```
include("a.php");
```

by the entire contents of the file `a.php`. We can do this since the top-level construct of a script is a statement list, and it is always safe to replace a statement by an arbitrary number of statements. The basic structure of the transform is the same as in the previous example. We set up a pattern to match calls to `include`. When the pattern match succeeds, we extract the value of the wildcard, try to read and parse the file with that name, and then replace the include by the contents of the file. Note the slightly different signature: `pre_bin_op` returns a node of type `AST_expr`—that is, it is safe to transform a binary operator into any other expression. Instead, `pre_eval_expr` gets passed a list of statements and returns void: once again, it is safe to transform a statement into any number of statements. The list is pre-initialized by `phc` to the empty list, and leaving it empty in the transform causes the statement to be deleted. The default implementation will simply add `in` to the list, leaving the statement untouched. In the `Expand_includes` transform, we add all statements in the parsed file to the list, but leave out `in`, thus replacing `in` by the contents of the file to be included.

4 Future Work and Conclusions

In this technical report, we have described the design and implementation of a front-end for `phc`, the open source PHP compiler.

The grammar formalism we have adopted for specifying the abstract syntax of PHP allows us to perform a mechanical translation from the specification to a C++ class structure to implement and support the tree visitor/transformation framework described above. Initially this was coded by hand, but when the structure of the AST began to stabilize and an increasing amount of infrastructure had been built upon it, the task of accommodating even minor refactorings of the abstract grammar grew unwieldy. To tackle the issue we developed a tool called `maketea`, written in Haskell, that can take our abstract grammar specification and automatically turn it into the appropriate class hierarchy and visitor/transformation APIs. Currently we are in the process of improving the robustness of this tool and have spun it off as a separate project [6].

Although our unparser is currently usable for many applications, it is not powerful enough for use in refactoring tools. We still need to find a solution whereby we can reproduce the user's original code using their original layout if no refactorings have been applied, or using a layout which resembles the user's as much as possible when the program structure has been changed (for example, when statements have been inserted or removed). This is a difficult problem, and one which cannot be fully solved; we will have to rely on heuristics.

Rather than immediately design an intermediate representation (IR) appropriate for low level optimizations within `phc`, we have opted instead to translate directly into C code. This code makes calls into the Zend run-time so that we are fully compatible with all existing PHP libraries, which is where the true power of the PHP scripting language lies. We intend to develop a three-address code IR which abstracts over the C code we currently generate, giving us a suitable representation upon which to apply traditional compiler optimizations. This will be an important first step in obtaining an efficient implementation of PHP, which by its nature is highly dynamic. For example, a simple assignment statement `$a = $b;` generates a rather large body of code—mostly to support the reference counting memory model used in the Zend run-time. It should be possible to remove a lot of of this overhead by applying standard compiler optimizations to the IR.

```
class Expand_includes : public AST_transform
{
private:
   Wildcard<Token_string>* filename;
   AST_method_invocation* pattern;

public:
   Expand_includes()
   {
      filename = new Wildcard<Token_string>;
      pattern =
         new AST_method_invocation(
            NULL,
            new Token_method_name(new String("include")),
            new List<AST_actual_parameter*>(
               new AST_actual_parameter(false, filename)
            )
         );
   }

public:
   void pre_eval_expr(AST_eval_expr* in, List<AST_statement*>* out)
   {
      // Check for calls to include
      if(in->expr->match(pattern))
      {
         // Matched! Try to parse the file
         AST_php_script* php_script = parse(filename->value->value, NULL, false);
         if(php_script == NULL)
         {
            cerr
            << "Could not parse file " << *filename->value->value
            << " on line " << in->get_line_number() << endl;
            exit(-1);
         }

         // Replace the include by the statements in the parsed file
         out->push_back_all(php_script->statements);
      }
      else
      {
         // No match; leave untouched
         out->push_back(in);
      }
   }
};
```

Figure 7: Expanding include directives

After that we will need to think about more specific optimizations. One important goal will be to minimize the number of calls to the Zend run-time. For example, an integer must normally be wrapped in a so-called "zval", the data structure used by the Zend run-time to store variables. Incrementing the integer entails a call into the Zend API which will check whether the zval is actually an integer, check if it happens to be part of a "copy-on-write" set, etc. Of course, incrementing an integer should be as simple as a single machine instruction to update the value of a register. By analysing the IR, it should be possible to remove a lot of the overhead associated with the Zend run-time. This is not an easy task. Incrementing an integer value could cause it to overflow, and PHP semantics dictate that the integer then becomes a float. So there is plenty of scope for reducing the overhead that arises due to PHP's dynamic properties, but it is these properties that will present as challenges during optimization.

The front-end of `phc` provides an excellent basis for developing tools which process PHP source code and has already been adopted for this purpose. To date we are aware of two projects: `Plumhead` [11] compiles PHP to bytecode for the Parrot virtual machine [13], and `PHT` [3] extends PHP with special constructs which can be used to guarantee that the output of a PHP script is valid HTML or XML. We envision that other software such as refactoring tools, style checkers, aspect weavers, script obfuscators, script optimizers and pretty printers would be much easier to develop based on `phc` than starting from scratch. Already `phc` has been used with coursework in an advanced compiler design course in the Department of Computer Science, Trinity College Dublin. The project has also formed the basis for a postgraduate students PhD research on compilation for dynamic languages.

Acknowledgements

We thank David Abrahamson for his careful proofreading of this report, and the early adopters of `phc` who submitted bug reports, patches, ideas and encouragement. In no particular order they are David Abrahamson, Sven Klemm, Andras Biczo, Daniel Barreiro, Andreas Korthaus, Conor McDermottroe, Daniel Fabian, Dan Libby, Bernhard Schmalhofer, Matthias Kleine and Paul Biggar. The work of John Gilbert was supported by a Trinity College Dublin Postgraduate Award.

A Abstract Grammar for PHP

A.1 Top-level structure

```
php_script ::= statement* ;
```

A.2 Statements

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

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

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

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"?  ;

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

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

static_declaration ::= VARIABLE_NAME expr? ;
global ::= variable_name ;
unset ::= variable ;

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

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

eval_expr ::= expr ;

nop ::= ;
```

A.3 Expressions

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

literal ::= INT<long> | REAL<double> | STRING<String*> | BOOL<bool> | NULL<> ;

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

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

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?* ;
variable_name ::= VARIABLE_NAME | reflection ;
reflection ::= expr ;

target ::= expr | CLASS_NAME ;

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

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

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

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

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

clone ::= expr ;
```

A.4 Additional Structure

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

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

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

References

[1] PHP: Hypertext preprocessor. http://www.php.net.

[2] PHP manual. http://www.php.net/manual/en/.

[3] BARREIRO, D. PHT – PHp with embedded HTml. http://www.satyam.com.ar/pht/.

[4] BOUWERS, E., AND BRAVENBOER, M. PHP-Front. http://www.program-transformation.org/PHP/PhpFront.

[5] CANDILLON, W. PHP aspect. http://phpaspect.org/.

[6] DE VRIES, E., AND GILBERT, J. maketea. http://www.maketea.org.

[7] DE VRIES, E., AND GILBERT, J. phc – the open source PHP compiler. http://www.phpcompiler.org/.

[8] DEURSEN, A. V., HEERING, J., AND KLINT, P., Eds. *Language Prototyping: An Algebraic Specification Approach: Vol. V*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[9] FREE SOFTWARE FOUNDATION. Bison. http://www.gnu.org/software/bison/manual/.

[10] GOLEMON, S. *Extending and Embedding PHP*. Sams, 2006.

[11] SCHMALHOFER, B. Plumhead. http://rakudo.org/parrot/index.cgi?plumhead.

[12] SIPSER, M. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[13] THE PERL FOUNDATION. The parrot virtual machine. http://www.parrotcode.org/.