DEVELOPMENT OF PHP TO UPML TRANSFORMER

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE

DOCTOR OF EDUCATION

BY

ZILI HU

MRS. DOLORES ZAGE – ADVISOR

BALL STATE UNIVERSITY

MUNCIE, INDIANA

JULY 2013
Keywords: Software Metrics, Software Quality Metrics, Programming Languages, Programming Language Translation, Extensible Markup Language (XML), Unified Programming Markup Language (UPML)
To

My almighty God

My wife, my daughter, and my parents

For

Your Love and Support
Acknowledgments

I first appreciate my advisor, Professor Dolores Zage, for her support and guidance, topic initiative, kind encouragement, hands-on assistance in trivial document preparation, course teaching and mentoring, help in my job-hunting, and her model of hard work in academic research.

I appreciate Dr. Wayne Zage for the opportunities of involvement of academic activities in the showcase conferences and weekly research discussions, his leadership in software and security research, and for his kind and encouraging smiles. I appreciate Dr. Paul Buis for his leadership in the academic program, hands-on help in my academic progress, and helpful suggestion in the selection of a thesis topic. I appreciate Dr. Yong Zhang and Dr. Jason Yang for their careful questioning and constructive suggestions.

I appreciate Dr. Bill Cupp, Professor Cindy Cupp, Dr. Ken Bielen, Dr. Roberto Ramos, Dr. Melvin Royer, Dr. Dennis Brinkman, Dr. Steve Conrad, Dr. Connie Lightfoot, Dr. Gail Green, Dr. John Lakenon, Dr. Richard Barnhart, Dr. Daniel Neumann, Dr. Daniel Kiteck, Mrs. Cheryl Edris, Mrs. Sara Atkinson, and many others, as my colleague and friends at Indiana Wesleyan University, for their companionship, encouragement, collaboration, hands-on help, assistance of paperwork, and prayers in my job and my life. I appreciate my spiritual sisters and brothers Hope Robinson, Leo Robinson, Jason Yang, Yingxia Zhang, Jie Chen, He Wang, Xiaotong He, etc. at Muncie Chinese Christian Fellowship (MCCF), Dr. Kuyen Li, Professor Sherry Li, Mrs. Dorothy Weatherly, etc. at Beaumont First Baptist Church, and Dr. Mei Zhong, for their friendship and help in my Life. I appreciate all those with friendship in my life.

I am eternally grateful to my dear parents for their love, support, and guidance.
# Table of Contents

1. Introduction ......................................................................................................................... 8  
   1.1. Software Metrics ........................................................................................................... 8  
   1.2. Translation of Programs in Various Programming Languages ........................................ 16  
   1.3. Issues in the State-of-the-Art ...................................................................................... 19  
   1.4. Contribution of this Work ............................................................................................ 22  
   1.5. Road Map of this Thesis .............................................................................................. 27  

2. Related Work ......................................................................................................................... 28  
   2.1. Software Metrics ........................................................................................................... 28  
   2.2. Programming Language Translation ............................................................................. 28  

3. Analysis of Programming Languages and Programming Techniques ............................... 33  

4. Implementation of UPML ..................................................................................................... 35  
   4.1 Design Consideration ...................................................................................................... 36  
   4.2 UPML Representation of Programming Techniques ...................................................... 37  
      4.2.1 UPML File ................................................................................................................ 38  
      4.2.2 HTML Code ............................................................................................................. 38  
      4.2.3 Program Code .......................................................................................................... 39  
      4.2.4 UPML Class ............................................................................................................. 39  
      4.2.5 UPML Interface ....................................................................................................... 41  
      4.2.6 UPML Function ....................................................................................................... 42  
      4.2.7 UPML Statement ..................................................................................................... 43  
      4.2.8 UPML Expression ................................................................................................. 44
5. Implementation of a PHP to UPML Transformer .......................................................... 57
5.1. Language and Format for the implementation ............................................................... 57
5.2. Work with the transformer online ............................................................................... 57
5.3. The Design of the PHP to UPML Transformer ............................................................... 63
5.4. Comparison with Related Work ..................................................................................... 69
6. Examples of Execution ........................................................................................................ 74
7. Conclusion and Future Work ............................................................................................ 86
Bibliography ......................................................................................................................... 90
Appendix A: Definition of UPML Tags ................................................................................. 93
Appendix B. PHP to UPML Transformer Source Code ......................................................... 112
Appendix C. PHP Tokens (from php.net) ........................................................................... 236
1. Introduction

We developed the Unified Programming Markup Language (UPML), with the goal to be used to develop a general software quality analysis platform with common software metrics and to serve as a gateway in software translation between various programming languages. In this section, a background overview of software metrics and programming language translation is provided. The current issues of software translation will be illustrated and discussed and the contribution of this thesis in these areas will be covered.

1.1. Software Metrics

Just like any product, quality needs to be measured and monitored. With the fast advancement of hardware and the increasing complexity of problems tackled by software, the “software crisis” may have existed from the start of software development history. The crisis manifested itself in several ways (www apl jhu edu) (Karch, 2011):

- Projects were running over-budget.
- Projects were running over-time.
- Software was very inefficient.
- Software was of low quality.
- Software often did not meet requirements.
- Projects were unmanageable and code difficult to maintain.
- Software was never delivered.
Even today, no single approach can prevent software project from overrunning and failures. Software quality management has become even more important with the development of complex systems.

As in any quality management, criteria of measurement must be established. Software metrics are used in software development and management. Software metrics are used to characterize the essential features of software quantitatively, so that classification, comparison, and mathematical analysis can be applied (S.D. Conte, 1986). Over the decades, many software metrics have been developed. These software metrics can be categorized from various perspectives. Basically, “software metrics can be classified into three categories: product metrics, process metrics, and project metrics. Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level. Process metrics can be used to improve software development and maintenance. Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process. Project metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity. Some metrics belong to multiple categories. For example, the in-process quality metrics of a project are both process metrics and project metrics.” (Kan, 2003) Product metrics and process metrics are more related with our interest in this paper. Everald E. Mills summarized the metrics in his curriculum module as follows (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

I. Product metrics

   a) Size of metrics
i. Lines of Code (LOC): Possibly the most widely used metric for program size.

ii. Function Points (FP): Computes the size in terms of the amount of functionality in a system, based upon the number of external user inputs, inquiries, outputs, and master files.

iii. Bang: Measures the total functionality of the software system delivered to the user.

b) Complexity Metrics

i. Cyclomatic Complexity – \( v(G) \): Proposed by McCabe that, \( v(G) = e - n + 2p \), where \( G \) is a control flow graph of a program, \( e \) is the number of edges, \( n \) is the number of nodes in the graph, and \( p \) is the number connected components (exit nodes), can be used as a measure of software complexity, called Cyclomatic Complexity (Geoffrey K. Gill, 1991).

ii. Extensions to \( v(G) \): An extension of Cyclomatic Complexity by Myers to measure the complexity using \( v'(G) = [l:u] \), where \( l \) and \( u \) are lower and upper bounds for the complexity (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

iii. Knots: Another control flow based measure suggested by Woodward, et al. the basis of this approach is program text, amended by lines which interconnect statements where control may be passed between them. These lines occasionally cross each other and thus create knots. The complexity is assumed to be proportional to the knot-count.
Indeed, well-structured, easy-to-read programs have less knots, but again data references are not included (Alan Jay Perlis, 1981).

iv. Information Flow: Counts the number of local information flows entering (fan-in) and exiting (fan-out) each procedure, calculates the procedure’s complexity by applying the formula c=procedure length x [fan-in x fan-out]² (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

c) Halstead’s Product Metrics

Halstead’s software science proposed a unified set of metrics that apply to several aspects of programs, as well as to the overall software production effort.

d) Quality Metrics

A quality metric is a metric that measures some quality characteristics of software such as correctness, efficiency, portability, maintainability, reliability, and so on. Some examples are as follows.

i. Defect Metrics: Tried to measure the number of defects. Since there is no effective procedure for counting the defects in the program, some alternative measures such as number of design changes, number of errors detected by code inspections, number of errors detected in program tests, and number of code changes required (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).
ii. Reliability Metrics: An expression to quantify the reliability of a software product. Measures such as Mean Time to Failure (MTTF), Mean Time to Repair (MTTR), Mean Time Between Failure (MTBF), Probability of Failure on Demand (POFOD), Rate of Occurrences of Failure (ROCOF), and Availability, have been proposed (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

iii. Maintainability Metrics: Measures or predicts the maintainability of the software product.

II. Process Metrics

a) General Considerations

Software metrics may be defined without specific reference to a well-defined model, such as LOC for program size. However, more often metrics are defined or used in conjunction with a particular model of the software development process. Models of various types are simply abstractions of the product or process. In general, models may be analytic-constructive or empirical-descriptive in nature (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

b) Empirical Models

Using an empirical model, the proposed project is compared to similar projects for which historical cost data are available. For example, in the Wolverton 1974 model, in order to determine actual software costs, each module was first classified as belonging to one of six basic types – control,
I/O, and so on. A level of difficulty is assigned to each module by categorizing it as new or old and as easy, medium, or hard. Finally, the size of the module is estimated, and the system cost is determined from historical cost data for software with similar size, type, and difficulty ratings (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

c) Statistical Models

C.E. Walston and C.P. Felix of IBM (1977) used data from 60 previous software projects. Based on LOC, a relationship of the form $E = aL^b$ was assumed, where $L$ is the number of lines of code, and $E$ is the total effort required. Regression analysis was used to find appropriate values of parameters $a$ and $b$, the resulting equation was $E = 5.2L^{0.91}$. Nominal programming productivity, in LOC per person-month, can then be calculated as $L/E$. In order to account for deviations from effort, a productivity index, $I$, which would increase or decrease the productivity, depending upon the nature of the project, was developed (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

d) Theory-based Models

i. Rayleigh Model

L. H. Putnam (1978) created a model of the software development process based upon the assumption that the personnel utilization during program development is described by a Rayleigh-type curve. Putnam assumed that either the overall staffing curve or the staffing curves for individual phases of the development cycle can be modeled
by an equation of this form. He then developed the relationship between the size of the software product and the development time (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

ii. Software Science Model

The software science equations by Halstead can be used as a simple theoretical model of the software development process.

e) Composite Models

i. Constructive Cost Model (COCOMO)

Boehm suggested COCOMO that provides three levels of models: basic, intermediate, and detailed. Boehm identified three modes of product development – organic, semidetached, and embedded – that aid in determining the difficulty of the project. The development effort equations are all of the form:

$$ E = a S^b m $$

where a and be are constants determined for each mode and model level; S is the value of source LOC; and m is the composite multiplier, determined from 15 cost-driver attributes (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

ii. SOFTCOST

Tausworthe, of the Jet Propulsion Laboratory, attempted to develop a software cost estimation model using the best features of other relatively successful models available at the time. His model incorporates the quality factors from Walson-Felix and the Rayleigh
iii. SPQR Model

T. Capers Jones developed a software cost estimation model called the Software Productivity, Quality, and Reliability (SPQR) model. The basic approach is similar to that of Boehm’s COCOMO model.

iv. COPMO

Thebaut developed a software development model that attempts to account specifically for the additional effort required when teams of programmers are involved on large projects. The general form of the equation for the effort, E, is assumed to be:

\[ E = a + bS + cP^d, \]

where a, b, c and d are constants to be determined from empirical data via regression analysis; S is the program size; and P is the average personnel level over the life of the project. This model presents an interesting form but it needs further development and calibration to be of widespread interest (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

v. ESTIMACS

Rubin developed a proprietary software estimating model that utilizes gross business specifications for its calculations. The ESTIMACS model addresses three important aspects of software
management – estimation, planning, and control (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988).

f) Reliability Models

A number of dynamic models of software defects have been developed. These models attempt to describe the occurrence of defects as a function of time, allowing one to define the reliability, R, and mean time to failure, MTTF.

In recent decade, security metrics have drawn more attention. While security metrics are hard to generate, many apply asset values, threat, and vulnerability as critical elements of overall risk (Payne, 2006). Security metrics can be considered as a part of software metrics. However, recent trends usually list security independently from software.

To compare the software metrics, there are even measures of software metrics, which are called meta-metrics, such as simplicity, validity, robustness, prescriptiveness, and analyzability (S.D. Conte, 1986).

1.2. Translation of Programs in Various Programming Languages

Since the invention of modern computers, many programming languages have been developed. According to the time they appeared, we can roughly put them in five generations (Clarke, 1991). Except the machine code, which is specifically designed for a computer machine, programs of all other kinds of languages need to be translated before they can be executed.

a. The Five Generations of Programming Languages

The first generation (programming) language (1GL) is a grouping of programming languages that are machine level languages used to program first-
generation computers. 1GL languages are specific to their computers and are directly coded in 0s and 1s. Therefore, no translation is needed for 1GL languages.

The second generation programming language (2GL) is a group of assembly languages, which are mnemonics of machine code. Mnemonics help programmers memorize and program. Even though mnemonics are direct representation of the machine code, they need to be translated into 0s and 1s. The translators are called assemblers.

The third generation programming language (3GL) is a group of languages that allow the programmers to focus on the logic procedures, the system objects, or business components that produce expected results, instead of the execution of the code. A term, high-level language, appeared to indicate that 3GLs are logically high above computer machines for human programmers to understand and use, without being bound to a specific kind of machines. 3GLs generally support structured programming with three basic control structures, sequence, selection, and loop. Popular general-purpose languages such as Fortran, ALGOL, COBOL, C, Pascal, and BASIC are typical 3GLs. These languages support structured programming techniques. Programs using these languages must be translated into machine code with either compilers, which are translators that convert entire program code into machine code before the program is executed, or interpreters, which are translators that convert a statement of a program when the statement is to be executed.

The fourth generation programming language (4GL) provides higher abstraction and statement power with support of object-oriented programming techniques. C++, Java, PHP, C#, and Visual Basic are all examples of 4GL. Similar to 3GLs, programs
of 4GLs must be translated into machine code for execution by either compilers or interpreters.

The fifth generation programming language (5GL) is a group of languages based on solving problems using constraints given to the program, rather than using an algorithm written by a programmer. 5GLs are designated to make the computer solve a given problem without the programmer. The programmer only needs to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or algorithm to solve them. Fifth-generation languages are used mainly in artificial intelligence research. Prolog, OPS5, and Mercury are examples of 5GLs. No doubt that programs of 5GLs need to be translated into machine code for execution.

Generally, each of the high level languages (3GLs, 4GLs, and 5GLs) has either a compiler or interpreter to translate into machine code. To make a programming platform to be somewhat universal, an intermediate language between the high level programming languages and the machine code appeared, such as Microsoft Common Intermediate Language (CIL) for its .NET languages. Source code of the high level languages such as C#, VB.NET, and J# are translated into CIL, and at runtime CIL is compiled to generate native code.

b. Types of Translators between Programming Languages

A translator is a computer program that translates a program written in a given programming language into a functionally equivalent program in a different language.

As mentioned above, a translator that translates a program of assembly language into machine code is called assembler. Examples include MASM, TASM, NASM,
and FASM. A translator that translates machine code back into an assembly language is called disassembler. Examples include gdb, IDA Pro, and OllyDbg.

A translator that translates a program of a high level language into a lower level language is called a compiler.

A translator that translates a high level language into an intermediate code, which will be immediately executed, is called an interpreter.

A translator that translates machine code to source language code is called a decompiler. Examples include DCC, Boomerang Decompilers, and Reverse Engineering Compiler (REC).

Theoretically, Every high level language can be translated into another high level Turing complete (Taylor, 2011) language. A program doing this translation is called a translator. Examples include Haxe, FORTRAN-to-Ada translators, CHILL-to-C++ translators, PASCAL-to-C translators, COBOL (DialectA)-to-COBOL(DialectB) translators.

1.3. Issues in the State-of-the-Art

a. Lack of General Software Metrics

As Mills (Mills, SEI Curriculum Module SEI-CM-12-1.1, 1988) and Jones (Jones, 2006) pointed out, each metric has its theoretical and practical weakness. Even though many metrics and a number of process models have been proposed, the topic itself is not mature yet. “There is no true international standard for any of the more widely used software metrics. Further, the adherents of each metric variant claim remarkable virtues for their choice, and often criticize rival metrics. The
software metrics domain is fragmented, incomplete, and gives the appearance of being more influenced by ‘metrics politics’ than by technical considerations.” (Jones, 2006)

In short, software metrics issues include:

a) No common criteria about what should be measured.
b) No common mechanism about how to measure.
c) No common environment about what platform to be used for general measurement.
d) No comprehensive representation of the measurement so that we cannot effectively compare the results.

We believe this status of software metrics research stems from its inherent complexity. We hope that, in the near future, when more practical statistical data about the use of the existing metrics and more in depth theoretical research will move the work forward to establish a solid conceptual foundation and standardized metric system. Among the many works, we believe a universal software analysis environment should be established to mitigate the four listed shortfalls and we believe that this work will contribute to establishing such an environment. More detail is following in section 1.4.

b. Less Effectiveness in Translation

In the translation of programs in different languages, all the mentioned translators, assemblers and disassemblers, compilers/interpreters and decompilers, and one-to-one high level language translators are well suited for each application because they are developed specifically for the source and targeted languages.
Our focus is automatic translation of a program in one high-level language to another high-level language. Even though a number of efforts have been attempted, successful results are still limited. “As programming languages become more and more diversified there is an increasing demand to translate programs written in one high-level language into another. Such translation can help us more effectively reuse the existing code, especially when automating translation is possible. However due to many subtle distinctions between different languages, usually only a subset of translation can be automated.” (Qiu, 1999)

The issues in high-level language translation include, but not limited to:

a) Each programming language inherent complexity makes translation difficult. Especially, current translation methods process both syntax and semantics of different languages at the same time, which is a burden in both development and maintenance.

b) Different elements in different languages add extra work to handle the difference in the translators of existing works. For example, C++ supports pointers but Java does not. The translators must add a mechanism to match the functionality. As to our knowledge there has not been a successful story of library translation.

c) Porting APIs may not be practical.

d) The resulting translated code is usually unreadable so that it is of little use.

e) We must develop a pair of translators (or translation modules) between two languages. Therefore, for n languages, we need n(n-1) translators. For example, we need ninety translators for ten languages.
f) When there is a change in a language, all related translators must be updated accordingly, which makes maintenance especially arduous.

1.4. Contribution of this Work

In this work, we developed a new language, called Unified Programming Markup Language (UPML) and implemented a PHP to UPML transformer. Our work contributes to both producing a general software analysis platform or environment and the translation of programs in various high-level programming languages.

We conceptually propose an innovative idea of UPML for developing a general software analysis environment and for programming language translation. The ideas are shown in the Figures 1, 2, and 3, and are individually discussed.

![Diagram](image)

**Figure 1.** UPML used to represent programs in other languages (PHP, Java, C/C++, etc.) and to be analyzed with a General Software Metric Analyzer

Our goal is to use UPML as a foundation for analysis. We want to establish a common set of metrics for evaluating and comparing software since the elements from each programming language is directly mapped to UPML constructs and UPML
constructs will be used to collect metrics. Classification, comparison, and mathematical analysis can be applied to the UPML elements of each program. Software metrics have been extensively studied and various categories of measures exist, including code metrics, requirement metrics, productivity metrics, design metrics, management metrics, product metrics, process metrics (Mills, Software Metrics, 1988) (Ashish Arora, A Survey on Software Quality Metrics, 2012), and vulnerability metrics (Software Engineering Institute, Carneigie Mellon Univeristy). However, most metrics have been defined and tested individually being applied in a limited environment. Little definitive work has been done to reduce the uncertainty associated with these metrics when applied in different environments. Even a relatively small set of universally useful metrics has not yet emerged (Ashish Arora, A Survey on Software Quality Metrics, 2012). This lack of a common software metrics can be credited to the nonexistence of a universal analysis environment. We posit that a universal analysis environment should possess the following characteristics:

a) **Platform independence.** Universal metrics can be applied to programs regardless of the operating systems executing the programs.

b) **Language independence.** Some existing metrics can measure programs in one or a few similar languages, but universal metrics must be able to measure programs in very different languages.

c) **Extensibility.** Since programming methodologies and techniques are rapidly advancing, universal software metrics must be capable of extending to measure new features.
d) **Comprehensiveness.** As a universal analysis environment, the metrics should be able to represent most (if not all) features in the languages to be measured.

e) **Visibility.** Differences and similarities between programs can be easily identified.

To satisfy the above requirements, common features of programming languages were identified and represented in XML. Each individual programming language structure was mapped to an XML representation. Applying this mapping, a programming file can be transformed into an XML file representing the information of a program written in any programming language. These representations in XML are called UPML. Possessing a unified representation not only makes final analysis easier by requiring only one analyzer but also provides the commonality required for direct comparisons among programming languages. Using the UPML representations, metrics can be created and subsequently counted. As seen in Figure 1, the programming language file is translated into UPML using a specific transformer program. UPML transformers for PHP, Java and C/C++ are being designed and written. Each transformation produces a UPML file. Further analysis of these languages will be accomplished by evaluating the UPML files. In Figure 1, the General Software Metric Analyzer measures software against established metrics and guidelines.
Figure 2. UPML Viewed as a Gateway to Convert Programs among the Mapped Languages

Once UPML represents a program from one of the mapped programming languages, the representation may be able to be transformed into one of the other mapped languages. If translation could not be completed entirely automatically (for example GUIs and any system calls might require programmer intervention or translating C pointer arithmetic to Java), the transformer could identify special cases and designate special handling. We realize that automatic translation is difficult. However, identifying nontransferable code could be as important in judging the quality of the code or as additional information whether to extend or port an
application. The translation from a programming language mapped to UPML and from UPML to a mapped programming language is illustrated in Figure 2.

Figure 3. UPML Viewed as a Universal Programming Language

Though UPML is not a programming language in the traditional sense, if an executor similar to a compiler or an interpreter is devised, UPML files can be executed directly, as represented in Figure 3.

Unified Programming Markup Language (UPML) is an instance of eXtensible Markup Language (XML) used to markup programming techniques. UPML represents an abstraction of general programming techniques that presently supports structured programming and object-oriented programming in a standardized unified format and stores the semantics of programs. Some key programming representations are illustrated in section 5 and the complete set of UPML definitions is listed in Appendix A.

Based on UPML, we expect to develop a general software analysis environment and a translation system between programming languages and UPML translating a program
into another language. As proof of concept, this thesis presents a PHP to UPML transformer. We selected PHP as the first transformation for the following reasons:

a) PHP supports both structured programming and object oriented programming.
b) PHP is one of the most popular programming languages that is used for both webpage-embedded programs and independent programs.
c) PHP supports the use of almost all popular databases.
d) PHP is syntax-loose inspiring UPML more extensive representations.
e) PHP is platform-independent.
f) PHP is compatible with most web servers.

1.5. Road Map of this Thesis

This thesis contains seven sections. In section 1, we review the related work in our targeted field. In section 2, we will analyze popular programming languages and programming techniques. In section 3, we will illustrate the implementation of UPML that represents the popular programming techniques. In section 4, we will analyze a general programming language and the popular programming techniques. In section 5, we will analyze the implementation of PHP to UPML transformer. In section 6, we present a few execution examples of PHP to UPML transformation. In section 7, we will reach our conclusion and discuss future work. UPML and PHP to UPML transformer code are attached at the end of this dissertation in Appendix A and B.
2. Related Work

2.1. Software Metrics

As Nuthakki (Murali K. Nuthakki, 2011) described, software developers are increasingly using Unified Modeling Language (UML) tools to automate source code generation, and acquisition of software metrics from Extensible Markup Language (XML) export of UML tools has been predominantly tool dependent. This means that XML has become one of the primary tools to represent software metrics and has been embedded in software design tools. Unfortunately, despite the overall need for a process to automate XML unification, no solution has yet been proposed.

Nuthakki (Murali K. Nuthakki, 2011) developed a parser-based framework, named UXSOM, to calculate software metrics independently from the UML tool. He announced that UXSOM is able to generate software metrics from existing XML-based outputs of UML tools, such as ArgoUML, UMLET, MagicDraw, Sparx Systems, and ESS model.

With the capability of extracting metrics from XML and Nuthakki’s independent framework in calculating software metrics, a unified metric analysis has not progressed. Our UPML is an attempt to standardize the XML representation first, and then extract software metrics from the standardized XML, UPML.

2.2. Programming Language Translation

2.2.1 Unified or universal programming
With more programming languages being developed to efficiently use the computer and network resources, or to suite programmers’ personal tastes, issues about learning and using these languages has appeared:

a) It has become a burden for programmers to learn additional languages. They have less time to focus on their business and applications.

b) It is difficult for a programming novice to select a language to learn for general use and aspiring programmers could be further discouraged by the multiplicity of languages that they must grasp.

c) Considerable efforts are expended in the management of software, computers, and networks, whose systems and languages must be integrated to work together.

d) Comparing the properties of different languages requires extensive work. While many software and security metrics have been developed, it is still hard to compare the quality and security between software packages of different languages.

To respond to these issues, devising a unified or universal programming languages (Michael Garland, 2012) (Vaudin, 1991) (Jungwon Kim, 2012) (Ana Lucia Varbanescu, 2011) (Miller, 1986) (Holst, 2005) and devising translation mechanisms between the high level programming languages (Qiu, 1999) (David P. Clark, 2004) (Marco Trudel, Automatic C to O-O Translation with C2Eiffel) (Marco Trudel, Automated Translation of Java Source Code to Eiffel) has been individually researched.

Unified or universal programming is appealing. However, “the unrealistic assumptions placed upon the syntax and semantics of programming languages
necessary to create a universal in intermediate language proved this impossible.” (David P. Clark, 2004) The attempts of unified programming are limited in some special areas (Miller, 1986) (Vaudin, 1991). Issues of developing translators directly between high-level languages are discussed in section 1.3.

We do not attempt to develop such a unified or universal programming language, or develop translators directly between two high-level languages. Instead, we devise a unified markup language, UPML, which will represent the programming techniques, as a gateway in automatic language translation.

2.2.2 Intermediate languages

Development of intermediate languages is not new. Most intermediate languages are developed for a platform to be compatible with multiple languages. For example, Microsoft Common Intermediate Language (CIL) was developed for its .NET languages. Source code of high-level languages such as C#, VB.NET, and J# are translated into CIL, and then at runtime CIL is compiled to generate native code. JAVA byte code is another example of intermediate language that makes Java compatible with various operating systems. Unfortunately, these intermediate languages serve specific purposes and are not readable, making them unusable for general translation.

2.2.3 XML in Language Translation

Many researchers have investigated using XML to represent language elements. Most of the XML representations are using eXtensible Stylesheet Language (XSL) to represent programming language syntax. Program semantic information is processed
in the translators. A research close to ours is done by David Clark et al (David P. Clark, 2004). They use eXtensible Stylesheet Language Transformations (XSLT), which was named Independent Stylesheet Language Translation (ISLT). ISLT requires each stylesheet to specify only the lexical and syntactical information specific to one language. It also includes optional semantic information for helping the translator to find appropriate mapping in a target language. One advantage is that only one stylesheet is required for one language so that only n XSL stylesheets are required for n languages, comparing to other potential approaches that use XSLT, which require n(n-1) XSL stylesheets. Another advantage is that any modification to a language affects only the stylesheet for that language without updating any other components. A third advantage is that the introduction of a new language into the translation domain requires the addition of only one new stylesheet. Further, stylesheets are extensible to add new features.

ISLT system architecture is depicted in figure 4. Where an Independent Stylesheet (IS) is a stylesheet associated with only one language; A Program Component List (PCL) is a set of components decomposed from the given source program code.
The translation procedure consists of three abstract phases: Extraction, Transformation, and Construction.

The ISLT translation approach is not very advantageous because, even though it reduces the number of translators from $n(n-1)$ to one, with $n$ added XSL stylesheets and one Generic Mapping Thesaurus (GMT), its translator is much more complex in that it contains four processes (Parser Generation, Program Decomposition, Deductive Mapping, and Program Composition) to handle all the involved languages plus the added burden of processing stylesheets. ISLT does not reduce any of the processing of syntax and semantics of all the involved languages and adds processing of stylesheets. To correctly implement the translator all the processing is combined into one translator making it more complex.
3. Analysis of Programming Languages and Programming Techniques

There are dozens of programming paradigms such as component-based, concurrent computing, parallel computing, data-driven, event-driven, time-driven, service-oriented, object-oriented, subject-oriented, role-oriented, domain-specific, generic, declarative, imperative, functional, and procedural. However, the dominant paradigms used in most applications are structured programming and object-oriented programming.

In a structured programming (SP) view, a system consists of data and processes (functions) that process data. Only three control structures – sequence, selection, and iteration – are necessary to implement any complex processes.

In an object-oriented programming (OOP) view, a system consists of a collection of interacting objects. OOP instantiates the objects from their classes and their interactions. An object class consists of properties (data that describe the class and implement encapsulation/information hiding) and methods (functions that implement interactions, which use the same techniques as those in SP). Object classes have relationships such as association, generalization (implements inheritance, polymorphism), and aggregation (implements composition). Objects’ interactions include responses to events, state transition and message passing. All OOP designs can be summarized in three groups of design patterns: creational (instantiation), structural (relationship), and behavioral (interaction) (James Rumbaugh, 1991) (Nirosh, 2011).

Today’s popular programming languages such as PHP, Java and C/C++ have language elements that implement part or all of the programming techniques in SP and OOP. Such a language (PHP, for example) can be typically decomposed into the following elements as shown in Figure 5.
a. Data (constants and variables): bool, integer, character, float, array, string, and list.

b. Operators:
   a) Object creators: clone, and new.
   b) Data type converters: (int), (float), (string), (array), (object), and (bool).
   c) Arithmetic operators: addition, subtraction, multiplication, division, modulus, increment, and decrement.
   d) Comparison operators: less than, greater than, equal, not equal, identical, and not identical.
   e) Logic operators: and, or, and not.
   f) Bitwise operators: and, or, not, and shift (left and right).
   g) Assignment, string, array operators.

c. Expressions that consist of operands (data) and operators.

d. Statements
   a) Simple statements: an expression with statement end (semicolon, for example).
   b) Structured statements: constructed with expressions and control words.

e. Functions that consist of function head (expression) and body (statements).

f. Classes that consist of class head (expression) and body (functions and statements).
4. Implementation of UPML

UPML is a set of instances of XML representations of the language elements as listed in Figure 5. In this section we discuss our consideration in designing UPML and introduce representations of a few key elements such as class, interface, function, statement, and
expression. A complete set of UPML is in Appendix A. UPML is open to add features to include new programming techniques and techniques in other programming paradigms.

4.1 Design Consideration

UPML aims to have the following properties:

a. **Extensible.** UPML preserves extensibility of XML, so that new tags can be introduced to cover most (if not all) programming paradigms.

b. **Inclusive.** UPML is open to add any missed or new programming techniques. This backups its extensibility.

c. **Uniform.** In UPML, a representation of a pattern is the same for all programming languages. For example, the following tags represent an assignment statement that assigns an integer value to a variable. UPML representation is

```
<statement>
  <assign>
    <variable>$myInt</variable>
    <literal_value>5</literal_value>
  </assign>
</statement>
```

**Figure 6.** An example of UPML representation for

In Java, the code is: “int myInt = 5;”, while in PHP, the code is: “$myInt = 5;”, and in Pascal, the code is “var myInt: integer; myInt := 5;”. PHP does not require data type in declaration because it is a syntax-loose language, and Pascal requires declaration before assignment. The uniform UPML representation applies to the different formats in these languages.
The uniformity property of UPML is the reason that UPML can be used as the gateway of transformation among different languages.

d. **Descriptive**: as the example shows, all tags use descriptive meaningful words instead of shorten forms or operators. For example `<integer>` is used instead of `<int>`, and `<assign>` instead of `< = >`. This is important not only for uniformity (for example, PASCAL uses “:=” for assignment so it is not accurately represented with “=”), but also for inclusion to further use UPML to represent declarative languages or even natural languages.

Attributes are avoided in tags whenever possible because XML attributes have issues as listed below (w3schools.com):

a) Attributes cannot contain multiple values (elements can)

b) Attributes cannot contain tree structures (elements can)

c) Attributes are not easily expandable (for future changes)

However, attributes are not eliminated because they are more efficient in some representations of information.

### 4.2 UPML Representation of Programming Techniques

The convention used in the following context is:

- Defined UPML tags are in bold, and attributes are in italic bold.
- Short format has slash (/) as the end.
- For webpage embedded languages such as PHP, HTML tags are represented by `(<!--) and (-->).`
- A bar ( | ) means “or” in a set of optional items.
• A slash (/) represents an end tag.
• In an expression or a statement, the operator or predicate (if exists) is first and encloses the operand(s) or subject(s).
• A pair of square brackets ([ ] and [ ] ) indicates an optional item.
• A star (*) denotes 0 or more items.
• A plus (+) denotes 1 or more items.
• A question mark (?) denotes 0 or one item.

The current work implements the basic core part of programming techniques in both structured programming and object-oriented programming, that is, classes, interfaces, functions, control structures, statements, and expressions. The details are following.

4.2.1 UPML File

A UPML file is enclosed in tag pair <upml />, which is defined as:

<upml language = “source code language type” source = “source file name” />

Example:

<upml language="php" source="a01.php">
  ...
</upml>

4.2.2 HTML Code

HTML code and other elements not belonging to the programming language are enclosed in UPML comment <!-- ...... -->:
Example:

```html
<!--
<html>
<head>
<title> This is an example of PHP file </title>
</head>
<body>
<h1>this is the first PHP code section</h1>
<p>how about a paragraph?</p>
-->
```

Note: this is a part of a HTML file, therefore the HTML end tags do not appear.

4.2.3 Program Code

The program code is enclosed in tag pair `<code />`, defined as

```html
<code section="section number"> <!--language elements: directives, includes, statements, functions, classes, and or interfaces --> ...
</code>
```

Example:

```html
<code section="0">
...
</code>
```

Note: For webpage embedded languages, such as PHP and JavaScript, a few sections may be embedded. The section number sequence starts from 0.

4.2.4 UPML Class

A Class is enclosed in class tag pair `<class />`, defined as

```html
<class type="public | private | protected | final | abstract"> <!--the default type is public -->
```
where `<inherit />` is defined as

```
<inherit />
```

and `<implement />` is defined as

```
<implement>

  <list />  <!-- a list of interfaces-->

</implement>
```

with `<body />` is defined as

```
<body>

  <statement />  * <!-- a group of statements-->

</body>
```

where `<statement />  *` is defined as a sequence of statements:

```
<statement />  <!-- to be defined in statement section-->

:

<statement />.
```

A list is `<list />`, defined as

```
<list dilimeter="comma | space | dash | slash ">
```
Example of PHP class. Statements will be explained in section 4.2.7.

PHP code:

```php
class bar extends foo
{
    $foo->printItem('baz'); // Output: 'Foo: baz'
}
```

UPML representation:

```xml
<class type="public">
    <name>bar</name>
    <inherit>
        <list delimiter="comma">foo</list>
    </inherit>
    <body>
        <statement>...</statement>
        <comment type="general">// Output: 'Foo: baz'</comment>
    </body>
</class>
```

4.2.5 UPML Interface

An interface is enclosed in interface tag pair `<interface />`, defined as

```xml
<interface>
```
The UPML representation of interface is very similar to class, except that there is no implement in the interface head.

### 4.2.6 UPML Function

A function is enclosed in function tag pair `<function />`, defined as

```xml
<function final="yes | no" type="public | protected | private | constructor | destructor | general | static | abstract">
  return="bool | int | float | string | null | void">
</function>
```

- **final**: “yes” or “no”
- **type**: “public | protected | private | constructor | destructor | general | static | abstract”
- **return**: “bool | int | float | string | null | void” (default type is void)

- constructor: function name is __construct() in php, same as class name in java and C++, etc.
- destructor: function name is __destruct() in php, no destructor in java, and (~) with class name in C++, etc.
- return indicates the return data type available in a programming language, such as int, char, bool, etc.
Example:

PHP code:

public function printHello()
{
    $i = 3;
    return $i;
}

UPML representation:

<function final="no" type="public" return="integer">
  <name>printHello</name>
  <argument></argument>
  <body>
    <statement>
      <assign>
        <variable>$i</variable>
        <literal_value>3</literal_value>
      </assign>
    </statement>
    <return>
      <variable>$i</variable>
    </return>
  </body>
</function>

4.2.7 UPML Statement

A simple statement <statement /> is defined as
**4.2.8 UPML Expression**

An expression may be a simple one, or complex one with control structures. They may be

<expression /> | <if /> | <if_else /> | <switch /> | <while /> | <do_while /> | <for /> | <foreach />
| <declare /> | <break /> | <condition /> | <default /> | <return />.

An if structure <if /> is defined as

<if>

<condition>

<expression />

</condition>

<body />
<!--body, to be defined below-->

</if>

An else structure <else /> is defined as

<else>

<body />

</else>

An if-else structure <if_else /> is defined as

<if />

<else />
Nested if-else statement can place <if /> behind <else />.

Example:

PHP code:

```php
<?php
    if ($a > $b)
    {
        echo "a is bigger than b";
    }
    elseif ($a == $b)
    {
        echo "a is equal to b";
    }
    else
    {
        echo "a is smaller than b";
    }
?>
```

UPML representation:

```xml
<code section="0">
    <if>
        <condition>
            <greater>
                <variable>$a</variable>
                <variable>$b</variable>
            </greater>
        </condition>
        <body>
            <output device="monitor">
                <literal_string>
                    "a is bigger than b"
                </literal_string>
            </output>
        </body>
    </if>
</code>
```
Switch-case structure contains `<switch />`, `<case />`, `<break />`, `<continue />`, `<default />`, etc. They are defined respectively as follows.
<switch/> is defined as

```
<switch>
  <expression/>
  <body/>
</switch>
```

<case/> is defined as

```
<case>
  <case_value/>
  <body/>
</case>
```

and <default/> is defined as

```
<default>
  <expression>*/
</default>
```

Break statement <break/> and continue statement <continue/> are already in the defined forms because they have no further content.

Example:

PHP code:

```php
switch ($i)
{
  case 0:
    echo "i equals 0";
    break;
  case 1:
    //
```
```php
$echo "i equals 1";
break;

$case 2:
$echo "i equals 2";
break;

default:
$echo "i no es igual a 0, 1 ni 2";
}

UPML representation is:

<switch>
  <expression>
    <variable>$i</variable>
  </expression>
  <body>
    <case>
      <case_value>
        <literal_value>0</literal_value>
      </case_value>
      <body>
        <output device="monitor">
          <literal_string>
            "i equals 0"
          </literal_string>
        </output>
        <break/>
      </body>
    </case>
    <case>
      <case_value>
```
<literal_value>1</literal_value>
</case_value>
<body>
<output device="monitor">
<literal_string>  
"i equals 1"
</literal_string>
</output>
<brack/>
</body>
</case>
<case>
<case_value>
<literal_value>2</literal_value>
</case_value>
<body>
<output device="monitor">
<literal_string>  
"i equals 2"
</literal_string>
</output>
<brack/>
</body>
</case>
<default>
<body>
<output device="monitor">
<literal_string>  
"i no es igual a 0, 1 ni 2"
</literal_string>
</output>
</body>
</default>
While structure `<while />` is defined as

```xml
<while>
  <condition>
    <smaller_equal>
      <variable>$i</variable>
      <literal_value>10</literal_value>
    </smaller_equal>
  </condition>
  <body />
</while>
```

Example:

PHP code:

```php
while ($i <= 10) {
    echo $i++;
}
```

UPML representation:

```xml
<while>
  <condition>
    <smaller_equal>
      <variable>$i</variable>
      <literal_value>10</literal_value>
    </smaller_equal>
  </condition>
  <body />
</while>
```
Do-while structure <do_while /> is defined as

```xml
<do>
  <body />
  <while> <!--conditon-->
</do>
```

Example:

PHP code:

```php
do {
    echo $i;
} while ($i > 0);
```

UPML representation:

```xml
<do>
  <body>
    <output device="monitor">
      <variable>$i</variable>
    </output>
  </body>
  <while>
    <greater>
      <variable>$i</variable>
      <literal_value>0</literal_value>
    </greater>
  </while>
</do>
```
For structure `<for />` is defined as

```
<for>
  <condition />  <!--the condition here consists of statements: the -->
  <body />  
</for>
```

The `<condition />` in for structure is special. It consists of statements. So the `<condition />` is further defined as

```
<condition>
  <initial_condition />  
  <end_condition />  
  <update_condition />
</condition>
```

where `<initial_condition />`, `<end_condition />`, and `<update_condition />` are defined the same way as usual `<condition />` as

```
<initial_condition>
  <expression />  
</initial_condition>
```

and

```
<end_condition>
  <expression />  
</end_condition>
```
and

Example:

PHP code:

```php
for ($i = 1; $i <= 10; $i++) {
    echo $i;
}
```

UPML representation:

```xml
<for>
    <condition>
        <initial_condition>
            <assign>
                <variable>$i</variable>
                <literal_value>1</literal_value>
            </assign>
        </initial_condition>
        <end_condition>
            <smaller_equal>
                <variable>$i</variable>
                <literal_value>10</literal_value>
            </smaller_equal>
        </end_condition>
    </condition>
    <update_condition>
</for>
```
For-each structure `<foreach>` is defined as

```
<foreach>
  <expression />
  <body />
</foreach>
```

Example:

PHP code:

```php
$arr = array(1, 2, 3, 4);
foreach ($arr as &$value) {
    $value = $value * 2;
}
```

UPML representation:

```
<statement>
  <assign>
```

Page 54
<variable>$arr</variable>
<array_define>
    <list delimiter="comma">
        <literal_value>1</literal_value>
        <literal_value>2</literal_value>
        <literal_value>3</literal_value>
        <literal_value>4</literal_value>
    </list>
</array_define>
</assign>
</statement>
<foreach>
    <expression>
        <as>
            <variable>$arr</variable>
            <reference>$value</reference>
        </as>
    </expression>
    <body>
        <statement>
            <assign>
                <variable>$value</variable>
                <multiply>
                    <variable>$value</variable>
                    <literal_value>2</literal_value>
                </multiply>
            </assign>
        </statement>
    </body>
</foreach>
Declare structure <declare/> is defined as

<declare>
  <expression> <!--directive expression-->
</declare>

Example:

PHP code:

    // these are the same:
    // you can use this:
    declare(ticks=1) {
      // entire script here
    }

    // or you can use this:
    declare(ticks=1);
    // entire script here

UPML representation:

<comment type="general">// these are the same:</comment>
<comment type="general">// you can use this:</comment>
<declare>
  <expression>
    <assign>
      ticks
    </assign>
    <literal_value>1</literal_value>
  </expression>
</declare>
<comment type="general">// or you can use this:</comment>
<declare>
 <expression>
  <assign>
    ticks
    <literal_value>1</literal_value>
  </assign>
 </expression>
</declare>
<comment type="general">// entire script here</comment>

An expression <expression /> encloses any possible content in tag pair <expression> and <expression>. An example of <expression /> can be found above for <declare/> statement.

A complete set of UPML tags is attached in Appendix A.

5. Implementation of a PHP to UPML Transformer

5.1. Language and Format for the implementation

As mentioned in section 1.4, we implemented the PHP to UPML transformer. We will also use PHP to implement transformers for other languages.

We implemented the transformer and located it online at a site with a specific domain name. The location is http://upml.net.

5.2. Work with the transformer online

The transformer is online at http://upml.net. To use this transformer, please follow the steps below.
a. Go to [http://upml.net](http://upml.net)

![Figure 6. Homepage of http://upml.net](image]

b. Click the “Translation” tab in the leftmost of the menu bar as shown in figure 6. You will see the page for translation shown in figure 7.

### Start to Translate

Please specify a program file you want to translate:  

Choose File  
No file chosen

Your file name is:  

Your file extension is:  

Please specify your target language and submit:  

Select your targeted language  
Submit

**Figure 7. Translation Page**
c. Click the “Choose File” button to select your source file for translation. At this time, only PHP is supported.

d. Once you choose the file, the file name and extension will appear in the textboxes as shown in Figure 8.

**Start to Translate**

Please specify a program file you want to translate:  

Choose File a01.php

Your file name is: a01

Your file extension is: php

Please specify your target language and submit: Select your targeted languaage

Submit

**Figure 8. File Selected**

e. You may select the targeted language to translate. However, at this time, only UPML is supported as the targeted representation.

f. Click the “Submit” button, you will see the information about the submission. If the file is already on the server, you will be notified. The file size is displayed also.

The file: uploads/a01.php already exists transforming PHP to UPML. Source filesize- 411

Click [here](#) to view the UPML file.

**Figure 9. File Submitted**

g. Click the link “here” to view the result as shown in Figure 10.

The source code for the result is shown below.

```html
<html>
<head>
```
<title>
This is an example of PHP file
</title>
</head>
<body>
<h1>this is the first PHP code section</h1>
<p>how about a paragraph?</p>
<?php
    // these are the same:

    // you can use this:
    declare(ticks=1) {
        // entire script here
    }

    // or you can use this:
    declare(ticks=1);
    // entire script here
?>
<h1>3. there is still something of HTML to end.</h1>
</body>
</html>

In the result, your will see the message at the top: “This XML file does not appear to have any style information associated with it. The document tree is shown below.”
At this time, we have not created a style sheet file yet. We use the default style in the browser for the display.

The text between “<!--” and “--->” is the HTML text in the source code.
This XML file does not appear to have any style information associated with it. The document tree is shown below.

```xml

<upml language="php" source="a05.php">

<html>
<head>
<title>This is an example of PHP file</title>
</head>
<body>
<h1>this is the first PHP code section</h1>
<p>how about a paragraph?</p>

--> 

<code section="0">
<!-- these are the same:
<!-- you can use this:

<declare>
  <expression>
    <assign>
      <label>ticks</label>
      <literal_value>1</literal_value>
    </assign>
  </expression>
</declare>

<!-- or you can use this:

<declare>
  <expression>
    <assign>
      <label>ticks</label>
      <literal_value>1</literal_value>
    </assign>
  </expression>
</declare>

<!-- entire script here

</code>

--> 

<h1>3. there is still something of HTML to end.</h1>
</body>
</html>

--> 

</upml>
```

Figure 10. Result
5.3. The Design of the PHP to UPML Transformer

a. Architecture

The work of this paper implemented a Web-based translation system using UPML as the gateway. Specifically, PHP is used for the implementation and the core part of programming in PHP is transformed to UPML. The architecture can be illustrated as in Figure 11.

Inside the translation system, an input program of a programming language is switched to a module of the given language to UPML; the result is stored in UPML representation. When a UPML is to be transformed to a programming language, it is switched to an according UPML to language transformation module. At this time, only PHP to UPML module has been implemented.

The user controls the switch by selecting the language.
b. **Solutions to the Key Issues**

The transformation system and the PHP to UPML transformer are implemented in PHP. Some Key issues in the implementation are given below.

a) **Identifying the language elements**

Identifying elements of a programming language is one of the most tedious tasks to do. Generally, a reader scans the input character stream of the given input to identify the key elements and structure in the language. Fortunately, PHP provides a powerful tokenizer, token_get_all, which tokenizes all elements in PHP language and provide information of the tokens, such as a sequence number, token definition, and language element. The information is very useful in locating the elements and identifying language structures such as class, interfaces, functions, and operators.

b) **Identifying the language structures**

Once the tokens are identified with token_get_all, the white-space is filtered because white-space of various lengths make it difficult in identifying the tokens. Fortunately, PHP provides a filter called array_filter that can be used to filter any given character. The array with space filtered out is stored in an array with PHP’s array_values.

The language structures are identified by their unique tokens. For example, class has token T_CLASS, function has token T_FUNCTION, array has token T_ARRAY, if has token T_IF, and switch has token T_SWITCH.

To identify the entire program structure, we read through the tokenized file and identify the HTML elements and the start and end of a section of PHP code. Therefore we locate the following tokens:
i. T_INLINE_HTML: HTML text.

ii. T_OPEN_TAG: the start of a section of PHP code.

iii. T_CLOSE_TAG: the end of a section of PHP code.

Between the open tag (<?PHP) and end tag ( ?>), we can identify the top level elements, which are those can appear without being enclosed in a body of another elements. The tokens of such elements include:

i. T_COMMENT: comments in PHP.

ii. T_DOC_COMMENT: doc style comments in PHP.

iii. T_START_HEREDOC: heredoc in PHP.

iv. T_INTERFACE: interface of PHP.

v. T_CLASS: PHP classes.

vi. T_FUNCTION: php functions.

vii. T_IF: if control structure.

viii. T_ELSE: else control structure.

ix. T_ELSEIF: else if control structure.

x. T_SWITCH with T_CASE, T_BREAK, T_CONTINUE, T_DEFAULT: switch, case, break, and continue for switch statements or for independent use.

xi. T_DO: do-while control.

xii. T_FOR: for loop control.

xiii. T_FOREACH: foreach loop control

xiv. T_DECLARE: declare statement is the same as include predictive.
xv. T_REQUIRE and T_REQUIRE_ONCE: require statement is similar to include predictive.

xvi. T_RETURN: return statement.

xvii. T_GOTO: goto statement. Though structured programming depreciated goto statement, PHP keeps but not encourages to use it. For completeness, we include this token.

xviii. T_GLOBAL: global is a key word that allows to use external variables.

xix. T_STATIC: keyword static is used to declare static variables.

xx. T_ECHO and T_PRINT: echo and print are used for output.

xxi. T_STRING: strings such as keyword goto, label etc. are special strings.

xxii. T_UNSET: used to destroy an array.

Once we identify the top-level elements, we need to identify elements within these top-level elements. These top-level elements consists of four types of structured units: classes/interfaces, functions, complex statements (with control structures), and simple statements. Except the simple statements, the other structured units can be divided into head (including name and condition) and body. Inside a body, all elements as those listed above except T_CLASS, T_INTERFACE, T_INLINE_HTML, T_OPEN_TAG, and T_CLOSE_TAG. Therefore, expressions are identified in head (its condition), and tokens except the few listed here are identified in body.
Some other tokens are not used in identifying independent elements but are used in identifying expressions and primitive elements such as T_DEC for operator decrement, and T_DNUMBER for float number.

A complete set of tokens is listed in Appendix C (php.net).

As shown in Figure 5, all elements finally consist of expressions. Therefore, identifying expressions is critical. Expressions consist of constants, variable, literal values, and operators. The key in identifying expressions is identifying how the operators are used to construct the expressions. Therefore it is critical to identify the operators and construct UPML statements in accordance with the operator precedence.

c) **Implementing the operator precedence in the expressions**

PHP operator precedence is listed below (PHP.net):

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Operators</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-associative</td>
<td>clone new</td>
<td><code>clone</code> and <code>new</code></td>
</tr>
<tr>
<td>left</td>
<td>[</td>
<td><code>array()</code></td>
</tr>
<tr>
<td>right</td>
<td>++ -- ~ (int) (float) (string) (array) (object) (bool) @</td>
<td><code>types</code> and <code>increment/decrement</code></td>
</tr>
<tr>
<td>non-associative</td>
<td>instanceof</td>
<td><code>types</code></td>
</tr>
<tr>
<td>right</td>
<td>!</td>
<td><code>logical</code></td>
</tr>
<tr>
<td>left</td>
<td>* / %</td>
<td><code>arithmetic</code></td>
</tr>
<tr>
<td>left</td>
<td>+ - .</td>
<td><code>arithmetic</code> and <code>string</code></td>
</tr>
<tr>
<td>Associativity</td>
<td>Operators</td>
<td>Additional Information</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>left</td>
<td>&lt;&lt; &gt; &gt;</td>
<td>bitwise</td>
</tr>
<tr>
<td>non-associative</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>comparison</td>
</tr>
<tr>
<td>non-associative</td>
<td>== != === !== &lt;&gt;</td>
<td>comparison</td>
</tr>
<tr>
<td>left</td>
<td>&amp;</td>
<td>bitwise and references</td>
</tr>
<tr>
<td>left</td>
<td>^</td>
<td>bitwise</td>
</tr>
<tr>
<td>left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>left</td>
<td>&amp;&amp;</td>
<td>logical</td>
</tr>
<tr>
<td>left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>left</td>
<td>? :</td>
<td>ternary</td>
</tr>
<tr>
<td>right</td>
<td>+= -= *= /= .= %= &amp;=</td>
<td>assignment</td>
</tr>
<tr>
<td>left</td>
<td>and</td>
<td>logical</td>
</tr>
<tr>
<td>left</td>
<td>xor</td>
<td>logical</td>
</tr>
<tr>
<td>left</td>
<td>or</td>
<td>logical</td>
</tr>
<tr>
<td>left</td>
<td>,</td>
<td>many uses</td>
</tr>
</tbody>
</table>

To correctly construct the UPML statements in accordance with the PHP operator, we start from identifying commas in a given expression, divide the expression by *commas*, and then identify each of the divided parts (sub expressions).
Then we identify *logical or, xor, and, assignment operators, ternary*, and so on, up to *clone* and *new* finally, in a reversed order as the listed operator precedence.

5.4. Comparison with Related Work

Most methods of programming language translation are direct translation from one to another. There are some efforts using XML as assistance in translation, one of such work called ISLT is provided by David Clark (David P. Clark, 2004). In the table below, we compare the typical methods of programming translation. We can reach our conclusion that, our method is simpler, more efficient, reusable, result-readable, extensible, and has the potential to be the foundation for building a general system analysis system.
<table>
<thead>
<tr>
<th>Items</th>
<th>Source to Source</th>
<th>ISLT</th>
<th>UPML</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is translated?</td>
<td>Syntax and semantics</td>
<td>Syntax and semantics</td>
<td>Semantics alone</td>
<td>The translation based on UPML is simpler and the result is readable.</td>
</tr>
<tr>
<td>What is predefined?</td>
<td>Nothing</td>
<td>Language styles for syntax in many Independent Styles (IS)</td>
<td>Unified Programming Markup Language (UPML) as predefined syntax patterns uniformly represents programming languages based on their semantics</td>
<td>IS’ separate languages from each other so that maintenance is simplified because update of a language or adding a new language does not affect others. UPML as a new extensible language can be easily updated with new features and new languages. UPML makes it possible to translate semantics alone.</td>
</tr>
<tr>
<td>What is shared between the languages?</td>
<td>Nothing</td>
<td>Nothing. An IS independently represents the styles of one single language.</td>
<td>All languages share the same predefined uniform UPML syntax patterns</td>
<td>The predefined syntax patterns in UPML make it a gateway in the transformation so that the number of translators is greatly reduced, and the transformation based on patterns is simplified than translations.</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>How is translation done?</td>
<td>Transformation including both syntax and semantics is coded in the one-to-one translators.</td>
<td>IS’ are used to identify syntax. Transformation including both syntax and semantics is coded in the universal translator.</td>
<td>Source to UPML transformation is done simply by extracting program information and filling it in the predefined UPML patterns; UPML to destination is done simply by mapping the UPML patterns to the program language patterns.</td>
<td>Source to source translation requires n(n-1) complex translators. ISLT requires only one universal translator, which integrates all the complex translation mechanisms and is more complex than source-to-source translation. UPML requires 2n much simpler translators.</td>
</tr>
<tr>
<td>What is stored in the translation?</td>
<td>Nothing.</td>
<td>Nothing.</td>
<td>UPML representation of the program.</td>
<td>The programs UPML representation can be used in software analysis [1].</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------</td>
<td>----------</td>
<td>-------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Is the resulting code readable?</td>
<td>Generally no, because extra code will be added in the syntax translation.</td>
<td>Not clear because the author has not mentioned. Since the translation is similar to source to source, the resulting code is very possibly not readable.</td>
<td>Yes, because the transformation is simply pattern match without syntax conversion.</td>
<td>Only UPML keeps the resulting code readable.</td>
</tr>
<tr>
<td>How many translators are needed for reciprocal</td>
<td>$n(n-1)$</td>
<td>1</td>
<td>$2n$ modules coordinated in 1 translation mechanism.</td>
<td>ISLT requires only one translator. However, since it integrates all the mechanisms of translating both syntax and semantics in all the involved languages,</td>
</tr>
</tbody>
</table>
### Table 1. Comparison of Solutions to Programming Language Translation

| translation between n languages? |  |  | its translator is extremely complex. UPML requires 2n translators (or modules for n languages), which are much simpler than source-to-source translators. |

Page 73
6. Examples of Execution

In this section, we offer a few executable examples of transformation from PHP code to UPML. These examples show the correctness of the transformation. Since we have implemented only the programming core of PHP language, that means, the general programming techniques in structured programming and object-oriented programming, are implemented. Therefore, basic programming such as structures of classes, interfaces, functions, statements, and control structures are implemented. No extension of PHP has been implemented yet. Therefore PHP library, specific definitions, database support, and so on, are not implemented.

**Example 1.** Approximates integral\(_a\_b\) \(f(x)\ \text{dx}\) with composite Simpson's rule with \(n\) intervals(php.net).

This example shows the UPML representation for function definition, complex mathematical calculations, selection, and loop control structures.

**PHP code:**

```html
<html>
<head>
<title>
This is a web page running sample PHP code.
</title>
</head>
<body>
</body>
```
<?php

function simpsonf($x)
{
    // returns f(x) for integral approximation with composite Simpson's rule
    return (pow((1+pow($x, (-4))), 0.5));
}

function simpsonsrule($a, $b, $n)
{
    // approximates integral_a_b f(x) dx with composite Simpson's rule with $n intervals
    // $n has to be an even number
    // f(x) is defined in "function simpsonf($x)"
    if($n%2==0)
    {
        $h=($b-$a)/$n;
        $S=simpsonf($a)+simpsonf($b);
        $i=1;
        while($i <= ($n-1))
        {
            $xi=$a+$h*$i;
            if($i%2==0)
            {
                $S=$S+2*simpsonf($xi);
            }
            else
            {
                $S=$S+4*simpsonf($xi);
            }
            $i++;
    }
function simpsonf($x):
    return (h/3 * S);

else:
    return ('$n has to be an even number');

?>
</body>
</html>

UPML representation:

```xml
<upml language="php" source="a01.php">
<!--
<html> <head> <title> This is a web page running sample PHP code. </title> </head> <body> <h1> demo about calculations, functions, control structures</h1> -->
<code section="0">
    <function final="no" type="general" return="string">
        <name>simpsonf</name>
        <argument>
            <variable>$x</variable>
        </argument>
        <body>
            <comment type="general">
                // returns f(x) for integral approximation with composite Simpson's rule
            </comment>
```
<function_expression>
  <name>pow</name>
  <arguments>
    <add>
      <literal_value>1</literal_value>
      <function_expression>
        <name>pow</name>
        <arguments>
          <add>
            <variable>$x$</variable>
            <negative>
              <literal_value>4</literal_value>
            </negative>
            <literal_value>0.5</literal_value>
          </add>
        </arguments>
      </function_expression>
    </add>
    <literal_value>0.5</literal_value>
  </arguments>
</function_expression>

<function final="no" type="general" return="string">
  <name>simpsonsrule</name>
  <argument>
    <comma>
      <variable>$a$</variable>
  </argument>
</function>
<variable>$b</variable>
<variable>$n</variable>

</argument>
</argument>
<body>

<comment type="general">
// approximates integral_a_b f(x) dx with composite Simpson's rule
with $n$ intervals
</comment>
<comment type="general">
// $n$ has to be an even number</comment>
<comment type="general">
// f(x) is defined in "function simpsonf($x)"
</comment>
<if>
  <condition>
    <is_equal>
      <modulo>
        <variable>$n</variable>
        <literal_value>2</literal_value>
      </modulo>
      <literal_value>0</literal_value>
    </is_equal>
  </condition>
</if>
<body>
<statement>
  <assign>
    <variable>$h</variable>
    <divide>
      <subtract>
        <variable>$b$</variable>
        <variable>$a$</variable>
      </subtract>
      <variable>$n$</variable>
    </divide>
  </assign>
</statement>
\[
\frac{S}{2} + \left( \frac{S}{2} \right) = \left( \frac{S}{2} \right) + \left( \frac{S}{2} \right) = S
\]

\[
\text{for } i = 1, 2, \ldots, n, \quad n \geq 2
\]
<subtract>
  <variable>$n</variable>
  <literal_value>1</literal_value>
</subtract>
</smaller_equal>
</condition>
<body>
<statement>
<assign>
  <variable>$x_i</variable>
  <add>
    <variable>$a</variable>
    <multiply>
      <variable>$h</variable>
      <variable>$i</variable>
    </multiply>
  </add>
</assign>
</statement>
<if>
  <condition>
    <is_equal>
      <modulo>
        <variable>$i</variable>
        <literal_value>2</literal_value>
      </modulo>
      <literal_value>0</literal_value>
    </is_equal>
  </condition>
<body>
<statement>
<assign>
<variable>$S</variable>
<add>
  <variable>$S</variable>
  <multiply>
    <literal_value>2</literal_value>
    <function_expression>
      <name>simpsonf</name>
      <arguments>
        <variable>$xi</variable>
      </arguments>
    </function_expression>
  </multiply>
</add>
</assign>
</statement>
</body>
</if>
<else>
<body>
<statement>
<assign>
  <variable>$S</variable>
  <add>
    <variable>$S</variable>
    <multiply>
      <literal_value>4</literal_value>
      <function_expression>
        <name>simpsonf</name>
        <arguments>
          <variable>$xi</variable>
        </arguments>
      </function_expression>
    </multiply>
  </add>
</assign>
</statement>
</body>
</else>
</multiply>
</add>
</assign>
</statement>
</body>
</else>

<statement>
  <post_increment>
    <variable>$i</variable>
  </post_increment>
</statement>
</body>
</while>

<return>
  <divide>
    <variable>$h</variable>
    <multiply>
      <literal_value>3</literal_value>
      <variable>$S</variable>
    </multiply>
  </divide>
</return>
</body>
</if>

<else>
  <body>
    <return>
      <literal_string>'$n has to be an even number'</literal_string>
    </return>
  </body>
</else>
Example 2. This example shows classes, class inheritance, object creation, object members, and so on.

PHP Code:

```php
<?php

class A {
    
    public $x = 'A';

    public function foo() {
        $b = new B;
        $b->bar();
        return $this->x;
    }
}

class B extends A {
    
    public function bar() {
        $this->x = 'B';
    }
}

$a = new A

echo $a->foo();   //A
```
UPML Representation:

```xml
<upml language="php" source="a03.php">
  <code section="0">
    <class type="public">
      <name>A</name>
      <body>
        <statement>
          <assign>
            <variable scope="public">$x</variable>
            <literal_string>'A'</literal_string>
          </assign>
        </statement>
        <function final="no" type="public" return="integer">
          <name>foo</name>
          <argument></argument>
          <body>
            <statement>
              <assign>
                <variable>$b</variable>
                <new_object>B</new_object>
              </assign>
            </statement>
          </body>
        </function>
        <statement>
          <object_name>
            <variable>$b</variable>
          </object_name>
          <object_member>
            <function_expression>
              <name>bar</name>
              <arguments></arguments>
            </function_expression>
          </object_member>
        </statement>
      </body>
    </class>
  </code>
</upml>
```
<function final="no" type="public" return="void">  
  <name>bar</name>  
  <argument></argument>  
</function>  

<statement>  
  <assign>  
    <object_name>  
      <variable>$this</variable>  
    </object_name>  
    <object_member>x</object_member>  
    <literal_string>'B'</literal_string>  
  </assign>  
</statement>
7. Conclusion and Future Work

In this work, we propose an XML-based markup language that represents programming techniques. We call the language Unified Programming Markup Language (UPML). The features of UPML can be summarized here:

a. UPML is an abstract representation of programming techniques. The representation of programming is based on programming paradigms such as structured programming and object-oriented programming, instead of the featured syntax of the programming language. Therefore, the abstraction makes it possible to store programming information in a program of any language. When a program from a programming
language is transformed to UPML, language styles are removed but the structure and semantic information are kept. When a program in UPML representation is to be transformed to a programming language, featured language syntax can be applied to the targeted code. This means, UPML can represent any programming languages whose paradigms are included in UPML.

b. Since UPML is based on XML, it is extensible. Therefore, theoretically, any programming paradigms can be added to UPML.

c. Our solution is solid based on the development of UPML languages, and we attempted to include programming features from the existing languages. Therefore, if a feature or function in one language is not in another language, we will absorb the feature or function in UPML, and produce the functionality in those languages that lack the feature or function. The predefined representation in UPML can simplify the translator because the process of the difference is not to be implemented in the translators.

d. For the same reason as in c, the translated result will be readable and not distorted as appeared in other translation solutions.

e. Using UPML as the gateway for translation among high-level programming languages, we can greatly decrease the numbers of translators. For example, if we implement translators directly between two languages, for ten of them, we need \(10 \times (10-1) = 90\) translators. With UPML, we need only \(2 \times 10 = 20\) translators, or translation modules if we embed the translation mechanism in one system as in our implementation.
f. Since UPML can inclusively represent most (if not all) programming techniques, UPML can hopefully be used as a general platform for software analysis. Software metrics can be applied to programs that are represented in UPML.

While UPML and its applications in programming language translation and general software analysis are appealing and promising, we still have much work to do to complete our expectation. Up to now we have drawn programming techniques only from PHP, and implemented only PHP to UPML transformer with implementation of only the core programming techniques. We can foresee much work needs to make UPML practical. The following list is what we think important to complete.

a. To complete UPML with features beyond the core programming techniques. For example, database support, function library, interface handlers with operating systems, are important content to be included in any representation of programming. Also, techniques in other popular languages such as Java, C/C++, and Python should be extracted and implemented in UPML.

b. Based on a more complete set of programming techniques in UPML, transformers from UPML to a programming language need to be implemented. This will be a direct proof of our assertion about the advantages of using UPML in translation.

c. Though imaginable, it is necessary to incorporate for software analysis for UPML.

d. The implemented PHP to UPML transformer needs to be expanded to support programming features beyond the core, such as function libraries and interfaces with other technologies.
In short, the work of the implementation of UPML and transformers based on UPML is still at the beginning stage. Much work needs to implement the functionality in the described system.
Bibliography


<table>
<thead>
<tr>
<th>Category</th>
<th>Section</th>
<th>Short Format</th>
<th>Definition</th>
<th>PHP</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>File</td>
<td><code>&lt;upml /&gt;</code></td>
<td>`&lt;upml language=&quot;language_short_name</td>
<td>file extension&quot; source=&quot;file_name&quot;&gt;`</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>`&lt;code /&gt;*</td>
<td>&lt;!-- --&gt;*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;/upml&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start/end</td>
<td></td>
<td><code>&lt;code /&gt;</code></td>
<td><code>&lt;code section=&quot;section_number&quot;&gt;</code></td>
<td></td>
<td>mark the start and the end of a section of code in a program file. A section of code may contain directives (declare, include, require, include_once, require_once, etc), classes, interfaces, functions, statements, etc.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;/code&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Class</td>
<td><code>&lt;class /&gt;</code></td>
<td>`&lt;class type=&quot;public</td>
<td>private</td>
<td>protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;/class&gt;</code></td>
<td>[abstract]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;name /&gt;</code></td>
<td>[public]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;inherit /&gt;?</code></td>
<td>[private]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;!-- possible inheritance from other classe(s)--&gt;</code></td>
<td>[protected]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;implement /&gt;?</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;!-- possible implement for interface--&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;body /&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;!--class body--&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;/class&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;name /&gt;</code></td>
<td><code>&lt;name&gt;string&lt;/name&gt;</code></td>
<td>NA</td>
<td>mark a name of any identify such as a class, a function, an interface, etc.</td>
</tr>
<tr>
<td></td>
<td>inherit</td>
<td><code>&lt;inherit /&gt;</code></td>
<td>`&lt;inherit type=&quot;public</td>
<td>private</td>
<td>protected&quot;&gt;`</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;!--the default class is public--&gt;</code></td>
<td>[public]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;list /&gt;</code></td>
<td>[private]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;!--a list fo parent classes</code></td>
<td>[protected]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>&lt;/list&gt;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|        |        | delimiter by comma-->
|        |        |     </inherit>

|        |        | comma list such as a list of argument
|        |        | a list of expressions delimited by comma or other characters.

|        |        | implement a_list_of_interface
|        |        | mark the implementation of interfaces.

| Interface | Interface | interface InterfaceName{
|           |           | interface_body
|           |           | }
|           |           | mark the definition of an interface.

| Function | Function | function final="yes|no"
type="constructor | destructor |
|          |          | general | static | abstract | public |
|          |          | return="data_type">
|          |          | !--final indicate the function is final or not, default is no -->
|          |          | !--default type is general-->
|          |          | !--return indicates the return data type-->
|          |          | name />
|          |          | !--function name-->
|          |          | list />
|          |          | !--argument list-->
|          |          | body />
|          |          | !--function body-->
|          |          | }
|          |          | __construct(),
|          |          | __destruct(), intval(),
|          |          | float(), strval(), array(),
|          |          | array($scalarValue),
|          |          | call_user_func(),
|          |          | usort(), echo(),
|          |          | in_array()
|          |          | mark the definition of various formats of functions.

| Statement | Body | statement, if, if..else,
|           |      | switch, while, do-while,
|           |      | for, foreach, declare,
|           |      | break, continue,
|           |      | default, require,
|           |      | a body of identifies such as a class, interface, function, program, etc consists of 0 or more statements.
<table>
<thead>
<tr>
<th><strong>Statement</strong></th>
<th>&lt;statement /&gt;</th>
<th>&lt;statement&gt; &lt;expression /&gt;</th>
<th>semicolon mark a statement</th>
<th>a simple statement with no control structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>If</strong></td>
<td>&lt;if /&gt;</td>
<td>&lt;if&gt; &lt;condition /&gt; &lt;body /&gt;</td>
<td>if(condition){ ...}</td>
<td>if structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Else</strong></td>
<td>&lt;else /&gt;</td>
<td>&lt;else&gt; &lt;body /&gt;</td>
<td>else{...}</td>
<td>else structure.</td>
</tr>
<tr>
<td><strong>Switch</strong></td>
<td>&lt;switch /&gt;</td>
<td>&lt;switch&gt; &lt;expression /&gt;</td>
<td>switch(condition)</td>
<td>switch structure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;case /&gt; *</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;default /&gt; ?</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>case</strong></td>
<td>&lt;case /&gt;</td>
<td>&lt;case&gt; &lt;expression /&gt;</td>
<td></td>
<td>case structure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;case&gt; &lt;!--literal value--&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>default</strong></td>
<td>&lt;default /&gt;</td>
<td>&lt;default&gt; &lt;statement /&gt;</td>
<td></td>
<td>default structure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>break;</td>
<td></td>
</tr>
<tr>
<td><strong>break</strong></td>
<td>&lt;break /&gt;</td>
<td>&lt;break /&gt;</td>
<td>break</td>
<td>break keyword.</td>
</tr>
<tr>
<td>continue</td>
<td>&lt;continue /&gt;</td>
<td>&lt;continue /&gt;</td>
<td>continue</td>
<td>continue keyword.</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>------------</td>
<td>--------</td>
<td>-----------------</td>
</tr>
<tr>
<td>while</td>
<td>&lt;while /&gt;</td>
<td>&lt;while /&gt;</td>
<td>while(...){} or while(...)...endwhile or do(...)while(..)</td>
<td>while structure or do - while structure</td>
</tr>
<tr>
<td>do-while</td>
<td>&lt;do-while /&gt;</td>
<td>&lt;do-while /&gt;</td>
<td>do{ .... }while(condition);</td>
<td>do-while structure</td>
</tr>
<tr>
<td>for</td>
<td>&lt;for /&gt;</td>
<td>&lt;for /&gt;</td>
<td>for(...) ; for(...)statement; for(...) {...}</td>
<td>for structure</td>
</tr>
<tr>
<td>foreach</td>
<td>&lt;foreach /&gt;</td>
<td>&lt;foreach /&gt;</td>
<td>foreach(...) ; foreach(...)statement; foreach(...) {...}</td>
<td>foreach structure.</td>
</tr>
<tr>
<td>Expression</td>
<td>&lt;expression /&gt;</td>
<td>&lt;expression /&gt;</td>
<td>expressions, operators, literal value, constants, variables, references, expressions, constructors, destructors, arrays</td>
<td>expressions.</td>
</tr>
<tr>
<td>Literal Expression</td>
<td>&lt;integer /&gt;</td>
<td>&lt;integer /&gt;</td>
<td>int, float, bool, char, string, decimal, hexadecimal, octal, binary, double, exponent, mixed, number, void</td>
<td>expressions of data types.</td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;float /&gt;</td>
<td>float number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;bool /&gt;</td>
<td>bool, boolean</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;char /&gt;</td>
<td>char</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;string /&gt;</td>
<td>string</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;decimal /&gt;</td>
<td>decimal number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;hexadecimal /&gt;</td>
<td>hexadecimal number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;octal /&gt;</td>
<td>octal number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;binary /&gt;</td>
<td>binary number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;double /&gt;</td>
<td>DNUM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;exponent /&gt;</td>
<td>exponent number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Description</td>
<td>Pseudo-type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;mixed /&gt;</code></td>
<td>Mixed type</td>
<td>mixed</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;number /&gt;</code></td>
<td>Number</td>
<td>number</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;callback /&gt;</code></td>
<td>Callback</td>
<td>callback</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;void /&gt;</code></td>
<td>Void</td>
<td>void</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td>Constant</td>
<td>constant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Control</td>
<td>\n (line feed), \r (carriage return), \t (tab), \v (vertical tab), \e (ESC)</td>
<td>string control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;LF /&gt;`</td>
<td>`&lt;LF /&gt;`</td>
<td>\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;CR /&gt;`</td>
<td>`&lt;CR /&gt;`</td>
<td>\r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;HT /&gt;`</td>
<td>`&lt;HT /&gt;`</td>
<td>\t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;VT /&gt;`</td>
<td>`&lt;VT /&gt;`</td>
<td>\v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;ESC /&gt;`</td>
<td>`&lt;ESC /&gt;`</td>
<td>\e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;FF /&gt;`</td>
<td>`&lt;FF /&gt;`</td>
<td>\f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;BACKSLASH /&gt;`</td>
<td>`&lt;BACKSLASH /&gt;`</td>
<td>\</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;DOLLAR /&gt;`</td>
<td>`&lt;DOLLAR /&gt;`</td>
<td>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;DOUBLEQUOTE /&gt;`</td>
<td>`&lt;DOUBLEQUOTE /&gt;`</td>
<td>&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;CHAR_OCTAL /&gt;`</td>
<td>`&lt;CHAR_OCTAL /&gt;`</td>
<td>[0-7]{1,3}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;CHAR_HEX /&gt;`</td>
<td>`&lt;CHAR_HEX /&gt;`</td>
<td>\x[0-9A-Fa-f]{1,2}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>`&lt;NULL /&gt;`</td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>More Parameter</td>
<td>`&lt;MORE_PARM /&gt;`</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>`&lt;variable /&gt;`</td>
<td>Variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Predefined Variables</td>
<td>`&lt;VAR_GLOBALS /&gt;`</td>
<td>Variables</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page 98
<table>
<thead>
<tr>
<th>Predefined Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SERVER, $HTTP_SERVER_VARS[deprecated]</td>
</tr>
<tr>
<td>PHP_SELF'</td>
</tr>
<tr>
<td>argv'</td>
</tr>
<tr>
<td>GATEWAY_INTERFACE'</td>
</tr>
<tr>
<td>SERVER_ADDR'</td>
</tr>
<tr>
<td>SERVER_NAME'</td>
</tr>
<tr>
<td>SERVER_SOFTWARE'</td>
</tr>
<tr>
<td>SERVER_PROTOCOL'</td>
</tr>
<tr>
<td>REQUEST_TIME'</td>
</tr>
<tr>
<td>REQUEST_TIME_FLOAT'</td>
</tr>
<tr>
<td>QUERY_STRING'</td>
</tr>
<tr>
<td>'DOCUMENT_ROOT'</td>
</tr>
<tr>
<td>'HTTP_ACCEPT'</td>
</tr>
<tr>
<td>'HTTP_ACCEPT_CHARSET'</td>
</tr>
<tr>
<td>'HTTP_ACCEPT_ENCODING'</td>
</tr>
<tr>
<td>'HTTP_ACCEPT_LANGUAGE'</td>
</tr>
<tr>
<td>'HTTP_CONNECTION'</td>
</tr>
<tr>
<td>'HTTP_HOST'</td>
</tr>
<tr>
<td>'HTTP_REFERER'</td>
</tr>
<tr>
<td>'HTTP_USER_AGENT'</td>
</tr>
<tr>
<td>'HTTPS'</td>
</tr>
<tr>
<td>'REMOTE_ADDR'</td>
</tr>
<tr>
<td>'REMOTE_HOST'</td>
</tr>
<tr>
<td>'REMOTE_PORT'</td>
</tr>
<tr>
<td>Variable Type</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>PATH_TRANSLATED</td>
</tr>
<tr>
<td>SCRIPT_NAME</td>
</tr>
<tr>
<td>REQUEST_URI</td>
</tr>
<tr>
<td>AUTH_DIGEST</td>
</tr>
<tr>
<td>AUTH_USER</td>
</tr>
<tr>
<td>AUTH_TYPE</td>
</tr>
<tr>
<td>PATH_INFO</td>
</tr>
<tr>
<td>ORIG_PATH_INFO</td>
</tr>
<tr>
<td>GET</td>
</tr>
<tr>
<td>POST</td>
</tr>
<tr>
<td>FILES</td>
</tr>
<tr>
<td>REQUEST</td>
</tr>
<tr>
<td>SESSION</td>
</tr>
<tr>
<td>ENV</td>
</tr>
<tr>
<td>Reference</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Function Expression</td>
</tr>
<tr>
<td>Constructor Expression</td>
</tr>
<tr>
<td>Destructor Expression</td>
</tr>
<tr>
<td>Argument List</td>
</tr>
<tr>
<td>Array</td>
</tr>
</tbody>
</table>

### Predefined Variables
- \$_COOKIE
- $HTTP_COOKIE_VARS [deprecated]
- $php_errormsg
- $HTTP_RAW_POST_DATA
- $http_response_header

### Reference
- Reference type: `integer | char | bool | float | string`
- Reference name

### Function Expression
- Function name (arguments)

### Constructor Expression
- Constructor name (argument)

### Destructor Expression
- Destructor name (argument)

### Argument List
- Argument names: `argument_name1, …, argument_nameN`

### Array
- Array representation: `[…]`
<table>
<thead>
<tr>
<th>Array Element</th>
<th>&lt; array_element /&gt;</th>
<th>&lt; array_element&gt;</th>
<th>&lt; name /&gt;</th>
<th>&lt; subscription &gt;</th>
<th>&lt; subscription &gt;</th>
<th>array_name subscrip-</th>
<th>array element</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>tion]</td>
<td></td>
</tr>
<tr>
<td>Left Expression</td>
<td>&lt; left_expression /&gt;</td>
<td>&lt; plus /&gt;</td>
<td>&lt; negate /&gt;</td>
<td>&lt; preincrease &gt;</td>
<td>&lt; predecrease &gt;</td>
<td>left operatos: +x, -x, ++x, --x, !</td>
<td></td>
</tr>
<tr>
<td>Positive Value</td>
<td>&lt;positive /&gt;</td>
<td>&lt;positive&gt;</td>
<td>&lt; expression /&gt;</td>
<td>plus sign: +x</td>
<td>Positive Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negative Value</td>
<td>&lt; negate /&gt;</td>
<td>&lt; negate&gt;</td>
<td>&lt; expression /&gt;</td>
<td>negative sign: -x</td>
<td>Negative Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prior Increment</td>
<td>&lt;pre_increment /&gt;</td>
<td>&lt;pre_increment&gt;</td>
<td>&lt; expression /&gt;</td>
<td>increment: ++</td>
<td>Prior Increment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prior Decrement</td>
<td>&lt;pre_decrement /&gt;</td>
<td>&lt;pre_decrement&gt;</td>
<td>&lt; expression /&gt;</td>
<td>decrement: --</td>
<td>Prior Decrement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Not</td>
<td>&lt;logical_negate /&gt;</td>
<td>&lt;logical_negate&gt;</td>
<td>&lt; expression /&gt;</td>
<td>logical negate: -</td>
<td>Logical Not</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitwise Complement</td>
<td>&lt;bit_complement /&gt;</td>
<td>&lt;bit_complement&gt;</td>
<td>&lt; expression /&gt;</td>
<td>bitwise complement</td>
<td>Bitwise Complement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Control</td>
<td>&lt;error_control /&gt;</td>
<td>&lt;error_control&gt;</td>
<td>&lt; expression /&gt;</td>
<td>Error control: @expression</td>
<td>Error Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expression Type</td>
<td>PHP Code</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------------------------</td>
<td>--------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Die Expression</strong></td>
<td><code>&lt;die&gt;</code></td>
<td>Die expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Return Expression</strong></td>
<td><code>&lt;return&gt;</code></td>
<td>Return expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Include or Require Expression</strong></td>
<td>`&lt;include</td>
<td>require&gt;`</td>
<td>Include or Require expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Require_once or Include_once Expression</strong></td>
<td>`&lt;include_once</td>
<td>require_once&gt;`</td>
<td>Require_once or Include_once expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Goto expression</strong></td>
<td><code>&lt;goto&gt;</code></td>
<td>Goto expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>New Object</strong></td>
<td>`&lt;clone</td>
<td>new&gt;`</td>
<td>New Object</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Throw Expression</strong></td>
<td><code>&lt;throw&gt;</code></td>
<td>Throw Expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cast to Int</strong></td>
<td><code>&lt;cast_int&gt;</code></td>
<td>Cast to Int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cast to Float</strong></td>
<td><code>&lt;cast_float&gt;</code></td>
<td>Cast to Float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cast to String</strong></td>
<td><code>&lt;cast_string&gt;</code></td>
<td>Cast to String</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cast to Array</strong></td>
<td><code>&lt;cast_array&gt;</code></td>
<td>Cast to Array</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cast to Object</strong></td>
<td><code>&lt;cast_object&gt;</code></td>
<td>Cast to Object</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cast to Null</td>
<td><code>&lt; cast_null /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>(unset)$var</code></td>
<td>Cast to Null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>--------------</td>
<td>--------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Right Expression</td>
<td><code>&lt; right_expression /&gt;</code></td>
<td><code>&lt; post_increment /&gt;</code></td>
<td><code>&lt; post_decrement /&gt;</code></td>
<td>post increment: x++</td>
<td>post decrement: x--</td>
<td>Right Expression</td>
<td></td>
</tr>
<tr>
<td>Post Increase</td>
<td><code>&lt; post_increment /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; post_increment /&gt;</code></td>
<td>post increment: x++</td>
<td>Post Increase</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Post Decrease</td>
<td><code>&lt; post_decrement /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; post_decrement /&gt;</code></td>
<td>post decrease: x--</td>
<td>Post Decrease</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Middle Expression</td>
<td><code>&lt; middle_expression /&gt;</code></td>
<td><code>&lt; add /&gt;</code></td>
<td><code>&lt; subtract /&gt;</code></td>
<td><code>&lt; multiply /&gt;</code></td>
<td><code>&lt; divide /&gt;</code></td>
<td><code>&lt; modulo /&gt;</code></td>
<td><code>&lt; equal /&gt;</code></td>
</tr>
<tr>
<td>Addition</td>
<td><code>&lt; add /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; add Assign /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td>x+y</td>
<td>Addition</td>
<td></td>
</tr>
<tr>
<td>Subtraction</td>
<td><code>&lt; subtract /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; subtract Assign /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td>x-y</td>
<td>Subtraction</td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>HTML Code</td>
<td>Example</td>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------</td>
<td>-----------</td>
<td>-------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplication</td>
<td><code>&lt; multiply /&gt;</code></td>
<td><code>x*y</code></td>
<td>Division</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Division</td>
<td><code>&lt; divide /&gt;</code></td>
<td><code>x/y</code></td>
<td>Modulus</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modulus</td>
<td><code>&lt; modulo /&gt;</code></td>
<td><code>x%y</code></td>
<td>Equality</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equality</td>
<td><code>&lt; equal /&gt;</code></td>
<td><code>x==y</code></td>
<td>Non Equality</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non Equality</td>
<td><code>&lt; not_equal /&gt;</code></td>
<td><code>x!=y</code></td>
<td>Identical</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identical</td>
<td><code>&lt; identical /&gt;</code></td>
<td><code>x===y</code></td>
<td>None Identical</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>None Identical</td>
<td><code>&lt; not_identical /&gt;</code></td>
<td><code>x!==y</code></td>
<td>Greater Than</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greater Than</td>
<td><code>&lt; greater /&gt;</code></td>
<td><code>&gt;</code></td>
<td>Less Than</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Less Than</td>
<td><code>&lt; less /&gt;</code></td>
<td><code>&lt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Syntax</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------</td>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greater or Equal</td>
<td><code>&lt; greater_equal</code></td>
<td><code>&gt;=</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Less or Equal</td>
<td><code>&lt; less_equal</code></td>
<td><code>&lt;=</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitwise And</td>
<td><code>&lt; bit_and</code></td>
<td><code>&amp;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitwise Or</td>
<td><code>&lt; bit_or</code></td>
<td>`</td>
<td>`</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitwise Xor</td>
<td><code>&lt; bit_xor</code></td>
<td><code>^</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Left Shift</td>
<td><code>&lt; left_shift</code></td>
<td><code>&lt;&lt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Right Shift</td>
<td><code>&lt; right_shift</code></td>
<td><code>&gt;&gt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Right Shift Fill Zero</td>
<td><code>&lt; right_shift_zero</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical And</td>
<td><code>&lt; logical_and</code></td>
<td><code>&amp;&amp;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Or</td>
<td><code>&lt; logical_or /&gt;</code></td>
<td><code>&lt; logical_or&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------</td>
<td>----------------</td>
<td>------------------</td>
<td>------------------</td>
<td>-----</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td><code>&lt; assign /&gt;</code></td>
<td><code>&lt; assign&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>=</code></td>
<td>Assignment</td>
<td></td>
</tr>
<tr>
<td>Addition then Assignment</td>
<td><code>&lt; add_assign /&gt;</code></td>
<td><code>&lt; add_assign&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>+=</code></td>
<td>Addition then Assignment</td>
<td></td>
</tr>
<tr>
<td>Subtraction then Assignment</td>
<td><code>&lt; subtract_assign /&gt;</code></td>
<td><code>&lt; subtract_assign&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>=-</code></td>
<td>Subtraction then Assignment</td>
<td></td>
</tr>
<tr>
<td>Multiplication then Assignment</td>
<td><code>&lt; multiply_assign /&gt;</code></td>
<td><code>&lt; multiply_assign&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>*=</code></td>
<td>Multiplication then Assignment</td>
<td></td>
</tr>
<tr>
<td>Division then Assignment</td>
<td><code>&lt; divide_assign /&gt;</code></td>
<td><code>&lt; divide_assign&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>/=</code></td>
<td>Division then Assignment</td>
<td></td>
</tr>
<tr>
<td>Modulus then Assignment</td>
<td><code>&lt; modulo_assign /&gt;</code></td>
<td><code>&lt; modulo_assign&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>%=</code></td>
<td>Modulus then Assignment</td>
<td></td>
</tr>
<tr>
<td>Concatenation</td>
<td><code>&lt; concatenate /&gt;</code></td>
<td><code>&lt; concatenate&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>.</code></td>
<td>Concatenation</td>
<td></td>
</tr>
<tr>
<td>Concatenation and Assignment</td>
<td><code>&lt; concatenate_assign /&gt;</code></td>
<td><code>&lt; concatenate_assign&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>&lt; expression /&gt;</code></td>
<td><code>.=</code></td>
<td>Concatenation and Assignment</td>
<td></td>
</tr>
<tr>
<td>Operator</td>
<td>Expression Type</td>
<td>Syntax</td>
<td>Result</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------</td>
<td>---------------------------------------------</td>
<td>----------------------</td>
<td>-------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Left Shift Assignment | < left_shift_assign /> | < left_shift_assign>    
< expression />    
< expression />    
</ left_shift_assign> | <<<=                       | Left Shift Assignment |
| Right Shift Assignment | < right_shift_assign /> | < right_shift_assign>    
< expression />    
< expression />    
</ right_shift_assign> | >>>=                       | Right Shift Assignment |
| Bit And Assignment | < bit_and_assign /> | < bit_and_assign>    
< expression />    
< expression />    
</ bit_and_assign> | &=                        | Bit And Assignment |
| Bit Xor Assignment | < bit_xor_assign /> | < bit_xor_assign>    
< expression />    
< expression />    
</ bit_xor_assign> | ^=                        | Bit Xor Assignment |
| Bit Or Assignment | < bit_or_assign /> | < bit_or_assign>    
< expression />    
< expression />    
</ bit_or_assign> | |=                        | Bit Or Assignment |
| Comma Divided Expression | < list delimiter="comma" /> | < list delimiter="comma">    
< literal|expression />    
---    
< literal|expression />    
</ list> | literal|expression,…,literal|expression | Comma Divided Expression |
<p>| Operator | Array Union | &lt; array_union /&gt; | array union: + | Array Union |
| Operator | Array Equality | &lt; array_equal /&gt; | array equal: == | Array Equality |
| Operator | Array Identity | &lt; array_identical /&gt; | array identical: === | Array Identity |</p>
<table>
<thead>
<tr>
<th>Category</th>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Inequality</td>
<td><code>&lt;array_inequality&gt;</code></td>
<td>array inequality: != or &lt;&gt;</td>
</tr>
<tr>
<td></td>
<td><code>&lt;array&gt;</code></td>
<td><code>array_inequality</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;array&gt;</code></td>
<td><code>array_inequality</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;array&gt;</code></td>
<td><code>array</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;array&gt;</code></td>
<td><code>array_inequality</code></td>
</tr>
<tr>
<td>Array non-identity</td>
<td><code>&lt;array_not_identical&gt;</code></td>
<td>array identical: !==</td>
</tr>
<tr>
<td></td>
<td><code>&lt;array&gt;</code></td>
<td><code>array_not_identical</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;array&gt;</code></td>
<td><code>array_not_identical</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;array&gt;</code></td>
<td><code>array_not_identical</code></td>
</tr>
<tr>
<td>Array Key-Value</td>
<td><code>&lt;array_key_value&gt;</code></td>
<td>array key-value for associative array: =&gt;</td>
</tr>
<tr>
<td></td>
<td><code>&lt;string&gt;</code></td>
<td><code>array_key_value</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;string&gt;</code></td>
<td><code>array_key_value</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;string&gt;</code></td>
<td><code>string</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;string&gt;</code></td>
<td><code>string</code></td>
</tr>
<tr>
<td>Array Identity</td>
<td><code>&lt;array_identical&gt;</code></td>
<td>array identical: ===</td>
</tr>
<tr>
<td></td>
<td><code>&lt;array&gt;</code></td>
<td><code>array_identical</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;array&gt;</code></td>
<td><code>array_identical</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;array&gt;</code></td>
<td><code>array_identical</code></td>
</tr>
<tr>
<td>Ternary</td>
<td><code>&lt;ternary&gt;</code></td>
<td>?::</td>
</tr>
<tr>
<td></td>
<td><code>&lt;condition&gt;</code></td>
<td><code>condition</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;condition&gt;</code></td>
<td><code>condition</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;expression&gt;</code></td>
<td><code>true value</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;expression&gt;</code></td>
<td><code>true value</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;expression&gt;</code></td>
<td><code>false value</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;expression&gt;</code></td>
<td><code>false value</code></td>
</tr>
<tr>
<td>Object operator</td>
<td><code>&lt;instanceof&gt;</code></td>
<td>object operator: instanceof</td>
</tr>
<tr>
<td></td>
<td><code>&lt;name&gt;</code></td>
<td><code>object name</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;name&gt;</code></td>
<td><code>object name</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;instanceof&gt;</code></td>
<td><code>class name</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;instanceof&gt;</code></td>
<td><code>class name</code></td>
</tr>
<tr>
<td>Others</td>
<td><code>&lt;execution&gt;</code></td>
<td><code>expression</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;expression&gt;</code></td>
<td><code>expression</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;expression&gt;</code></td>
<td><code>expression</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;expression&gt;</code></td>
<td><code>expression</code></td>
</tr>
</tbody>
</table>

Page 109
| Traits | < trait /> | < trait>  
|        |            | <string />  
|        |            | <!--trait-->  
|        |            | < trait_title-->  
|        |            | < trait_body>  
|        |            | < statement_group />  
|        |            | </ trait_body>  
|        |            | </ trait>  
| Try-Catch | < try_catch /> | < try_catch>  
|           |            | < try />  
|           |            | < catch />  
|           |            | [ < finally /> ]  
|           |            | </ try_catch>  
| Try block | < try /> | < try>  
|            |            | < statement_group />  
|            |            | </ try>  
| Catch block | < catch /> | < catch>  
|             |            | < exception />  
|             |            | <!--exception-->  
|             |            | < statement_group />  
|             |            | [ < catch /> ]  
|             |            | <!--when multiple catches are allowed-->  
|             |            | </ catch>  
| Exception block | < exception /> | < exception>  
|                 |            | <string />  
|                 |            | <!--exception type: exception class name-->  
|                 |            | < variable />  
|                 |            | </ exception>  
| Comment | < comment /> | < comment>  
|           |            | type="general|doc|heredoc|nowdoc">  
|           |            | <!--default is general-->  
|           |            | <string />  
|           |            | </ comment>  
|           |            | general comment: // or */..*/  
|           |            | PHPDoc Style document /** ...*/  
|           |            | Heredoc: <<<Heredoc_ID  
|           |            | ...  
|           |            | Heredoc_ID  
|           |            | Nowdoc: <<<'Nowdoc_ID'  
|           |            | Comment |
Appendix B. PHP to UPML Transformer Source Code

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=Cp1252">
<title>Transform the Uploaded File</title>
</head>

<body>
<?php

//#####The Start of Main Program

//#####start declaration of global variables

//receive the file from the upload

$uploadFile = $_FILES['sourcefile'];
$uploadFName = $_FILES['sourcefile']['name'];
$uploadFType = $_FILES['sourcefile']['type'];
$uploadFSize = $_FILES['sourcefile']['size'];
$uploadFTemp = $_FILES['sourcefile']['tmp_name'];
$uploadError = $_FILES['sourcefile']['error'];

$sourceFName = $_POST['fname'];
$sourceFExt = $_POST['fext'];

//file types allowed ['xml', 'php', 'js', 'java', 'c', 'cpp', 'py']
$allowType = array('xml', 'php', 'js', 'java', 'c', 'cpp', 'py');
$updir = 'uploads/';       // sets the folder where the uploaded files are copied
$max_size = 1000;         // sets maximum file size allowed (in KB)

//global arrays
$tokens = array();        //record tokens of the source code
$elementSeq_array = array(); //record sequence of elements statements, functions, classes; each has type/start
$countSection = -1;     //count the number of PHP code sections; as array subscription

Page 112
$countInterface = -1; //count the number of interfaces; as an array subscription
$countClass = -1; //count the number of classes; as an array subscription
$countFunction = -1; //count the number of functions; as an array subscription
$startSection = array(); //start of sections, record start position of a section
$endSection = array(); //end of sections, record end position of a section
$startInterface = array(); //start of Interface, record start position of a section
$endInterface = array(); //end of Interface, record end position of a section
$startClass = array(); //start of classes, record start position of a section
$endClass = array(); //end of classes, record end position of a section
$startFunction = array(); //start of functions, record start position of a section
$endFunction = array(); //end of functions, record end position of a section
$currentClassName = ""; //record current class name for identifying constructors and destructors
$currentSwitchEnd = 0; //record of current switch block, used to help case to mark its end
$braceArray = array(); //record location of pairs of "(" and ")"
$dBraceArray = array(); //dynamic record location of pairs of "(" and ")"
$bracketArray = array(); //record location of pairs of "[" and "]"
$dBracketArray = array(); //dynamic record location of pairs of "[" and "]"
$class_array = array(); //each element has name/start/end/abstract/parent/implement
$interface_array = array(); //each element has name/start/end/parent
$function_array = array(); //each element has
name/start/end/arguStart/arguEnd/declaration/definition/owner
$structure_array = array(); //record control structure element; each has keyWord/start/end/owner
$statement_array = array(); //each element has start/end/owner
$argument_array = array(); //record of arguments of functions; each as start/end/owner
$expression_array = array(); //record of expressions; each as start/end/owner
$expArray = array(); //an array to hold expressions inside round bracket

//global variables
$doMark = false; //mark do-while structure; when do meets, the value is true
$upmlText = array(); //resulting upml text (can be a few sections)
$htmlText = array(); //resulting html text (can be in a few sections)

//#####end declaration of global variables

//verify the uploaded file
verifySourceFile();

//read the content of the file to transform
$fileToTransform = fopen($updir.$uploadFName, 'r');
$contents = fread($fileToTransform, filesize($uploadFTemp));
fclose($fileToTransform);

//result file
$resultUPML = "created/".$sourceFName.".upml.xml"; //new algorithm
$stemUPML = fopen($resultUPML, 'w'); // new algorithm
//print UPML head
produceUPMLHead();
//identify top level elements in program: predictive, statements, functions, and classes
transformToUPML();
fclose($stemUPML); //close the resulting UPML file when finished.

#######The End of Main Program###############################################################

function verifySourceFile() // code referred at coursesweb.net
{
    global $uploadFile, $uploadFName, $uploadFType, $uploadFSize, $uploadFTemp, $uploadError;
    global $sourceFName, $sourceFExt, $contents, $allowType, $updir, $max_size;
    global $braceArray;

    // If is received a valid file from the 'sourcefile' form field
    if (isset($uploadFile))
    {
        // check for errors
        if ($uploadError > 0)
        {
            echo 'Error: '. $uploadError. '<br />';  
        }
    }
    else
    { /* get the name, size (in kb) and type (the extension) of the file */
        $fname = $uploadFName;
        $fsize = $uploadFSize / 1024;
        $ftype = end(explode('.', strtolower($fname)));
// checks if the file already exists
if (file_exists($updir. $fname))
{
    echo 'The file: '. $updir.$fname. ' already exists <br />';}
else
{// if the file not exists, check its type (by extension) and size
    if (in_array($ftype, $allowType))
    {
        // check the size
        if ($fsize <= $max_size)
        {
            // uses function to copy the file from temporary folder to $updir
            if (!move_uploaded_file ($uploadFTemp, $updir. $fname))
            {
                echo 'The file '. $fname. ' could not be copied, try again <br />';}
            else {
                echo $fname. ' ('. $fsize. ' kb) was successfully uploaded <br />';}
        }
        else {
            echo $fname. ' exceeds the maximum permitted size, '. $max_size. ' KB <br />';}}
    else {echo $fname. ' - invalid file type. <br />';}}
}
else echo 'Invalid form data. <br />';
}// end of function verify_source_file($fname, $ftype, $fsize, $ftemp, $ferror) definition

function transformToUPML()
{
    global $uploadFile, $uploadFName, $uploadFType, $uploadFSize, $uploadFTemp, $uploadError;
    global $sourceFName, $sourceFExt, $contents, $allowType, $updir, $max_size;
    global $braceArray;

    switch($sourceFExt)
    {
    case xml:
        //echo "identifying XML.<br />";
        //identifyXML();
        echo "transforming XML to UPML.";
        transformXMLToUPML();
        break;
    case php:
        //echo "identifying php.<br />";
        //identifyPHP();
        echo "transforming PHP to UPML.";
        transformPHPToUPML();
        break;
    case js:
        //echo "identifying JavaScript.<br />";
        //identifyJS();
        echo "transforming JavaScript to UPML.";
        transformJSToUPML();
        break;
    case java:
        //echo "identifying Java.<br />";
        //identifyJava();
        echo "transforming Java to UPML.";
        transformJavaToUPML();
        break;
    case c:
        //echo "identifying C.<br />";
//identifyC();

`echo "transforming C to UPML.";`
transformCToUPML();
break;

case cpp:
    `echo "identifying C++.<br />
//identifyCPP();
    echo "transforming C++ to UPML.";`
transformCPPToUPML();
break;

default:
    `echo "Cannot identify the language. <br />";`
}
//end of identifyStructure($uploadFile, $sourceType)

//+++++Start of TransformPHPtoUPML+++++
function transformPHPToUPML()
{
    global $uploadFile, $uploadFName, $uploadFType, $uploadFSize, $uploadFTemp, $uploadError;
    global $contents, $temFile, $tokens, $elementSeq_array, $class_array, $interface_array,
    $function_array, $structure_array, $statement_array, $argument_array, $expression_array;
    global $countSection, $startSection, $endSection;
    global $countInterface, $startInterface, $endInterface;
    global $countClass, $startClass, $endClass;
    global $countFunction, $startFunction, $endInterface;
    global $temUPML, $upmlText, $htmlText;
    global $currentState, $braceArray, $bracketArray;

    `echo "Source filesize- ".strlen($contents)."<br />";`

    // $tokens = array_values(array_filter($tokens = token_get_all($fileStr), "whiteSpace"));
    // remove items of T_WHITESPACE
    $tokens = array_values(array_filter($tokens = token_get_all($contents), "whiteSpace"));
    // remove items of T_WHITESPACE
    //echo "<pre><br />
    //print_r($tokens);
    //echo "<br /></pre>";
$braceArray= findBracePair($tokens);//find pairs of braces for future use
$bracketArray=findBracketPair($tokens);//find pairs of brackets for future use

//produceUPMLHead();

//SET UP ARRAYS FOR ELEMENTS:
//1. predictive: use include() function, therefore can be skipped
//2. statements:
//2.1 ownership - global, a function, a class, or an interface
//2.2 type:
//2.2.1 function expression:
//2.2.1.1 function name
//2.2.1.2 function argument list
//2.3.2 structured expression:
//2.3.2.1 if
//2.3.2.2 else
//2.3.2.3 switch/case/default
//2.3.2.4 for
//2.3.2.4.1 condition expressions: a list of statements
//2.3.2.4.2 for body
//2.3.2.5 foreach
//2.3.2.6 while
//2.3.2.7 do..while
//3. functions:
//3.1. name
//3.2. ownership - global, a class, interface(no definition)
//3.3. accessibility - default/public, protected, private
//3.4. arguments - a list of expressions - a comma list
//4. classes:
//4.1. abstract or not
//4.2. implement an interface or not
//4.3. extend parent classes or not
//5. interfaces:
//5.1 name
//5.2 extend parent interfaces or not
// ALGORITHM
// read the tokens one by one in sequence

$sub0 = 0;  // subscription of identified tokens of the program
$abs = false;  // mark a class as abstract or not
$inCode = 0;  // mark inside a section of PHP code or not
$levelFun = 0;  // mark of function: 0—not inside a function; 1—inside a function
$levelClass = 0;  // mark of class: 0—not inside a class; 1—inside a class
$levelInterface = 0;  // mark of interface: 0—not inside, 1—inside an interface

while ($sub0 < sizeof($tokens))  // identify each of the identified tokens
{
    $sub = $sub0;
    switch ($tokens[$sub][0])
    {
    case T_INLINE_HTML:
        $htmlText[$countSection+1] .= "<!--
";
        $htmlText[$countSection+1] .= $tokens[$sub][1]."\n";
        $htmlText[$countSection+1] .= "-->\n";
        break;
    case T_OPEN_TAG:// start a section of PHP code
        $countSection++;
        $startSection[$countSection]=$tokens[$sub][1];// section start line
        // To add: all html code between the end of last section to here
        $upmlText[$countSection] .= "<code section=""$countSection"">\n";
        $inCode++;
        // $sub++;
        break;
    case T_CLOSE_TAG:// end a section of PHP code
        $upmlText[$countSection] .= "</code>";
        $inCode--;
        // $sub++;
        break;
    case T_COMMENT:// comment: // or /*..*/
        $upmlText[$countSection] .= "<comment type=""general"">\n".$tokens[$sub][1]."</comment>\n";
        break;
    }
case T_DOC_COMMENT: //PHPDoc Style Comment: /**..*/
$upmlText[$countSection] .= "<comment type="doc">
	$tokens[$sub][1]."</comment>";
//@sub++;
break;

case T_START_HEREDOC: //PHP Here Doc start <<<EOT...EOT
$name="".$tokens[$sub+2][1]."">\n"
//search the start of this hereDoc content and end
$upmlText[$countSection] .= $tokens[$sub+1][1];
$upmlText[$countSection] .= "</comment>\n";
$sub += 2; /*move 2 tokens forward: hereDoc content and end*/
break;

case T_INTERFACE: //PHP interface
$levelInterface++; //enter a level of interface
$countInterface++; //a new Interface
$startInterface[$countInterface] = $tokens[$sub][2]; //record the interface start

@interfaceStartSub=$sub; //@sub of the interface
$interfaceEnd=identifyBodyEnd($sub); //search the end of this interface

//identify head of the interface
if($tokens[$interfaceStartSub-1][0]==T_PRIVATE) //access scope identifier
$upmlText[$countSection] .= "<interface type="private">\n";
else if($tokens[$interfaceStartSub-1][0]==T_PROTECTED)
$upmlText[$countSection] .= "<interface type="protected">\n";
else $upmlText[$countSection] .= "<interface type="public">\n";

//interface name
$interfaceBodyStart=identifyBodyStart($sub); //search the start of this interface body

interface body

identifyHeadExtension($interfaceStartSub, $interfaceBodyStart);
//identify body of the interface

identifyBody($interfaceBodyStart, $interfaceEnd); // $interfaceSub is the start "{", $sub is the end "}" of the body

//end of the interface
$upmlText[$countSection] .= "</interface>\n";
$sub=$interfaceEnd;
$levelInterface--; // exit a level of an interface
break;
case T_CLASS: // start of definition of a class
$levelClass++; // enter a level of class
$countClass++; // count a new class
// $startClass[$countClass]=$tokens[$sub][2]; // record the class start
$classStartSub=$sub; // $sub of the class
$classEnd=identifyBodyEnd($sub); // search the end of this class

// identify head of the interface
if($tokens[$classStartSub-1][0]==T_PRIVATE) // access scope identifier
    $upmlText[$countSection] .= "<class type="private">\n";
else if($tokens[$classStartSub-1][0]==T_PROTECTED)
    $upmlText[$countSection] .= "<class type="protected">\n";
else if($tokens[$classStartSub-1][0]==T_ABSTRACT)
    $upmlText[$countSection] .= "<class type="abstract">\n";
else if($tokens[$classStartSub-1][0]==T_FINAL)
    $upmlText[$countSection] .= "<class type="final">\n";
else
    $upmlText[$countSection] .= "<class type="public">\n";
$upmlText[$countSection] .= "<name>".$tokens[++$classStartSub][1]."</name>\n"; // class name

$classBodyStart=identifyBodyStart($sub); // search the start of the class body
identifyHeadExtension($classStartSub, $classBodyStart);

// identify body of the class
identifyBody($classBodyStart, $classEnd); // $classSub is the start "{", $sub
is the end "}" of the body

    // end of the class
    $upmlText[$countSection] .= "</class>
";
    $sub=$classEnd;
    $levelClass--; // exit a level of an class
    break;

    case T_FUNCTION: // start of a function definition
        $sub = identifyFunction($sub);
        break;

    case T_IF:
        $sub = identifyIfBlock($sub);
        break;

    case T_ELSE:
        $sub = identifyElseBlock($sub);
        break;

    case T_ELSEIF:
        $sub = identifyElseIfBlock($sub);
        break;

    case T_SWITCH:
        $sub = identifySwitchBlock($sub);
        $currentSwitchEnd = $sub; // used to help case identification
        break;

    case T_CASE:// ***not working yet. needs further attention
        $upmlText[$countSection] .= "<case>
";
        if($tokens[$sub+1][0]<> ":")
        {
            $caseValue = findCaseColon($sub+1);
            $upmlText[$countSection] .= "<case_value>
";
            $sub = identifyExpression($sub+1,$caseValue);
            $upmlText[$countSection] .= "</case_value>
";
        }
        $sub = identifyCaseBlock($sub);
        $upmlText[$countSection] .= "</case>
";
        break;

    case T_BREAK:
        $sub = identifyBreakBlock($sub);
        break;
case T_CONTINUE:
    $sub = identifyContinueBlock($sub);
    break;

case T_DEFAULT:
    //if($tokens[$sub+1][0]<> ":")
    // $sub = identifyExpression($sub,sizeof($tokens));
    $sub = identifyDefaultBlock($sub);
    break;

case T_WHILE:
    $sub = identifyWhileBlock($sub);
    break;

case T_DO:
    $sub = identifyDoBlock($sub);
    break;

case T_FOR:
    $sub = identifyForBlock($sub);
    break;

case T_FOREACH:
    $sub = identifyForEachBlock($sub);
    break;

case T_DECLARE:
    $sub = identifyDeclareBlock($sub);
    break;

case T_RETURN:
    $sub = identifyReturnStatement($sub);
    break;

case T_REQUIRE:
    $sub = identifyRequireStatement($sub);
    break;

case T_REQUIRE_ONCE:
    $sub = identifyRequireonceStatement($sub);
    break;

case T_INCLUDE:
    $sub = identifyIncludeStatement($sub);
    break;

case T_INCLUDE_ONCE:
    $sub = identifyIncludeOnceStatement($sub);
break;
case T_GOTO:
    $sub = identifyGotoStatement($sub);
break;
case T_VARIABLE:
    $sub = identifyVariableStartStatement($sub);
break;
case T_GLOBAL:
    $sub = identifyGlobalStartStatement($sub);
break;
case T_STATIC:
    $sub = identifyStaticStartStatement($sub);
break;
case T_ECHO:
    $sub = identifyEchoStatement($sub);
break;
case T_PRINT:
    $sub = identifyPrintStatement($sub);
break;
case T_STRING:
    if ($tokens[$sub][1] == "goto") //goto statement
    {
        $sub = identifyGotoStatement($sub);
break;
    }
    else if ($tokens[$sub+1][1] == ":") //label
    {
        $upmlText[$countSection] .= "<label>";
        $upmlText[$countSection] .= $tokens[$sub][1];
        $upmlText[$countSection] .= "</label>\
"
        $sub = $sub + 1; //move to the end of the label and its ":"
break;
    }
    else if ($tokens[$sub+1][1] == ")") //function expression
    {
        $agnd = identifyBraceEnd($sub+1);
        $upmlText[$countSection] .= "</function_expression>\
";
        break;
    }
case T_UNSET:
    $unsetEnd=identifyBraceEnd($sub+1);
    $upmlText[$countSection] .= "<unset>\n";
    identifyExpression($sub+2, $unsetEnd-1);
    $upmlText[$countSection] .= "</unset>\n";
    $sub = $unsetEnd+1;
    break;

default:
    break;
//end switch
$sub0 = $sub + 1;
//end while($sub0 < size of tokens)

closeResultUPMLFile();

}//end of function transformPHPToUPML()

//+++++End of TransformPHPtoUPML+++++

//++++++produceUPMLHead($file, $inputFile)++++++
function produceUPMLHead()
{
    global $temUPML, $uploadFName;
    fwrite($temUPML, "<?xml version="1.0"?>\n<upml language="php"
source="$uploadFName">\n");
    return;
}
function closeResultUPMLFile()
{
    global $upmlText, $resultUPML, $stemUPML, $countSection, $tokens, $contents, $htmlText;

    for($i = 0; $i < ($countSection+1); $i++)
    {
        fwrite($stemUPML, $htmlText[$i]."\n"); 
        fwrite($stemUPML, $upmlText[$i]."\n"); 
    }

    if($htmlText[$i]<>"") fwrite($stemUPML, $htmlText[$i]."\n");
    fwrite($stemUPML, "</upml>");
    fclose($stemUPML);

    echo "Click <a href="http://upml.net/code/$resultUPML">here</a> to view the UPML file.<br />

    return;
}

function whiteSpace($ar)
{
    if(is_string($ar)) return $ar;
    else
    {
         list($id, $text, $line) = $ar;
        if($id != T_WHITESPACE) return $ar;
        else return null;
    }
return;
}
// find the end position of an body of class, interface, and function
function identifyBodyEnd($subScription)
{
    global $FileName, $tokens;
    $numLeftCurve = 0;  // count '{' appeared after interface keyword
    $numRightCurve = 0; // count '} ' appeared after interface keyword
    do // look for equal number of '{' (start interface body) and '}' (end interface body)
    {
        if($tokens[$subScription] == '{')
            $numLeftCurve++;
        else if($tokens[$subScription] == '}')
            $numRightCurve++;
        ++$subScription;
    } while ((numLeftCurve == 0) || ($numLeftCurve <> $numRightCurve) || ($tokens[$subscription][0] == T_CLOSE_TAG));
    // find left curve, then the number of "{" and "}" is equal, or end of the PHP code section
    return $subScription-1; // return the location of sub of "}"
}

// find the end of a pair of brace, used for conditions, calculations, etc.
function identifyBraceEnd($subScription)
{
    global $FileName, $tokens;
    // echo "enter findEndBrace: ".$subScription."<br />
    $numLeftBrace = 0; // count '(' appeared after interface keyword
    $numRightBrace = 0; // count ')' appeared after interface keyword
    do // look for equal number of '(' (start interface body) and ')' (end interface body)
    {
        if($tokens[$subScription] == '(')
            $numLeftBrace++;
        else if($tokens[$subScription] == ')')
            $numRightBrace++;
        // echo "numLeftCurve: ".$numLeftBrace."<br />
        // echo "numRightCurve: ".$numRightBrace."<br />
        ++$subScription;
    } while (($numLeftBrace == 0) || ($numLeftBrace <> $numRightBrace)); // find left curve,
then the number of "{" and "}" is equal
   //echo "exit findEndBody".$subScription."<br />
   return $subScription-1; //return the sub location ")"
}

//find the end of a pair of bracket, used for arrays.
function identifyBracketEnd($subScription)
{
   global $temFile, $tokens;
   //echo "enter findEndBrace: ".$subScription."<br />
   $numLeftBracket = 0; //count ']' appeared after array name
   $numRightBracket = 0; //count '[' appeared after array name
   do
   {//look for equal number of [' (start interface body) and '] (end interface body)
   if($tokens[$subScription]=="[")
   $numLeftBracket++;
   else if($tokens[$subScription]=="]")
   $numRightBracket++;
   //echo "numLeftCurve: ".$numLeftBrace."<br /");
   //echo "numRightCurve: ".$numRightBrace."<br /");
   ++$subScription;
   }while(($numLeftBracket == 0) || ($numLeftBracket <> $numLeftBracket));//find left
curve, then the number of "[" and "]" is equal
   //echo "exit findEndBody".$subScription."<br />
   return $subScription-1; //return the sub location "]"
}//end of identifyBracketEnd

function identifyStatementEnd($subScription)
{
   global $tokens;
   $tempVar = $subScription;
   $countBrace = 0; //count "(" and ")", it should be 0 before ";"
   $countBracket = 0; //count "[" and "]", it should be 0 before ";"
   $countQuote = 0; //count "'", it should be even before ";"
   $countDblQuote = 0; //count """, it should be even even before ";"

   while($tokens[$tempVar]<>";");
|| $countBrace<>0
|| $countBracket<>0
|| $countQuote%2<>0
|| $countDblQuote%2<>0) // look for ";
{
    if ($tokens[$tempVar]=="(")$countBrace++;
    else if ($tokens[$tempVar]==")")$countBrace--;
    else if ($tokens[$tempVar]=="[")$countBracket++;
    else if ($tokens[$tempVar]=="]")$countBracket--;
    else if ($tokens[$tempVar]=="'")$countQuote++;
    else if ($tokens[$tempVar]==""\")$countDblQuote++;
    $tempVar++;
}

return $tempVar;
} // end of identifyStatementEnd($subscription)

// identify head extension and start of body ($headSub) of a class or an interface
function identifyHeadExtension($classInterfaceSt, $bodyStart)
{
    global $countSection, $tokens, $upmlText, $currentClassName;
    $locImplements = searchLoc($classInterfaceSt, $bodyStart, T_IMPLEMENTS);
    $locExtends = searchLoc($classInterfaceSt, $bodyStart, T_EXTENDS);
    if ($locImplements <> $classInterfaceSt && $locExtends <> $classInterfaceSt)
    {
        if ($locExtends < $locImplements)
        {
            $upmlText[$countSection] .= "< inherit >\n";
            $upmlText[$countSection] .= "< list dilimiter=\"comma\" >\n";
            for ($i = $locExtends+1; $i<$locImplements; $i++)
            {
                $upmlText[$countSection] .= $tokens[$i][1]." \n"; // an extended item
            }
            $upmlText[$countSection] .= "</ list >\n";
            }
$upmlText[$countSection] .= "</inherit>\n";

$upmlText[$countSection] .= "<implements>\n";
$upmlText[$countSection] .= "<list dilimiter="comma">\n";

for($i = $locImplements+1; $i<$bodyStart; $i++)
{
    $upmlText[$countSection] .= $tokens[$i][1]." \n"; //an implemented item
}
$upmlText[$countSection] .= "</list>\n";
$upmlText[$countSection] .= "</implements>\n";

else if($locImplements < $locExtends)
{
    $upmlText[$countSection] .= "<implments>\n";
$upmlText[$countSection] .= "<list dilimiter="comma">\n";
for($i = $locImplements+1; $i<$locExtends; $i++)
{
    $upmlText[$countSection] .= $tokens[$i][1]." \n"; //an implemented item
}
$upmlText[$countSection] .= "</list>\n";
$upmlText[$countSection] .= "</implments>\n";

$upmlText[$countSection] .= "<inherit>\n";
$upmlText[$countSection] .= "<list dilimiter="comma">\n";

for($i = $locExtends+1; $i<$bodyStart; $i++)
{
    $upmlText[$countSection] .= $tokens[$i][1]." \n"; //an extended item
}
$upmlText[$countSection] .= "</list>\n";
$upmlText[$countSection] .= "</inherit>\n";
}
}//end of both are present
else if ($locImplements <> $classInterfaceSt && $locExtends == $classInterfaceSt)
{
    $upmlText[$countSection] .= "<implemements>\n";
    $upmlText[$countSection] .= "<list dilimiter="comma">\n";
    for($i = $locImpelments+1; $i<$bodyStart; $i++)
    {
        $upmlText[$countSection] .= $tokens[$i][1]." \n"; //an implemented item
    }
    $upmlText[$countSection] .= "</list>\n";
    $upmlText[$countSection] .= "</implemements>\n";
}  //end of implements is present

else if ($locImplements == $classInterfaceSt && $locExtends <> $classInterfaceSt)
{
    $upmlText[$countSection] .= "<inherit>\n";
    $upmlText[$countSection] .= "<list dilimiter="comma">\n";
    for($i = $locExten+1; $i<$bodyStart; $i++)
    {
        $upmlText[$countSection] .= $tokens[$i][1]." \n"; //an extended item
    }
    $upmlText[$countSection] .= "</list>\n";
    $upmlText[$countSection] .= "</inherit>\n";
}  //end of extends is present

}  //end of identifyHeadExtension

//search location of keyword implement between keyword interface/class and the start of the interface/class body
function searchLoc($st, $nd, $key)
{
    global $tokens, $countSection, $upmlText;
    for($i=$st; $i<$nd; $i++)
        if($tokens[$i][0] == $key)return $i; //if find T_EXTENDS or T_IMPLMENTS
    return $st; //not find T_EXTENDS or T_IMPLMENTS
}

//identify the body of a class, an interface, a function, or a structure body
function identifyBody($bodyStart, $bodyEnd)
if($tokens[$bodyStart] == """)$sub0 = $bodyStart+1; //subscription of identified tokens of the program
else $sub0 = $bodyStart;
$abs = false; //mark a class as abstract or not
$inCode = 0; //mark inside a section of PHP code or not
$upmlText[$countSection] .= "<body>
";

while ($sub0 < $bodyEnd) //identify each of the identified tokens
{
    $sub = $sub0;
    switch ($tokens[$sub][0])
    {
        case T_COMMENT: //comment: // or /*...*/
            $upmlText[$countSection] .= "<comment type="general">
".$tokens[$sub][1]."</comment>
";
            break;
        case T_DOC_COMMENT: //PHPDoc Style Comment: /**...*/
            $upmlText[$countSection] .= "<comment type="doc">
".$tokens[$sub][1]."</comment>
";
            break;
        case T_START_HEREDOC: //PHP Here Doc start <<<EOT...EOT !!! more work needed
            $upmlText[$countSection] .= "<comment type="heredoc">
".$tokens[$sub+2][1]."</comment>
";
            break;
    }
}
$upmlText[$countSection] .= "</comment>
";
$sub += 2;  //move 2 tokens forward: hereDoc content and end
break;
case T_FUNCTION:  //start of a function definition
    $sub = identifyFunction($sub);
    break;
case T_IF:
    $sub = identifyIfBlock($sub);
    break;
case T_ELSE:
    $sub = identifyElseBlock($sub);
    break;
case T_ELSEIF:
    $sub = identifyElseIfBlock($sub);
    break;
case T_SWITCH:
    $sub = identifySwitchBlock($sub);
    $currentSwitchEnd = $sub;  //used to help case identification
    break;
case T_CASE://***not working yet. needs further attention
    $upmlText[$countSection] .= "<case>
";
    if ($tokens[$sub+1][0] <> ":")
    {
        $caseColon = findCaseColon($sub+1);
        //echo "caseColon="."$caseColon."<br />
"
        $upmlText[$countSection] .= "<case_value>
";
        $sub = identifyExpression($sub+1,$caseColon-1);
        $upmlText[$countSection] .= "</case_value>
";
    }
    $sub = identifyCaseBlock($sub);
    $upmlText[$countSection] .= "</case>
";
    break;
case T_BREAK:
    $sub = identifyBreakBlock($sub);
    break;
case T_CONTINUE:
    $sub = identifyContinueBlock($sub);
break;
case T_DEFAULT:
$sub = identifyDefaultBlock($sub);
break;
case T_WHILE:
$sub = identifyWhileBlock($sub);
break;
case T_DO:
$sub = identifyDoBlock($sub);
break;
case T_FOR:
//echo "in For:";
$sub = identifyForBlock($sub);
break;
case T_FOREACH:
$sub = identifyForEachBlock($sub);
break;
case T_DECLARE:
$sub = identifyDeclareBlock($sub);
break;
case T_RETURN:
$sub = identifyReturnStatement($sub);
break;
case T_REQUIRE:
$sub = identifyRequireStatement($sub);
break;
case T_REQUIRE_ONCE:
$sub = identifyRequireonceStatement($sub);
break;
case T_INCLUDE:
$sub = identifyIncludeStatement($sub);
break;
case T_INCLUDE_ONCE:
$sub = identifyIncludeOnceStatement($sub);
break;
case T_GOTO:
$sub = identifyGotoStatement($sub);
break;

case T_VARIABLE:
    //if($varScope<>"")$upmlText[$countSection] .= "<".$varScope.">
"
    $sub = identifyVariableStartStatement($sub);
    //if($varScope<>"")$upmlText[$countSection] .= "</".$varScope.">
"
    break;

case T_GLOBAL:
    $sub = identifyGlobalStartStatement($sub);
    break;

case T_STATIC:
    $sub = identifyStaticStartStatement($sub);
    break;

case T_ECHO:
    $sub = identifyEchoStatement($sub);
    break;

case T_PRINT:
    $sub = identifyPrintStatement($sub);
    break;

case T_STRING:
    if($tokens[$sub][1]=='goto') //goto statement
    {
        $sub = identifyGotoStatement($sub);
        break;
    }
    else if($tokens[$sub+1]==':') //label
    {
        $upmlText[$countSection] .= "<label>
"
        $upmlText[$countSection] .= $tokens[$sub][1];
        $upmlText[$countSection] .= "</label>
"
        $sub = $sub + 1; //move to the end of the label and its ":"
    }
    else if($tokens[$sub+1]=='(') //function expression
    {
        //echo "entered T_STRING\n"
        $agnd = identifyBraceEnd($sub+1);
$upmlText[$countSection] .= "<function_expression>\n";
$upmlText[$countSection] .= "<name>\n";
$upmlText[$countSection] .= $tokens[$sub][1]."\n";
$upmlText[$countSection] .= "</name>\n";
$upmlText[$countSection] .= "<argument>\n";
identifyExpression($sub+2, $agnd-1);
$upmlText[$countSection] .= "</argument>\n";
$upmlText[$countSection] .= "</function_expression>\n";
$sub = $agnd;
}
break;
case T_UNSET:
    $unsetEnd=identifyBraceEnd($sub+1);
    $upmlText[$countSection] .= "<unset>\n";
    identifyExpression($sub+2, $unsetEnd-1);
    $upmlText[$countSection] .= "</unset>\n";
    $sub = $unsetEnd+1;
    break;

default:
    break; //end switch
$sub0 = $sub + 1;
}//end while()
$upmlText[$countSection] .= "</body>\n";

}//end of identifyBody

//identify end of case value such as "case A:"
function identifyCase($caseValue)
{
    global $tokens, $upmlText, $countSection;
    $tempVar = $caseValue;
    while($tokens[$tempVar]<":" && $tempVar <sizeof($tokens))
    {
        $tempVar++;
    }
}
return $tempVar;
}

// identify a function definition
function identifyFunction($funStart)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;

    $levelFunction++; // enter a level of function
    $countFunction++; // count a new function

    $returnType = "void"; // detect return data type of the function

    $functionStart = $funStart;
    $functionEnd = identifyBodyEnd($functionStart);
    // *** return data type identification needs more work
    // $dataType = findReturnType($functionStart, $functionEnd); // find return in the function to know the return data type
    if ($tokens[$functionStart-1][0]==T_PUBLIC)
        $functionType = "public";
    else if ($tokens[$functionStart-1][0]==T_PROTECTED)
        $functionType = "protected";
    else if ($tokens[$functionStart-1][0]==T_PRIVATE)
        $functionType = "private";
    else if ($tokens[$functionStart-1][0]==T_ABSTRACT || $tokens[$functionStart-2][0]==T_ABSTRACT)
        $functionType = "abstract";
    else if ($tokens[$functionStart-1][0]==T_STATIC || $tokens[$functionStart-2][0]==T_STATIC)
        $functionType = "static";
    else if ($tokens[$functionStart+1][1]=="__construct") // function name is _construct
        $functionType = "constructor";
    else if ($levelClass==1 && strcmp($tokens[$functionStart+1][1], $currentClassName)== 0)
        $functionType = "constructor";
    else if ($tokens[$functionStart+1][1]=="__destruct") // function name is _destruct
        $functionType = "constructor";
    else if ($tokens[$functionStart+1][1]=="__construct") // function name is _construct
        $functionType = "constructor";
$functionType = "destructor";
else $functionType = "general";

if($tokens[$functionStart-1][0]==T_FINAL || $tokens[$functionStart-2][0]==T_FINAL)$finalMark="yes";
else $finalMark = "no";

for($i=$functionStart; $i<=$functionEnd; $i++)
{
    if($tokens[$i][0]==T_RETURN && $tokens[$i+1][0]<>";"
    {
        $returnType = gettype($tokens[$i+1][0]);
    }
}

$upmlText[$countSection] .= "<function final=""$finalMark." type=""$functionType." return=""$returnType."">"
//function head
$upmlText[$countSection] .= "<name>".$tokens[$functionStart+1][1]."</name>";
//function name

//identify function argument lists
$argumentStart = $functionStart + 2; //function funName(arguList){statements}
$argumentEnd = identifyBraceEnd($argumentStart);
//process function argument list
$upmlText[$countSection] .= "<argument>"
identifyExpression($argumentStart, $argumentEnd);
$upmlText[$countSection] .= "</argument>";

//process function body
$funBodyStart = $argumentEnd+1; //start at "{" or ";"

if($tokens[$funBodyStart][0]==";")$funBodyEnd=$funBodyStart;//no real body
else $funBodyEnd = identifyBodyEnd($funBodyStart);//end at "}"

if($funBodyStart <> $funBodyEnd)
    identifyBody($funBodyStart, $funBodyEnd);
$upmlText[$countSection] .= "</function>\n";

$levelFunction--; // exit function body
$functionEnd = $funBodyEnd;
return $functionEnd;
}

// identify if block
function identifyIfBlock($ifTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;

    $ifStart = $ifTokSub;
    //$ifEnd = identifyBodyEnd($ifStart);

    // identify if condition
    $conditionStart = $ifStart + 1; // position sub of "(
    $conditionEnd = identifyBraceEnd($conditionStart); // position sub of ")"
    $upmlText[$countSection] .= "<if>\n";
    // process if condition
    $upmlText[$countSection] .= "<condition>\n";
    // identify comparison expression
    identifyExpression($conditionStart, $conditionEnd);
    $upmlText[$countSection] .= "</condition>\n";

    // process if body
    $ifBodyStart = $conditionEnd+1; // start at "{" of start of the single statement
    if($tokens[$ifBodyStart]=="{")
        $ifBodyEnd = identifyBodyEnd($ifBodyStart); // end at "}"
    else // a single statement with no "{" and "}"
        $ifBodyEnd = identifyStatementEnd($ifBodyStart); // end at ";"
    identifyBody($ifBodyStart, $ifBodyEnd);
    // 1.single empty statement
    // 2.single statement
//3.in "{" and "}"
$upmlText[$countSection] .= "</if>\n";

$ifEnd = $ifBodyEnd;

return $ifEnd;
}//end of identifyIfBlock

//identify Else block
function identifyElseBlock($elseTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;

    $elseStart = $elseTokSub;
    //elseEnd = identifyBodyEnd($elseStart);
    
    $upmlText[$countSection] .= "<else>\n"; //start transforming else block

    if($tokens[$elseStart+1][0]== T_IF)//a if block after else
    {
        $ifBlockEnd=identifyIfBlock($elseStart+1);
        $upmlText[$countSection] .= "</else>\n";
        $elseEnd = $ifBlockEnd;
        //elseEnd = $elseStart; //go to end so that identifyIfBlock will work
        return $elseEnd;
    }
    else
    {
        //process if body
        $elseBodyStart = $elseStart+1; //start at "{" of start of the single statement
        if($tokens[$elseBodyStart]==""
            $elseBodyEnd = identifyBodyEnd($elseBodyStart); //end at "}"
        else //a single statement with no "{" and "}"
            $elseBodyEnd = identifyStatementEnd($elseBodyStart); //end at ";"
        identifyBody($elseBodyStart, $elseBodyEnd);
        //1.single empty statement (",")
    }
}
//2. single statement
//3. in "{" and "}"
$upmlText[$countSection] .= "</else>\n";

$elseEnd = $elseBodyEnd;
return $elseEnd;
}

//return $elseEnd;
}//end of identifyElseBlock

//identify elseif block
function identifyElseIfBlock($elseifTokSub) {
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    $elseifStart = $elseifTokSub;
    $upmlText[$countSection] .= "</else>\n"; //start transforming else block
    $ifBlockEnd = identifyIfBlock($elseifStart); //go to if block, if and else is the same
    here: elseif
    $upmlText[$countSection] .= "</else>\n";
    $elseifEnd = $ifBlockEnd;
    // $elseEnd = $elseStart; // go to end so that identifyIfBlock will work
    return $elseifEnd;
} //end of identifyElseIfBlock

//identify switch block
function identifySwitchBlock($switchTokSub) {
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    $switchStart = $switchTokSub;
    $switchEnd = identifyBodyEnd($switchStart); // go to end so that identifySwitchBlock will work
    return $switchEnd;
} //end of identifySwitchBlock
//identify switch condition
$conditionStart = $switchStart + 1; //postion sub of "(
$conditionEnd = identifyBraceEnd($conditionStart); //position sub of ")"
$upmlText[$countSection] .= "<switch>
";
//process switch condition
$upmlText[$countSection] .= "<expression>
";
    //identify comparison expression
    identifyExpression($conditionStart, $conditionEnd); //process switch body
$upmlText[$countSection] .= "</expression>
";
//process switch body
$switchBodyStart = $conditionEnd+1; //start at "{" of start of the single statement
$switchBodyEnd = identifyBodyEnd($switchBodyStart); //end at "}"
$currentSwitchEnd = $switchBodyEnd;
//echo "switchBodyStart=".$switchBodyStart." switchBodyEnd=".$switchBodyEnd."<br />";
identifyBody($switchBodyStart, $switchBodyEnd);
    //1. case statement
    //2. break statement
    //3. continue statement
    //4. default statement
$upmlText[$countSection] .= "</switch>
";

$switchEnd = $switchBodyEnd;

    return $switchEnd;
} //end identify switch block

//identify case block
function identifyCaseBlock($caseTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
        $currentClassName, $countSection;
    global $currentPage

    //find ":" as the end of the head each case
    $caseHeadEnd = $caseTokSub; //this is the position of colon
    //this is the position of colon
    $caseHeadEnd = findCaseColon($caseHeadStart);
//echo "caseHeadEnd=".$caseHeadEnd."<br />");
if($tokens[$caseHeadEnd+1]=="{="/if case body inside "{" and "}"
{
    $caseBodyStart = $caseHeadEnd+1;
    $caseBodyEnd = identifyBodyEnd($caseBodyStart); //end at "}"
    //echo "in if, caseBodyEnd=".$caseBodyEnd."<br />");
}
else //no "{" and "}" surrounding the case body
{
    $caseBodyStart = $caseHeadEnd + 1; //start at case head ":"
    //case body end at either continue, break, default, or the next case
    $caseBodyEnd = $caseBodyStart; //caseBodyEnd gets temporary value
    if((i = findCaseEnd($caseBodyEnd))>$caseBodyStart)
        $caseBodyEnd = $i;
    else
    {
        $caseBodyEnd = $currentSwitchEnd; //if no continue, break, default, or case
        follows, use the end of current Switch
        echo "*Possibly missed BREAK, CONTINUE, and DEFAULT in a switch-case at
token number: ".$caseTokSub."<br />");
        echo "or Case block is out of the current switch block.<br />");
    }
    //echo "in else, caseBodyEnd=".$caseBodyEnd."<br />");
}
//echo "caseBodyStart=".$caseBodyStart."<br />");
//echo "caseBodyEnd=".$caseBodyEnd."<br />");

identifyBody($caseBodyStart, $caseBodyEnd);
return $caseBodyEnd;
}//end of identifyCaseBlock($caseTokSub)

//help function for identifyCaseBlock - find next continue in current switch
function findCaseEnd($caseBodyEnd)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,$currentClassName, $countSection;

    Page 143
global $currentSwitchEnd;

//echo "in findCaseEnd, caseBodyEnd="$caseBodyEnd."<br ">
//echo "in findCaseEnd, currentSwitchEnd="$currentSwitchEnd."<br ">

for($i=$caseBodyEnd; $i<$currentSwitchEnd; $i++)
{
    if($tokens[$i][0]==T_CONTINUE) return $i+1; //location of continue
    else if($tokens[$i][0]==T_BREAK) return $i+1; //location of break
    else if($tokens[$i][0]==T_DEFAULT) return $i+1; //location of default
    else if($tokens[$i][0]==T_CASE)
    {
        echo "Possibly missed BREAK, CONTINUE, and DEFAULT in a switch-case at token number: ".$caseTokSub."<br ">
        return $i-1; //location of next case
    }
}
//not find continue, break, break, or case
//echo "caseBodyEnd in findCaseEnd="$caseBodyEnd."<br ">
return $caseBodyEnd;

//find case head colon
function findCaseColon($caseTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    $num = $caseTokSub+1;
    while($tokens[$num] <> ":" && $num < sizeof($tokens))$num++;
    //echo "case colon position: ".$num."<br ">
    return $num;
}

//identify break block
function identifyBreakBlock($breakTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
$currentClassName, $countSection;
    upmlText[$countSection] .= "<break />\n";
    return $breakTokSub;
  }

//identify continue block
function identifyContinueBlock($continueTokSub)
{
  global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
            $currentClassName, $countSection;
  upmlText[$countSection] .= "<continue />\n";
  return $continueTokSub;
}

//identify default block
function identifyDefaultBlock($defaultTokSub)
{
  global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
            $currentClassName, $countSection;
  global $currentSwitchEnd;

  upmlText[$countSection] .= "<default>\n";
  // ":" is the end of the head of default
  $defaultHeadStart = $defaultTokSub;
  $defaultHeadEnd = identifyBodyEnd($defaultHeadStart);
  //end at "}"
  if($tokens[$defaultHeadEnd+1]=="{")//if default body inside "{" and "}"
  {
    $defaultBodyStart = $defaultHeadEnd+1;
    $defaultBodyEnd = identifyBodyEnd($defaultBodyStart);//end at "}"
  }
  else //no "{" and "}" surrounding the case body
  {
    $defaultBodyStart = $defaultHeadEnd; //start at case head ":
    //case body end at either continue, break, default, or the next case
    $defaultBodyEnd = $currentSwitchEnd; //caseBodyEnd gets temporary value
  }

Page 145
identifyBody($defaultBodyStart, $defaultBodyEnd);

    $upmlText[$countSection] .= "</default>\n";
    return $defaultBodyEnd;
} //end of identifyCaseBlock($caseTokSub)

//identify while block
function identifyWhileBlock($whileTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;

    $whileStart = $whileTokSub;

    //identify while condition
    $conditionStart = $whileStart + 1; //postion sub of "(
    $conditionEnd = identifyBraceEnd($conditionStart); //position sub of ")

    if($tokens[$conditionEnd+1]==":")$whileEnd = identifyWhileEnd($conditionEnd+1);
    else $whileEnd = identifyBodyEnd($whileStart); //while body in "{" and "}"

    $upmlText[$countSection] .= "<while>\n";
    //process while condition
    $upmlText[$countSection] .= "<condition>\n";
    //identify comparison expression
    identifyExpression($conditionStart, $conditionEnd);
    $upmlText[$countSection] .= "</condition>\n";

    //process while body
    $whileBodyStart = $conditionEnd+1; //start at "{" of start of the single statement
    $whileBodyEnd = $whileEnd; //identifyBodyEnd($whileBodyStart); //end at "}"
    identifyBody($whileBodyStart, $whileBodyEnd);

    $upmlText[$countSection] .= "</while>\n";

    //$whileEnd = $whileBodyEnd;
return $whileEnd;
}

//identify the endwhile of while loop
function identifyWhileEnd($tokSub)
{
    global $tokens;
    //
    //echo "enter findEndBody: ".$subScription."<br />
    $i = $tokSub;
    while($tokens[$i][0] <> T_ENDWHILE) //look for equal number of '{' (start interface body) and '}' (end interface body)
    {
        $i++;
        if($i>100000)return $i; //force to stop with a big number
    }
    return $i; //return the location of sub of "}
}

//identify the do-while loop
function identifyDoBlock($doTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;

    $doStart = $doTokSub;

    $doBodyStart = $doStart+1; //body start at "{
    $doBodyEnd = identifyBodyEnd($doBodyStart); //body end at "}

    $doConditionStart = $doBodyEnd+2; //do{...}while(..), from "}" to "{
    $doConditionEnd = identifyBraceEnd($doConditionStart); //identify "

    $upmlText[$countSection] .= "<do>
    //do body
    identifyBody($doBodyStart, $doBodyEnd);

    //process while condition
$upmlText[$countSection] .= "<while>
";
//identify comparison expression
identifyExpression($doConditionStart, $doConditionEnd);
$upmlText[$countSection] .= "</while>
";

$upmlText[$countSection] .= "</do>
";
$doEnd = $doConditionEnd;

return $doEnd;

 }//end do-while block

//identify for loop
function identifyForBlock($forTokSub)
{

global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
$currentClassName, $countSection;

$forStart = $forTokSub;
//echo "entered identifyForBlock, forStart=".$forStart."<br />";
//identify for condition
$conditionStart = $forStart + 1; //position sub of "(
$conditionEnd = identifyBraceEnd($conditionStart); //position sub of ")"

$forBodyStart = $conditionEnd + 1;

if($tokens[$forBodyStart]==="{")/for body in "{" and "}"
    $forBodyEnd = identifyBodyEnd($forBodyStart); //identify ")"
else $forBodyEnd = identifyStatementEnd($forBodyStart); //identify for body of a single statement

$upmlText[$countSection] .= "<for>
";
//process for condition
$upmlText[$countSection] .= "<condition>
";
//identify comparison expression
//echo "in for, conditionStart=".$conditionStart." conditionEnd=".$conditionEnd."<br
/>";
identifyForCondition($conditionStart, $conditionEnd);
$upmlText[$countSection] .= "</condition>\n";

//process for body
identifyBody($forBodyStart, $forBodyEnd);

$upmlText[$countSection] .= "</for>\n";

$forEnd = $forBodyEnd;

return $forEnd;
}

//identify for condition
function identifyForCondition($st, $nd)
{
    global $tokens, $countSection, $upmlText;

    $t = 0; //record the times of occurrence of semicolon in "for" condition
    $loc = $st; //temporary record of semicolon in for condition
    $brcArray = findDynamicBracePair($st+1, $nd-1); //find braces in for condition
    $brkArray = findDynamicBracketPair($st+1, $nd-1); //find brackets in for condition
    for($i = $st; $i<$nd; $i++)
    {
        if($tokens[$i]==";")
        {
            if($t == 0 && !inBraces($i, $st+1, $nd-1, $brcArray) && !inBrackets($i, $st+1, $nd-1, $brkArray))
            {
                $upmlText[$countSection] .= "<initial_condition>\n";
                identifyExpression($st+1, $i-1);
                $upmlText[$countSection] .= "</initial_condition>\n";
                $t++;
            loc = $i;
        }
        else if($t == 1 && !inBraces($i, $st+1, $nd-1, $brcArray) && !inBrackets($i, $st+1, $nd-1, $brkArray))
    {
function identifyForEachBlock($foreachTokSub) {
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
            $currentClassName, $countSection;

    $foreachStart = $foreachTokSub;

    //identify foreach condition
    $conditionStart = $foreachStart + 1; //position sub of "(
    $conditionEnd = identifyBraceEnd($conditionStart); //position sub of ")"

    $foreachBodyStart = $conditionEnd + 1;
    //echo "entered foreach<br />
    //echo "foreachBodyStart=".$foreachStart."<br />
    //echo "conditionStart=".$conditionStart."<br />
    //echo "conditionEnd=".$conditionEnd."<br />

    if($tokens[$foreachBodyStart]=="{") //for body in "{" and "}"
        $foreachBodyEnd = identifyBodyEnd($foreachBodyStart); //identify ")"
    else $foreachBodyEnd = identifyStatementEnd($foreachBodyStart); //identify for body of a

    //echo "foreachBodyEnd=".$foreachBodyEnd."<br />
    $upmlText[$countSection] .= "<foreach>

    
}
//process for condition
$upmlText[$countSection] .= "<expression>\n";
//identify comparison expression
identifyForeachCondition($conditionStart, $conditionEnd);
$upmlText[$countSection] .= "</expression>\n";

//process for body
identifyBody($foreachBodyStart+1, $foreachBodyEnd-1);

$upmlText[$countSection] .= "</foreach>\n";
$foreachEnd = $foreachBodyEnd;

return $foreachEnd;
)//end of identifyForeachBlock

function identifyForeachCondition($st, $nd) {
    global $tokens, $upmlText, $countSection;
    $pos=$st;
    for($i = $st; $i<=$nd; $i++) {
        if($tokens[$i][0] == T_AS) {
            $pos = $i;
            break;
        }
        else if($i==$nd) {
            echo "possible missing of AS in foreach condition\n";
        }
    }
    $upmlText[$countSection] .= "<as>\n";
    identifyExpression($st+1, $i-1);
    identifyExpression($i+1, $nd-1);
    $upmlText[$countSection] .= "</as>\n";
}

Page 151
//identify declare block
function identifyDeclareBlock($declareTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;

    $declareStart = $declareTokSub;

    //identify foreach condition
    $conditionStart = $declareStart + 1; //postion sub of "("
    $conditionEnd = identifyBraceEnd($declareStart); //position sub of ")"

    $declareBodyStart = $conditionEnd + 1;

    if($tokens[$declareBodyStart]=="{" //for body in "{" and "}"
        $declareBodyEnd = identifyBodyEnd($declareBodyStart); //identify "}"
    else $declareBodyEnd = identifyStatementEnd($declareBodyStart);//identify for body of a
    single statement

    $upmlText[$countSection] .= "<declare>
"
    //process declare condition
    $upmlText[$countSection] .= "<expression>
"
    //identify expression
    identifyDeclareCondition($conditionStart, $conditionEnd);
    $upmlText[$countSection] .= "</expression>
"

    //process declare body
    //****identifyBody($declareStart, $declareBodyEnd) -- body of the program. So we can
    omit this body here.
    $upmlText[$countSection] .= "</declare>
"

    $declareEnd = $declareBodyEnd;

    return $declareEnd;
}
function identifyDeclareCondition($st, $nd)
{
    identifyExpression($st, $nd);
}

//identify Return statement
function identifyReturnStatement($returnTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;

    $returnStart = $returnTokSub;
    $returnBodyStart = $returnStart+1;
    $returnBodyEnd = identifyStatementEnd($returnBodyStart);
    $upmlText[$countSection] .= "<return>
";

    //return body
    identifyExpression($returnBodyStart, $returnBodyEnd - 1);//the end is the semicolon, so subtract 1
    $upmlText[$countSection] .= "</return>
";

    $returnEnd = $returnBodyEnd;
    return $returnEnd;
}

//identify require statement
function identifyRequireStatement($requireTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;

    $requireStart = $requireTokSub;
    $requireBodyStart = $requireStart+1;
    $requireBodyEnd = identifyStatementEnd($requireBodyStart);
    //echo "requireBodyStart=".$requireBodyStart." requireBodyEnd=".$requireBodyEnd."<br
$upmlText[$countSection] .= "<require>
";

//identifyBody($requireBodyStart, $requireBodyEnd)
identifyExpression($requireBodyStart, $requireBodyEnd-1);

$upmlText[$countSection] .= "</require>
";

$requireEnd = $requireBodyEnd;
return $requireEnd;

//identify require_once statement
function identifyRequireonceStatement($requireonceTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
           $currentClassName, $countSection;

    $requireonceStart = $requireonceTokSub;
    $requireonceBodyStart = $requireonceStart+1;
    $requireonceBodyEnd = identifyStatementEnd($requireonceBodyStart);
    $upmlText[$countSection] .= "<requireOnce>
";

    //****identifyBody($requireBodyStart, $requireBodyEnd)
    identifyExpression($requireonceBodyStart, $requireonceBodyEnd-1);

    $upmlText[$countSection] .= "</requireOnce>
";

    $requireonceEnd = $requireonceBodyEnd;
    return $requireonceEnd;

}

//identify include statement
function identifyIncludeStatement($includeTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
           $currentClassName, $countSection;

    $includeStart = $includeTokSub;
    $includeBodyStart = $includeStart+1;
    $includeBodyEnd = identifyStatementEnd($includeBodyStart);
    $upmlText[$countSection] .= "<include>
";

    //****identifyBody($requireBodyStart, $requireBodyEnd)
    identifyExpression($includeBodyStart, $includeBodyEnd-1);

    $upmlText[$countSection] .= "</include>
";

    $includeEnd = $includeBodyEnd;
    return $includeEnd;

}
$currentClassName, $countSection;

    $includeStart = $includeTokSub;
    $includeBodyStart = $includeStart+1;
    $includeBodyEnd = identifyStatementEnd($includeBodyStart);
    $upmlText[$countSection] .= "<include>
"

    //****identifyBody($includeBodyStart, $includeBodyEnd)
    identifyExpression($includeBodyStart, $includeBodyEnd-1);

    $upmlText[$countSection] .= "</include>
"

    $includeEnd = $includeBodyEnd;
    return $includeEnd;

}

//identify include_once statement
function identifyIncludeOnceStatement($includeonceTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;

    $includeonceStart = $includeonceTokSub;
    $includeonceBodyStart = $includeonceStart+1;
    $includeonceBodyEnd = identifyStatementEnd($includeonceBodyStart);
    $upmlText[$countSection] .= "<includeOnce>
"

    //****identifyBody($includeonceBodyStart, $includeonceBodyEnd)
    identifyExpression($includeonceBodyStart, $includeonceBodyEnd-1);

    $upmlText[$countSection] .= "</includeOnce>
"

    $includeonceEnd = $includeonceBodyEnd;
    return $includeonceEnd;

}
//identify goto statement
function identifyGotoStatement($gotoTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
          $currentClassName, $countSection;

    $gotoStart = $gotoTokSub;
    $gotoBodyStart = $gotoStart+1;
    $gotoBodyEnd = identifyStatementEnd($gotoBodyStart);
    //echo "gotoStart=".$gotoStart." gotoBodyStart=".$gotoBodyStart." gotoBodyEnd=".$gotoBodyEnd."<br />"
    $upmlText[$countSection] .= "<goto>
    identifyExpression($gotoBodyStart, $gotoBodyEnd-1);

    $upmlText[$countSection] .= "</goto>"

    $gotoEnd = $gotoBodyEnd;
    return $gotoEnd;
}

//identify an expression start with a variable
function identifyVariableStartStatement($variableTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
          $currentClassName, $countSection;
    global $braceArray;
    $variableStart = $variableTokSub;
    $variableEnd = identifyStatementEnd($variableStart);  //variableEnd is the position of the semicolon
    $upmlText[$countSection] .= "<statement>
    identifyExpression($variableStart, $variableEnd-1);  //the item at $variableEnd is semicolon
    $upmlText[$countSection] .= "</statement>";
return $variableEnd;
}

//identify a statement starts with global
function identifyGlobalStartStatement($globalTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;

    //identify the end of a global statement
    $subEnd = identifyStatementEnd($globalTokSub);

    $exStart = $globalTokSub + 1; //keyword global is not a part of the expression in the
statement
    $exEnd = $subEnd - 1;

    $upmlText[$countSection].= "<global>\n";
    identifyExpression($exStart, $exEnd);
    $upmlText[$countSection].= "</global>\n";
    return $subEnd;
}

//identify a statement starts with static
function identifyStaticStartStatement($staticTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;

    //identify the end of a static statement
    $subEnd = identifyStatementEnd($staticTokSub);

    $exStart = $staticTokSub + 1;
    $exEnd = $subEnd - 1;
function identifyStatic($exStart, $exEnd)
{
    $upmlText[$countSection] = "<static>
    identifyExpression($exStart, $exEnd);
    $upmlText[$countSection] = "</static>
    
    return $subEnd;
}

function identifyEchoStatement($echoTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;
    
    //identify the end of a static statement
    $subEnd = identifyStatementEnd($echoTokSub);
    
    $exStart = $echoTokSub + 1;
    $exEnd = $subEnd - 1;
    //echo "in identifyEchoStatement, exStart=".$exStart." exEnd=".$exEnd."<br />
    $upmlText[$countSection] = "<output device="monitor">
    identifyExpression($exStart, $exEnd);
    $upmlText[$countSection] = "</output>
    
    return $subEnd;
} //end of identifyEchoStatement

//identify a statement starts with echo
function identifyPrintStatement($printTokSub)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;
    
    //identify the end of a print statement
    $subEnd = identifyStatementEnd($printTokSub);
$exStart = $printTokSub + 1;
$exEnd = $subEnd - 1;

$upmlText[$countSection] = "<output device="printer">\n";
identifyExpression($exStart, $exEnd);
$upmlText[$countSection] = "</output>\n";

return $subEnd;

//**!!identify an Expression
function identifyExpression($exStart, $exEnd)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray, $bracketArray,
    $dBraceArray, $dBracketArray;
    /**1. looking for "(" and ")" pairs and "[" and "]" pairs*/
    unset($dBraceArray); //clear $dBraceArray
    unset($dBracketArray); //clear $dBracketArray

    /***1.1 detect if the expression is inside a pair of braces, brackets
     * if so, use the content without the braces/brackets to transform
     */
    if(($exStart == $exEnd -1) && $tokens[$exStart]=="(" && $tokens[$exEnd]==")") //function argument list is empty
        return $exEnd + 1;

    //for case like $i++, $i--, ++$i, --$i, etc.
    if(($exStart == $exEnd -1) && $tokens[$exStart][0]==T_INC)
    {
        $upmlText[$countSection] .= "<expression>\n<pre_increment>\n";
        //identifyPrimitive($exEnd, $exEnd);
        $upmlText[$countSection] .= "<variable>\n";
    
    
}
$upmlText[$countSection] .= $tokens[$exEnd][1]."\n";
$upmlText[$countSection] .= "</variable>\n";
$upmlText[$countSection] .= "</pre_increment>\n</expression>\n";
return $exEnd + 1;
}
else if (($exStart == $exEnd -1) && $tokens[$exStart][0]==T_DEC)
{
$upmlText[$countSection] .= "<expression>\n<pre_decrement>\n";
//identifyPrimitive($exEnd, $exEnd);
$upmlText[$countSection] .= "</variable>\n";
$upmlText[$countSection] .= $tokens[$exEnd][1]."\n";
$upmlText[$countSection] .= "</variable>\n";
$upmlText[$countSection] .= "</pre_decrement>\n</expression>\n";
return $exEnd + 1;
}
else if (($exStart == $exEnd -1) && $tokens[$exEnd][0]==T_INC)
{
$upmlText[$countSection] .= "<expression>\n<post_increment>\n";
//identifyPrimitive($exStart, $exEnd);
$upmlText[$countSection] .= "</variable>\n";
$upmlText[$countSection] .= $tokens[$exStart][1]."\n";
$upmlText[$countSection] .= "</variable>\n";
$upmlText[$countSection] .= "</post_increment>\n</expression>\n";
return $exEnd + 1;
}
else if (($exStart == $exEnd -1) && $tokens[$exEnd][0]==T_DEC)
{
$upmlText[$countSection] .= "<expression>\n<post_decrement>\n";
//identifyPrimitive($exStart, $exEnd);
$upmlText[$countSection] .= "</variable>\n";
$upmlText[$countSection] .= $tokens[$exStart][1]."\n";
$upmlText[$countSection] .= "</variable>\n";
$upmlText[$countSection] .= "</post_decrement>\n</expression>\n";
return $exEnd + 1;
}

$expr = removeBraceBracket($exStart, $exEnd);  // $expr holds the new $exStart and $exEnd
//after remove leftmost "(" and rightmost ")", the start and end of the expression become:
$xStart = $expr[0];
$xEnd = $expr[1];

if (($xStart == $xEnd - 1) && $tokens[$xStart][0]=="-") // negative item like -4.
{
    $upmlText[$countSection] .= "<expression>\n<negative>\n";
    identifyPrimitive($xEnd, $xEnd);
    $upmlText[$countSection] .= "</negative>\n</expression>\n";
    return $exEnd + 1;
}
else if (($xStart == $xEnd - 1) && $tokens[$xStart][0]=="+") // positive item like +4.
{
    $upmlText[$countSection] .= "<expression>\n<positive>\n";
    identifyPrimitive($xEnd, $xEnd);
    $upmlText[$countSection] .= "</positive>\n</expression>\n";
    return $exEnd + 1;
}

$dBraceArray=findDynamicBracePair($xStart, $xEnd);
$dBracketArray=findDynamicBracketPair($xStart, $xEnd);

$commaExpressionArray = array(); // an array hold expressions (start and end location)

/** 2. divide an expression into groups according to operator precedence*/
/** 2.1. separate by commas*/
if ($xStart == $xEnd) // this could be a single item
{
    identifyPrimitive($xStart, $xEnd);
    return $exEnd + 1;
}
else if ($xStart == ($xEnd - 1)) // this could be inc, dec, &$x
{
    if ($tokens[$xStart][0]=="&")
        
        Page 161
{  
    identifyPrimitive($xStart, $xEnd);  
    return $exEnd + 1;  
}
$exArray = array();
$exArray[0]= $xStart;
$exArray[1]= $xEnd;
$idcExpressionArray = findIncDecCast($xArray);
//echo "in identifyExpression, idcExpressionArray:\n";
//print_r($idcExpressionArray);
transformIncDecCast($idcExpressionArray);
return $exEnd + 1;
}
else $commaExpressionArray = findComma($xStart, $xEnd);
//echo "expressions separated by commas: \n";
//print_r($commaExpressionArray);
//echo "<br />
/**2.2. transform commas and then separate ORs*/
if(sizeof($commaExpressionArray)>1)//if there are more than 1 item, which means there
is at least one comma between
    $upmlText[$countSection] .= "<list dilimiter="comma">\n";
transformComma($commaExpressionArray);//transform commas then separate ORs
if(sizeof($commaExpressionArray)>1)//if there are more than 1 item, which means there
is at least one comma between
    $upmlText[$countSection] .= "</list>\n";
return $exEnd+1;
}

function removeBraceBracket($st, $nd)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClassName,
            $currentClassName, $countSection;
    global $braceArray, $bracketArray, $dBraceArray, $dBracketArray;
    $tempVar = $st;
    Page 162
countBracePair = 0; //count left and right braces
$countBracketPair = 0; //count left and right brackets
$ex = array(); //hold expression

if($tokens[$st]=="(" && $tokens[$nd]==")")
{
    while($tempVar <= $nd)
    {
        if($tokens[$tempVar]==")")
            $countBracePair++;
        else if($tokens[$tempVar]==")")
            $countBracketPair--;
        if($countBracketPair==0)
            break; //the leftmost "(" found paired "")"
        $tempVar++; 
    }
}
if($tempVar==$nd && $st<=$nd-2) //after found paired "(" and ")", if the leftmost pairs with the right most, remove the outmost "(" and ")"
{
    $ex[0]=$st + 1;
    $ex[1]=$nd - 1;
    return $ex;
}

$tempVar = $st; //initialize the temporary variable for detecting brackets

if($tokens[$st]=="[" && $tokens[$nd]=="]")
{
    while($tempVar <= $nd)
    {
        if($tokens[$tempVar]=="]")
            $countBracketPair++;
        else if($tokens[$tempVar]=="]")
            $countBracketPair--;
        if($countBracketPair==0)break; //the leftmost "(" found paired "")"
        $tempVar++; 
    }
}
if ($tempVar==$nd && $st<=$nd-2) //after found paired "(" and ")", if the leftmost pairs with the right most, remove the outmost "(" and ")"
{
    $ex[0]=$st + 1;
    $ex[1]=$nd - 1;
    return $ex;
}

$ex[0]=$st;
$ex[1]=$nd;
return $ex;

} //end of removeBraceBracket()

function findComma($commaStart, $commaEnd)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray, $bracketArray; //, $dBraceArray, $dBracketArray;
    $tempVar = $commaStart;
    $lc = $tempVar; //left side of a comma
    $rc = $tempVar; //right side of a comma
    $commaNum = -1; //count number of comma, this number is used as array subscription
    $exStart = $commaStart; //record the start of an expression
    $exEnd = $commaEnd; //record the end of an expression
    $exArray = array(); //record an expression separated by a comma
    $commaExArray = array(); //record all the expressions separated by commas
    //echo "in findComma, commaEnd=".$commaEnd."<br />";
    $dBraceArray=findDynamicBracePair($exStart, $exEnd);

    while ($tempVar <= $commaEnd)
    {
        //echo "enter while in findcomma.tempVar=".$tempVar."<br />";
        if ($tokens[$tempVar] == ",") //find a comma
        {
            //echo "enter comma <br />";
        }
if(inBraces($tempVar,$commaStart, $commaEnd, $dBraceArray)); //if the comma is between a pair of "(" and ")", do nothing but to find the next comma
else if(inBrackets($tempVar,$commaStart, $commaEnd, $dBracketArray)); //if
the comma is between a pair of "[" and "]", do nothing but to find the next comma
else
{
    $commaNum++;
    //echo "commaNum=".$commaNum."<br />";
    //if($commaNum==1)$upmlText[$countSection] .= "<comma>
"; //when a comma is found for the first time, start a comma list
    $lc = $tempVar - 1; //left side of the comma
    $rc = $tempVar + 1; //right side of the comma
    //exStart = $commaStart; //start of the left expression separated by the comma
    $exEnd = $lc; //end of the left expression separated by the comma
    $exArray[0]=$exStart;
    $exArray[1]=$exEnd;
    //echo "exArray is:";
    //print_r($exArray);
    //echo "<br/>
";
    array_push($commaExArray, $exArray); //add the array in the expression array list
    $exStart = $rc; //next search start is the right side of the comma
    //findComma($rc, $commaEnd, $braceArray); //repeat finding comma in the right expression
} //end else
 //echo "start=".$exStart."<br />";
} //end if
//prepare next turn of looking for comma
//if($rc > $tempVar)$tempVar=$rc;
//else $tempVar++;
$tempVar++;
//echo "exit while in findcomma. tempVar=".$tempVar."<br />";
} //end of while looking for commas
$exArray[0]=$rc;
$exArray[1]=$commaEnd;
array_push($commaExArray, $exArray); //the last expression, or the entire expression is
//echo "<br />in findComma, expressions separated by commas:"."<br />
return $commaExArray; //return the found array of strings (marked with start and end location) separated by commas
}//end of function findComma

//transform expressions separated by commas
function transformComma($commaExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
    global $braceArray, $bracketArray, $dBraceArray, $dBracketArray;
    
    $orExpressionArray = array(); //an array that hold items (start and end) separated by ORs
    //echo "enter transformComma.<br />
    for($i=0; $i<sizeof($commaExArray); $i++)
    {
        $orExpressionArray = findOr($commaExArray[$i]); //identify ORs
        //echo "expressions separated by or: <br />
        //print_r($orExpressionArray);
        //echo "<br />
        if(sizeof($orExpressionArray)>1)//if there are more than 1 item, which means there is at least one comma between
            $upmlText[$countSection] .= "<logical_or>
            transformOr($orExpressionArray);//transform commas then separate ORs
            if(sizeof($orExpressionArray)>1)//if there are more than 1 item, which means there is at least one comma between
                $upmlText[$countSection] .= "</logical_or>
        }
    }
    //end for loop
}//end of transformComma

//find expressions separate by ors
function findOr($orExpressionArray)//expression with possible ORs. the OR array consists of


```php
start and end location
{

global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
$currentClassName, $countSection;
global $braceArray, $bracketArray;//, $dBraceArray, $dBracketArray;
$tempVar = $orExpressionArray[0];

$lo = $tempVar; //left side of an OR
$ro = $tempVar; //right side of an OR
$orNum = -1; //count number of or, this number is used as array subscription
$exStart = $orExpressionArray[0]; //record the start of an expression
$exEnd = $orExpressionArray[1]; //record the end of an expression
$orStart = $exStart;//start of an expression separated by OR
$orEnd = $exStart; //end of an expression separated by OR
$exArray = array(); //record an expression separated by a OR
$orExArray = array(); //record all the expressions separated by OR
$dBraceArray = findDynamicBracePair($exStart, $exEnd);

while($tempVar <= $exEnd)
{
    if($tokens[$tempVar][0] == T_LOGICAL_OR ) //find a logical OR
    {
        if(inBraces($tempVar, $exStart, $exEnd, $dBraceArray)); //if the OR is between a pair of "(" and ")", do nothing but to find the next OR
        else if(inBrackets($tempVar, $exStart, $exEnd, $dBracketArray)); //if the OR is between a pair of "[" and "]", do nothing but to find the next OR
        else
        {
            $orNum++;
            $lo = $tempVar - 1; //left side of the OR
            $ro = $tempVar + 1; //right side of the OR
            $orEnd = $lo; //end of the left expression separated by the OR
            $exArray[0]=$orStart;
            $exArray[1]=$orEnd;
            array_push($orExArray, $exArray);//add the array in the expression
            array list
            $orStart = $ro; //next search start is the right side of the OR
```

```
Page 167
```
//end of while looking for ORs
$exArray[0]=$ro;
$exArray[1]=$exEnd;
array_push($orExArray, $exArray); //the last expression, or the entire expression is no OR

//echo "<br />in findComma, expressions separated by ORs:".print_r($orExArray)."<br />
return $orExArray; //return the found array of strings (marked with start and end location) separated by ORs
}//end of findOr

//transform expressions separated by ORs
function transformOr($orExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
        $currentClassName, $countSection;
    global $braceArray, $bracketArray, $dBraceArray, $dBracketArray;

    $xorExpressionArray = array(); //an array that hold items (start and end) separated by ORs
    for($i=0; $i<sizeof($orExArray); $i++)
    {
        $xorExpressionArray = findXor($orExArray[$i]); //identify XORs
        if(sizeof($xorExpressionArray)>1) //if there are more than 1 item, which means there is at least one XOR between
            $upmlText[$countSection] .= "<logical_xor>\n";
        transformXor($xorExpressionArray); //transform commas then separate XORs
        if(sizeof($xorExpressionArray)>1) //if there are more than 1 item, which means there is at least one XOR between
            $upmlText[$countSection] .= "</logical_xor>\n";
    } //end for loop
}//end of transformOr

//find expressions separated by XORs
function findXor($xorExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
    global $braceArray, $bracketArray; //, $dBraceArray, $dBracketArray;
    $tempVar = $xorExArray[0];

    $lx = $tempVar; // left side of an XOR
    $rx = $tempVar; // right side of an XOR
    $xorNum = -1; // count number of XOR, this number is used as array subscription
    $exStart = $xorExArray[0]; // record the start of an expression
    $exEnd = $xorExArray[1]; // record the end of an expression
    $xorStart = $exStart; // start of an expression separated by XOR
    $xorEnd = $exStart; // end of an expression separated by XOR
    $exArray = array(); // record an expression separated by a XOR
    $xorExArray = array(); // record all the expressions separated by XOR
    $dBraceArray = findDynamicBracePair($exStart, $exEnd);

    while($tempVar <= $exEnd)
    {
        if($tokens[$tempVar][0] == T_LOGICAL_XOR) // find a logical XOR
        {
            // echo "!!!!!$tempVar<br/>";
            if(inBraces($tempVar, $exStart, $exEnd, $dBraceArray)); // if the XOR is between a pair of "(" and ")", do nothing but to find the next XOR
            else if(inBrackets($tempVar, $exStart, $exEnd, $dBracketArray)); // if the XOR is between a pair of "[" and "]", do nothing but to find the next XOR
            else
            {
                $xorNum++;
                $lx = $tempVar - 1; // left side of the XOR
                $rx = $tempVar + 1; // right side of the XOR
                $xorEnd = $lx; // end of the left expression separated by the comma
                $exArray[0] = $xorStart;
                $exArray[1] = $xorEnd;
                array_push($xorExArray, $exArray); // add the array in the expression
            }
        } // end else
    } // end while
} // end else

Page 169
//end if
$tempVar++;
}//end of while looking for logical_xor
$exArray[0]=$rx;
$exArray[1]=$exEnd;
array_push($xorExArray, $exArray); //the last expression, or the entire expression is no XOR
return $xorExArray; //return the found array of strings (marked with start and end location) separated by XORs
}//end of findXor

//transform XOR and find AND
function transformXor($xorExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
    global $braceArray;
    $andExpressionArray = array(); //an array that hold items (start and end) separated by ANDs

    for($i=0; $i<sizeof($xorExArray); $i++)
    {
        $andExpressionArray = findAnd($xorExArray[$i]); //identify ANDs
        //echo "expressions separated by ANDs: <br />";
        //print_r($andExpressionArray);
        //echo "<br />";
        if(sizeof($andExpressionArray)>1)//if there are more than 1 item, which means there is at least one AND between
            $upmlText[$countSection] .= "<logical_and>
";
        transformAnd($andExpressionArray); //transform commas then separate ANDs
        if(sizeof($andExpressionArray)>1)//if there are more than 1 item, which means there is at least one AND between
            $upmlText[$countSection] .= "</logical_and>
";
    }//end for loop
}//end of transformXor
function findAnd($andExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray, $bracketArray; //, $dBraceArray, $dBracketArray;
    $tempVar = $andExArray[0];

    $la = $tempVar; // left side of an AND
    $ra = $tempVar; // right side of an AND
    $andNum = -1; // count number of AND, this number is used as array subscription
    $exStart = $andExArray[0]; // record the start of an expression
    $exEnd = $andExArray[1]; // record the end of an expression
    $andStart = $exStart; // start of an expression separated by AND
    $andEnd = $exStart; // end of an expression separated by AND
    $exArray = array(); // record an expression separated by a AND
    $andExArray = array(); // record all the expressions separated by AND
    $dBraceArray = findDynamicBracePair($exStart, $exEnd);
    while ($tempVar <= $exEnd)
    {
        if ($tokens[$tempVar][0] == T_LOGICAL_AND) // find a logical AND
        {
            if (inBraces($tempVar, $exStart, $exEnd, $dBraceArray)); // if the AND is
            between a pair of "(" and ")", do nothing but to find the next AND
            else if (inBrackets($tempVar, $exStart, $exEnd, $dBracketArray)); // if the AND
            is between a pair of "[" and "]", do nothing but to find the next AND
            else
            {
                $andNum++;
                $la = $tempVar - 1; // left side of the AND
                $ra = $tempVar + 1; // right side of the AND
                $andEnd = $la; // end of the left expression separated by the AND
                $exArray[0] = $andStart;
                $exArray[1] = $andEnd;
                array_push($andExArray, $exArray); // add the array in the expression
            }
        }
    }
}
$andStart = $ra; // next search start is the right side of the AND
 }// end else
 $tempVar++;
 }// end of while looking for logical_AND
 $exArray[0] =$ra;
 $exArray[1] =$exEnd;
 array_push($andExArray, $exArray); // the last expression, or the entire expression is no AND
 return $andExArray; // return the found array of strings (marked with start and end location) separated by AND
 }// end of findAnd

// transform AND and find Assignment
function transformAnd($andExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;

    $assignExpressionArray = array(); // an array that hold items (start and end) separated by ANDs

    // echo "enter transformAnd.<br /"; // transform ANDs and find Assignment
    for($i=0; $i<sizeof($andExArray); $i++)
    {
        $assignExpressionArray = findAssign($andExArray[$i]); // identify Assignments
        // echo "expressions separated by Assignment: <br /">;
        // print_r($assignExpressionArray);
        // echo "<br />";
        transformAssign($assignExpressionArray); // transform commas then separate
    }// end of for loop
}

// find expressions separated by assignments
function findAssign($assignExArray)
{
$countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
$currentClassName, $countSection;
$braceArray, $bracketArray;
$tempVar = $assignExArray[0];

$lass = $tempVar; //left side of an Assignment
$rass = $tempVar; //right side of an Assignment
$assNum = -1; //count number of AND, this number is used as array subscription
$exStart = $assignExArray[0]; //record the start of an expression
$exEnd = $assignExArray[1]; //record the end of an expression
$assStart = $exStart; //start of an expression separated by Assignment
$assEnd = $exStart; //end of an expression separated by Assignment
$exArray = array(); //record an expression separated by a Assignment
$markAssign = ""; //mark the last assignment operator for the last item
$assExArray = array(); //record all the expressions separated by Assignment
$dBraceArray = findDynamicBracePair($exStart, $exEnd);
while($tempVar <= $exEnd)
{
    if(($tokens[$tempVar][0] == "=") || //find a assignment sign
        ($tokens[$tempVar][0] == T_PLUS_EQUAL) ||
        ($tokens[$tempVar][0] == T_MINUS_EQUAL) ||
        ($tokens[$tempVar][0] == T_MUL_EQUAL) ||
        ($tokens[$tempVar][0] == T_DIV_EQUAL) ||
        ($tokens[$tempVar][0] == T_CONCAT_EQUAL) ||
        ($tokens[$tempVar][0] == T_MOD_EQUAL) ||
        ($tokens[$tempVar][0] == T_AND_EQUAL) ||
        ($tokens[$tempVar][0] == T_OR_EQUAL) ||
        ($tokens[$tempVar][0] == T_XOR_EQUAL) ||
        ($tokens[$tempVar][0] == T_SL_EQUAL) ||
        ($tokens[$tempVar][0] == T_SR_EQUAL) ||
        ($tokens[$tempVar][0] == T_DOUBLE_ARROW))
    {
        if(inBra($tempVar, $exStart, $exEnd, $dBraceArray)); //if the = is between
            a pair of "(" and ")", do nothing but to find the next ";"
        else if(inBrackets($tempVar, $exStart, $exEnd, $dBracketArray)); //if the = is
            between a pair of "[" and "]", do nothing but to find the next ";"
        else
            //do the things
    }
}
{  
    $assNum++;
    $lass = $tempVar - 1;  //left side of the assignment
    $rass = $tempVar + 1;  //right side of the assignment
    $assEnd = $lass;  //end of the left expression separated by the assignment

    $exArray[0]=$assStart;
    $exArray[1]=$assEnd;
    if ($tokens[$tempVar][0] == "=") $exArray[2]="assign";
    if ($tokens[$tempVar][0] == T_PLUS_EQUAL) $exArray[2]="plus_assign";
    if ($tokens[$tempVar][0] == T_MINUS_EQUAL) $exArray[2]="minus_assign";
    if ($tokens[$tempVar][0] == T_MUL_EQUAL) $exArray[2]="mul_assign";
    if ($tokens[$tempVar][0] == T_DIV_EQUAL) $exArray[2]="div_assign";
    if ($tokens[$tempVar][0] == T_CONCAT_EQUAL) $exArray[2]="concat_assign";
    if ($tokens[$tempVar][0] == T_MOD_EQUAL) $exArray[2]="mod_assign";
    if ($tokens[$tempVar][0] == T_AND_EQUAL) $exArray[2]="and_assign";
    if ($tokens[$tempVar][0] == T_OR_EQUAL) $exArray[2]="or_assign";
    if ($tokens[$tempVar][0] == T_XOR_EQUAL) $exArray[2]="xor_assign";
    if ($tokens[$tempVar][0] == T_SL_EQUAL) $exArray[2]="shift_left_assign";
    if ($tokens[$tempVar][0] == T_SR_EQUAL) $exArray[2]="shift_right_assign";
    array_push($assExArray, $exArray);  //add the array in the expression array list

    $markAssign = $exArray[2];  //mark the assignment operator for the last item
    $assStart = $rass;  //next search start is the right side of the assignment

}  //end if else
$tempVar++;
}  //end of while looking for assignment

$exArray[0]=$rass;
$exArray[1]=$exEnd;
$exArray[2]=$markAssign;
array_push($assExArray, $exArray);  //the last expression, or the entire expression has no assignment

return $assExArray;  //return the found array of strings (marked with start and end
location) separated by assignment
}//end of findAssign

//transform Assignment and find Ternary
function transformAssign($assExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;

    $ternaryExpressionArray = array(); //an array that hold items (start and end) separated
    by Ternary

    $markAssOp = array(); //record location where appear a different operator
    $markNum = 0; //record number of different operators
    $markAssOp[0] = 0; //the start position, for the first op

    $sizeAssExArray = sizeof($assExArray);
    //echo "sizeAssExArray=","<br /">";
    if ($sizeAssExArray > 1)
    {
        //search the positions that appear a different operator
        for ($j = 1; $j < $sizeAssExArray; $j++)
        {
            if ($assExArray[$j][2] <> $assExArray[$j - 1][2])
            {
                $markNum++;
                $markAssOp[$markNum] = $j;
            }
        }
        //echo "markNum=$markNum<br/>";
        //echo "markAssOp:<br />";
        //print_r($markAssOp);
        //echo "<br /">";
    }

    for ($i = 0; $i < $sizeAssExArray; $i++)
    {

ternaryExpressionArray = findTernary($assExArray[$i]); // left item of an assignment op; identify Ternary

if($sizeAssExArray<=1)
    transformTernary($ternaryExpressionArray); // transform ternary then

boolean or

else // more than one item
{
    // echo "enter transformAssign else <br />";
    if(in_array($i, $markAssOp)) // recorded op that be different from the previous op
        $upmlText[$countSection] .= "<".$assExArray[$i][2]."<\n"; // start a new op

    transformTernary($ternaryExpressionArray); // transform ternary then

    boolean or

    if($i==($sizeAssExArray-1)) // the last item
        for($k=$markNum; $k>=0; $k--)
        { // end each of the marked operators
            $upmlText[$countSection] .= "</".$assExArray[$markAssOp[$k]][2]."</\n";
        }

    } // end for loop

// end of transformAssign

function findTernary($ternExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray, $bracketArray; //, $dBraceArray, $dBracketArray;

    $tempVar = $ternExArray[0];
$lt = $ternExArray[0]; //start of a Ternary
$rt = $ternExArray[1]; //end of a Ternary
$condStart = $lt; //condition start of a Ternary
$condEnd = $rt; //condition end of a Ternary
$trueStart = $lt; //true value start of a Ternary
$trueEnd = $rt; //true value end of a Ternary
$falseStart = $lt; //false value start of a Ternary
$falseEnd = $rt; //false value end of a Ternary
$ternNum = -1; //count number of Ternary, this number is used as array subscription
$exStart = $ternExArray[0]; //record the start of an expression
$exEnd = $ternExArray[1]; //record the end of an expression
$ternStart = $exStart; //start of an expression separated by Ternary
$ternEnd = $exStart; //end of an expression separated by Ternary
$ternExpressionArray = array(); //record all the expressions separated by Ternary
$colonFound = false; //mark if a ternary ?: is found

//initialize the ternary
$ternExpressionArray[0][0] = $condStart; //condition start
$ternExpressionArray[0][1] = $condEnd; //condition end
$ternExpressionArray[1][0] = $trueStart; //true value start
$ternExpressionArray[1][1] = $trueEnd; //true value end
$ternExpressionArray[2][0] = $falseStart; //false value start
$ternExpressionArray[2][1] = $falseEnd; //false value end

$borExpressionArray = array(); //an array that hold items (start and end) separated by logical OR(||)
$dBraceArray = findDynamicBracePair($exStart, $exEnd);

while ($tempVar <= $exEnd)
{
    if ($tokens[$tempVar][0] == "?") //find "?" of a Ternary
    {
        $condEnd = $tempVar - 1; //the location of the end of condition
        $trueStart = $tempVar + 1; //the location of the start of true value
        //search ";"
        $tempVar1 = $tempVar + 1; //a temporary variable used in an inner while loop
    
    loop
while($tempVar1 < $exEnd)
{
    if((tokens[$tempVar1][0]==":")) //find ":" of the Ternary
        && (!inBraces($tempVar, $exStart, $exEnd, $dBraceArray))//?
    is not inside a pair of braces
        && (!inBraces($tempVar1, $exStart, $exEnd, $dBraceArray))//: is not inside a pair of braces
        && (!braceBetween($tempVar, $tempVar1))//the ? and : are separated by "(" or ")"
    && (!inBrackets($tempVar, $exStart, $exEnd, $dBracketArray))//? is not inside a pair of brackets
    && (!inBrackets($tempVar1, $exStart, $exEnd, $dBracketArray))// : is not inside a pair of brackets
        && (!bracketBetween($tempVar, $tempVar1))//the ? and : are separated by "[" or "]"
    {
        $trueEnd = $tempVar1 - 1; //the location of the end of true value
        $falseStart = $tempVar1 + 1; //the location of the start of false value
        $condStart; //$exStart;//condition start
        $ternExpressionArray[0][0] = $condStart; //$exStart;//condition end
        $ternExpressionArray[1][0] = $trueStart; //true value start
        $ternExpressionArray[2][0] = $falseStart; //false value start
        $ternExpressionArray[2][1] = $falseEnd; //false value end
        $colonFound=true;
    }//end of construct Ternary
}
//true value end
if($colonFound==true)break;
} //end while looking for :
} //end if find "?"
$tempVar++;
if($colonFound==true) break;
} //end of while looking for Ternary
return $ternExpressionArray; //return the found array of strings (marked with start and end location) separated by Ternary
} //end of findTernary

//search if "(" or ")" is between two position
function braceBetween($temp1, $temp2)
{

    global $tokens;
    $nLeftBrace = 0; //number of left braces appeared
    $nRightBrace = 0; //number of right braces appeared
    $locLeftBrace = 0; //accumulated position value of left braces
    $locRightBrace = 0; //accumulated position value of right braces

    for($i=$temp1; $i<=$temp2; $i++)
    {
        //echo "tokens[".$i."]=".$tokens[$i][0]."<br />");
        if($tokens[$i][0]=="(")
        {
            $nLeftBrace++;
            $locLeftBrace += $i;
        }
        else if($tokens[$i][0]==")")
        {
            $nRightBrace++;
            $locRightBrace += $i;
        }

        if($nRightBrace < $nLeftBrace) return true; //at any time, if more right brace, true
    }

    if($nLeftBrace<>$nRightBrace) return true; //if finally number of "(" and that of ")" not equal, true
return false;
}

function bracketBetween($temp1, $temp2) {
    global $tokens;
    $nLeftBracket = 0;  // number of left brackets appeared
    $nRightBracket = 0; // number of right brackets appeared
    $locLeftBracket = 0; // accumulated position value of left brackets
    $locRightBracket = 0; // accumulated position value of right brackets

    for($i=$temp1; $i<=$temp2; $i++) {
        // echo "tokens[".$i."]=".$tokens[$i][0]."<br />
        if($tokens[$i][0]=="[") {
            $nLeftBracket++;
            $locLeftBracket += $i;
        } else if($tokens[$i][0]=="]") {
            $nRightBracket++;
            $locRightBracket += $i;
        }

        if($nRightBracket < $nLeftBracket) return true; // at any time, if more right bracket, true
    }
    if($nLeftBracket<$nRightBracket) return true; // if finally number of "[" and that of "]" not equal, true
    return false;
}

// search if more items are before the leftmost "(" function moreBeforeBraces($tempVar, $exStart) {
}
global $tokens;
$i = $tempVar;
$nBraces = 0; //number of "(" and ")"
$locLeft = 0; //location of the leftmost "(" 
while($i<=$exStart)
{
    if($tokens[$i]=="")
    {
        $nBraces++;
        $locLeft = $i;
    }
    else if($tokens[$i]==")")
    {
        $nBraces--;
        $i--;
    }
    if($nBraces <> 0)return true; //number of "(" and that of ")" not match
else if($locLeft > $exStart)return true; //more items before the leftmost "(" 
else return false;
}

//search if more items are behind the rightmost ")"
function moreBehindBraces($tempVar, $exEnd)
{
    global $tokens;
    $i = $tempVar;
    $nBraces = 0; //number of "(" and ")"
    $locRight = 0; //location of the rightmost ")"
    while($i <= $exEnd)
    {
        if($tokens[$i]=="")
        {
            $nBraces++;
        }
        else if($tokens[$i]==")")
        {
            $nBraces--;
            $locRight = $i;
        }
$i++;
}
if ($nBraces <> 0) return true; // number of "(" and that of ")" not match
else if ($locRight < $exEnd) return true; // more items before the leftmost "(
else return false;
}

// transform Ternary and find boolean OR( || )
function transformTernary($ternExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;

    $borExpressionArray = array(); // an array that hold items (start and end) separated by logical OR( || )

    if ($ternExArray[0][0]==$ternExArray[1][0])
    { // start of condition and start of true value are the same, no real ternary
        $borExpressionArray = findBOR($ternExArray[0][0], $ternExArray[2][1]); // not a real ternary

        // echo "expressions separated by boolean OR: <br />";
        // print_r($borExpressionArray);
        // echo "<br />
        if (sizeof($borExpressionArray)>1) // if there are more than 1 item, which means there is at least one XOR between
            $upmlText[$countSection] .= "<".boolean_or.">\n"; // start a new op
        transformBOR($borExpressionArray); // transform boolean OR( || ) found in condition of ternary
    }
    else
    {
        $upmlText[$countSection] .= "</".boolean_or.">\n"; // start a new op
    
}
$borExpressionArray = findBOR($ternExArray[0][0], $ternExArray[0][1]); // identify logical OR(||) from the expression of condition in the ternary

if (sizeof($borExpressionArray) > 1) // if there are more than 1 item, which means there is at least one XOR between
    $upmlText[$countSection] .= "<boolean_or><\n";
    transformBOR($borExpressionArray); // transform boolean OR(||) found in condition of ternary

if (sizeof($borExpressionArray) > 1) // if there are more than 1 item, which means there is at least one XOR between
    $upmlText[$countSection] .= "</boolean_or></\n";

$upmlText[$countSection] .= "</condition><\n";

$borExpressionArray = findBOR($ternExArray[1][0], $ternExArray[1][1]); // identify boolean OR(||) from the expression of condition in the ternary

if (sizeof($borExpressionArray) > 1) // if there are more than 1 item, which means there is at least one XOR between
    $upmlText[$countSection] .= "<boolean_or><\n";
    transformBOR($borExpressionArray); // transform boolean OR(||) found in condition of ternary

if (sizeof($borExpressionArray) > 1) // if there are more than 1 item, which means there is at least one XOR between
    $upmlText[$countSection] .= "</boolean_or></\n";

$upmlText[$countSection] .= "</value_of_true_condition><\n";

$borExpressionArray = findBOR($ternExArray[2][0], $ternExArray[2][1]); // identify boolean OR(||) from the expression of condition in the ternary

if (sizeof($borExpressionArray) > 1) // if there are more than 1 item, which means there is at least one XOR between
    $upmlText[$countSection] .= "<boolean_or><\n";
    transformBOR($borExpressionArray); // transform boolean OR(||) found in condition of ternary

$upmlText[$countSection] .= "</boolean_or></\n";

$upmlText[$countSection] .= "</value_of_false_condition><\n";
if(sizeof($borExpressionArray)>1)//if there are more than 1 item, which means there is at least one XOR between

$upmlText[$countSection] .= "</boolean_or>\n"; //start a new op
$upmlText[$countSection] .= "</value_of_false_condition>\n";
$upmlText[$countSection] .= "</ternary>\n";

}//end of transFormTernary

//find expressions separate by logical OR(||)
function findBOR($borExpressionArray0, $borExpressionArray1)//expression with possible ORs. the OR array consists of start and end location
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
    global $braceArray, $bracketArray; //, $dBraceArray, $dBracketArray;
    $tempVar = $borExpressionArray0;

    $lbor = $tempVar; //left side of an boolean OR
    $rbor = $tempVar; //right side of an boolean OR
    $borNum = -1; //count number of lor, this number is used as array subscription
    $exStart = $borExpressionArray0; //record the start of an expression
    $exEnd = $borExpressionArray1; //record the end of an expression
    $borStart = $exStart; //start of an expression separated by boolean OR
    $borEnd = $exStart; //end of an expression separated by boolean OR
    $exArray = array(); //record an expression separated by a boolean OR
    $borExArray = array(); //record all the expressions separated by boolean OR
    $dBraceArray=findDynamicBracePair($exStart, $exEnd);

    while($tempVar <= $exEnd)
    {
        if($tokens[$tempVar][0] == T_BOOLEAN_OR ) //find a boolean OR
        {
            if(inBraces($tempVar,$exStart, $exEnd, $dBraceArray)); //if the boolean OR is between a pair of "(" and ")", do nothing but to find the next OR
                else if(inBrackets($tempVar,$exStart, $exEnd, $dBracketArray)); //if the boolean OR is between a pair of "[" and "]", do nothing but to find the next OR
                else
                    
Page 184
$borNum++;
$lbor = $tempVar - 1; //left side of the boolean OR
$rbor = $tempVar + 1; //right side of the boolean OR
$borEnd = $lbor; //end of the left expression separated by the boolean OR

$exArray[0] =$borStart;
$exArray[1] =$borEnd;
array_push($borExArray, $exArray); //add the array in the expression
array list

$borStart = $rbor; //next search start is the right side of the boolean OR

} //end of while looking for boolean ORs
$exArray[0] =$rbor;
$exArray[1] =$exEnd;
array_push($borExArray, $exArray); //the last expression, or the entire expression is no boolean OR

return $borExArray; //return the found array of strings (marked with start and end location) separated by ORs

} //end of findBOR

function transformBOR($borExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;

    $bandExpressionArray = array(); //an array that hold items (start and end) separated by boolean ANDs

    //transform expressions separated by ORs
    //echo "enter transformBOR.<br />
    //echo "borExArray is:<br />
    //print_r($borExArray);
    //echo "<br />

    return $borExArray; //return the found array of strings (marked with start and end location) separated by ORs
for($i=0; $i<sizeof($borExArray); $i++)
{
    //echo "process boolean OR<br />
    $bandExpressionArray = findBand($borExArray[$i]); //identify boolean ANDs
    //echo "expressions separated by boolean AND: <br />
    //print_r($bandExpressionArray);
    //echo "<br />
    if(sizeof($bandExpressionArray)>1) //if there are more than 1 item, which means there is at least one XOR between
        $upmlText[$countSection] .= "<boolean_and>\n"
        transformBand($bandExpressionArray); //transform commas then separate boolean ANDs
        if(sizeof($bandExpressionArray)>1) //if there are more than 1 item, which means there is at least one XOR between
            $upmlText[$countSection] .= "</boolean_and>\n"
    }
//end for loop
}//end of transformBOR

//find expressions separate by logical OR(||)
function findBand($bandExpArray) //expression with possible ORs. the OR array consists of start and end location
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray, $bracketArray; //, $dBraceArray, $dBracketArray;
    $tempVar = $bandExpArray[0];

    $lband = $tempVar; //left side of an boolean AND
    $rband = $tempVar; //right side of an boolean AND
    $bandNum = -1; //count number of boolean AND, this number is used as array subscription
    $exStart = $bandExpArray[0]; //record the start of an expression
    $exEnd = $bandExpArray[1]; //record the end of an expression
    $bandStart = $exStart; //start of an expression separated by boolean AND
    $bandEnd = $exStart; //end of an expression separated by boolean AND
    $exArray = array(); //record an expression separated by a boolean AND
    $bandExArray = array(); //record all the expressions separated by boolean AND
$dBraceArray = findDynamicBracePair($exStart, $exEnd);

while ($tempVar <= $exEnd)
{
    if ($tokens[$tempVar][0] == T_BOOLEAN_AND) // find a logical boolean AND
    {
        if (inBraces($tempVar, $exStart, $exEnd, $dBraceArray)); // if the boolean AND is between a pair of "(" and ")", do nothing but to find the next OR
            else if (inBrackets($tempVar, $exStart, $exEnd, $dBracketArray)); // if the boolean AND is between a pair of "[" and "]", do nothing but to find the next AND
        else
        {
            $bandNum++;
            $lband = $tempVar - 1; // left side of the boolean boolean AND
            $rband = $tempVar + 1; // right side of the boolean boolean AND
            $bandEnd = $lband; // end of the left expression separated by the boolean AND
            $exArray[0] = $bandStart;
            $exArray[1] = $bandEnd;
            array_push($bandExArray, $exArray); // add the array in the expression array list
        }
    }
    $tempVar++;
} // end of while looking for boolean AND
$exArray[0] = $rband; // next search start is the right side of the boolean AND
$exArray[1] = $exEnd;
array_push($bandExArray, $exArray); // the last expression, or the entire expression is no boolean AND
return $bandExArray; // return the found array of strings (marked with start and end location) separated by boolean AND
} // end of findBand

// transform expressions separated by boolean ANDs
function transformBand($bandExArray)
{

Page 187
global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
global $braceArray;

$bitORExpressionArray = array(); //an array that hold items (start and end) separated by bit OR

for($i=0; $i<sizeof($bandExArray); $i++)
{
  //echo "process boolean AND<br />
  $bitORExpressionArray = findBitOR($bandExArray[$i]); //identify boolean bit ORs
  //echo "expressions separated by bit OR: <br />
  //print_r($bitORExpressionArray);
  //echo "<br />
  if(sizeof($bitORExpressionArray)>1)//if there are more than 1 item, which means there is at least one bit ORs between
    $upmlText[$countSection] .= "<bit_or>\n"
    transformBitOR($bitORExpressionArray); //transform commas then separate boolean ANDs
  }
  //end for loop
}
//end of transformBand

function findBitOR($bitOrExpArray)//expression with possible ORs. the OR array consists of start and end location
{
  global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
global $braceArray, $bracketArray; //, $dBraceArray, $dBracketArray;

  $tempVar = $bitOrExpArray[0];
  $lbitOr = $tempVar; //left side of abit Or
  $rbitOr = $tempVar; //right side of a bit Or
  $bitOrNum = -1; //count number of bit Or, this number is used as array subscription
$exStart = $bitOrExpArray[0];  // record the start of an expression
$exEnd = $bitOrExpArray[1];    // record the end of an expression
$bitOrStart = $exStart;        // start of an expression separated by bit Or
$bitOrEnd = $exStart;          // end of an expression separated by bit Or
$exArray = array();           // record an expression separated by a bit Or
$bitOrExpressionArray = array();  // record all the expressions separated by bit Or
$dBraceArray = findDynamicBracePair($exStart, $exEnd);

while ($tempVar <= $exEnd) {
    if ($tokens[$tempVar][0] == "|") { // find a bit Or
        if (inBraces($tempVar, $exStart, $exEnd, $dBraceArray)); // if the bit Or is
            // between a pair of "(" and ")", do nothing but to find the next bit Or
        else if (inBrackets($tempVar, $exStart, $exEnd, $dBracketArray)); // if the bit OR is between a pair of "[" and "]", do nothing but to find the bit OR
        else
            {
                $bitOrNum++;
                $lbitOr = $tempVar - 1; // left side of the boolean bit Or
                $rbitOr = $tempVar + 1; // right side of the boolean bit Or
                $bitOrEnd = $lbitOr; // end of the left expression separated by the bit Or
                $exArray[0] = $bitOrStart;
                $exArray[1] = $bitOrEnd;
                array_push($bitOrExpressionArray, $exArray);  // add the array in the expression array list
                $bitOrStart = $rbitOr; // next search start is the right side of the bit Or
            }
    } else {$tempVar++;
        // end of while looking for bit Or
    }
}

$exArray[0] = $rbitOr;
$exArray[1] = $exEnd;
array_push($bitOrExpressionArray, $exArray); // the last expression, or the entire expression is no bit Or

return $bitOrExpressionArray; // return the found array of strings (marked with start
and end location) separated by bit Or
} //end of findBitOR

//transform expressions separated by bit ORs
function transformBitOR($bitOrExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;

    $bitXorExpressionArray = array(); //an array that hold items (start and end) separated by bit ORs
    for($i=0; $i<sizeof($bitOrExArray); $i++)
    {
        $bitXorExpressionArray = findBitXor($bitOrExArray[$i]); //identify bit Xors
        if(sizeof($bitXorExpressionArray)>1) //if there are more than 1 item, which means there is at least one bit Xors between
            $upmlText[$countSection] .= "<bit_xor>\n";

            transformBitXor($bitXorExpressionArray); //transform commas then separate bit Xors
        
        if(sizeof($bitXorExpressionArray)>1) //if there are more than 1 item, which means there is at least one bit Xors between
            $upmlText[$countSection] .= "</bit_xor>\n";
    } //end for loop
} //end of transformBitOR

function findBitXor($bitXorExpArray) //expression with possible Xor. the Xor array consists of start and end location
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray, $braceArray; //, $dBraceArray, $dBracketArray;

    $tempVar = $bitXorExpArray[0];
$lbitXor = $tempVar; // left side of abit Xor
$rbitXor = $tempVar; // right side of a bit Xor
$bitXorNum = -1; // count number of bit Xor, this number is used as array subscription
$exStart = $bitXorExpArray[0]; // record the start of an expression
$exEnd = $bitXorExpArray[1]; // record the end of an expression
$bitXorStart = $exStart; // start of an expression separated by bit Xor
$bitXorEnd = $exStart; // end of an expression separated by bit Xor
$exArray = array(); // record an expression separated by a bit Xor
$bitXorExpressionArray = array(); // record all the expressions separated by bit Xor
$braceArray = findDynamicBracePair($exStart, $exEnd);
while ($tempVar <= $exEnd)
{
    if ($tokens[$tempVar][0] == "^") // find a bit Xor
    {
        if (inBraces($tempVar, $exStart, $exEnd, $dBraceArray)); // if the bit And is between a pair of "(" and ")", do nothing but to find the next bit And
        elseif (inBrackets($tempVar, $exStart, $exEnd, $dBracketArray)); // if the bit AND is between a pair of "[" and "]", do nothing but to find the bit AND
        else
        {
            $bitXorNum++;
            $lbitXor = $tempVar - 1; // left side of the bit Xors
            $rbitXor = $tempVar + 1; // right side of the bit Xors
            $bitXorEnd = $lbitXor; // end of the left expression separated by the bit Xors
            $exArray[0] = $bitXorStart;
            $exArray[1] = $bitXorEnd;
            array_push($bitXorExpressionArray, $exArray); // add the array in the expression array list
            $bitXorStart = $rbitXor; // next search start is the right side of the bit Xors
        }
    }
    $tempVar++;
} // end of while looking for bit Xor
$exArray[0] = $rbitXor;
$exArray[1] = $exEnd;

Page 191
array_push($bitXorExpressionArray, $exArray); //the last expression, or the entire expression is no bit Xors
return $bitXorExpressionArray; //return the found array of strings (marked with start and end location) separated by bit Xors
}//end of findBitXor

//transform expressions separated by bit Xor
function transformBitXor($bitXorExpressionArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
    global $braceArray;

    $bitXorExpressionArray = array(); //an array that hold items (start and end) separated by bit Xors
    for($i=0; $i<sizeof($bitXorExpressionArray); $i++)
    {
        //echo "process bit Xor<br />
        $bitAndExpressionArray = findBitAnd($bitXorExpressionArray[$i]); //identify bit Ands
        //echo "expressions separated by bit And: <br />
        //print_r($bitAndExpressionArray);
        //echo "<br />
        //echo "size of bit and is: ".sizeof($bitAndExpressionArray)."<br />
        if(sizeof($bitAndExpressionArray)>1) //if there are more than 1 item, which means there is at least one bit Ands between
            $upmlText[$countSection] .= "<bit_and>
        transformBitAnd($bitAndExpressionArray); //transform commas then separate bit Ands
        if(sizeof($bitAndExpressionArray)>1) //if there are more than 1 item, which means there is at least one bit Ands between
            $upmlText[$countSection] .= "</bit_and>
    }//end for loop
}//end of transformBitXor

function findBitAnd($bitAndExpArray)//expression with possible And. the And array consists of start and end location
global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;

$braceArray;

$tempVar = $bitAndExpArray[0];

$lbitAnd = $tempVar; //left side of a bit And
$rbitAnd = $tempVar; //right side of a bit And
$bitAndNum = -1; //count number of bit Or, this number is used as array subscription
$exStart = $bitAndExpArray[0]; //record the start of an expression
$exEnd = $bitAndExpArray[1]; //record the end of an expression
$bitAndStart = $exStart; //start of an expression separated by bit And
$bitAndEnd = $exStart; //end of an expression separated by bit And
$bitAndExpressionArray = array(); //record all the expressions separated by bit And

while($tempVar <= $exEnd)
{
    if($tokens[$tempVar][0] == "&") //find a bit And
    {
        $bitAndNum++;
        $lbitAnd = $tempVar - 1; //left side of the bit Ands
        $rbitAnd = $tempVar + 1; //right side of the bit Ands
        $bitAndEnd = $lbitAnd; //end of the left expression separated by the bit Ands

        $exArray[0]=$bitAndStart;
        $exArray[1]=$bitAndEnd;
        array_push($bitAndExpressionArray, $exArray); //add the array in the expression array list
    }
}
$tempVar++; //end of while looking for bit And
$exArray[0]=$rbitAnd;
$exArray[1]=$exEnd;
array_push($bitAndExpressionArray, $exArray); //the last expression, or the entire expression is no bit Xors
return $bitAndExpressionArray; //return the found array of strings (marked with start and end location) separated by bit Xors
}//end of findBitAnd

//transform expressions separated by bit And
function transformBitAnd($bitAndExArray)
{
  global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
  global $braceArray;
  $compareExpressionArray = array(); //an array that hold items (start and end) separated by bit Xors
  for($i=0; $i<sizeof($bitAndExArray); $i++)
  {
    $compareExpressionArray = findCompare($bitAndExArray[$i]); //identify comparison
    transformCompare($compareExpressionArray); //transform commas then separate ANDs
  }//end for loop
}//end of transformBitAnd

//find expressions separated by comparisons
function findCompare($compareExArray)
{
  global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
  global $braceArray, $bracketArray; // $dBraceArray, $dBracketArray;

$tempVar = $compareExArray[0];
$lcomp = $tempVar;  //left side of an Comparison
$rcomp = $tempVar;  //right side of an Comparison
$compNum = -1;  //count number of Comparison, this number is used as array subscription
$exStart = $compareExArray[0];  //record the start of an expression
$exEnd = $compareExArray[1];  //record the end of an expression
$compStart = $exStart;  //start of an expression separated by Comparison
$compEnd = $exStart;  //end of an expression separated by Comparison
$exArray = array();  //record an expression separated by a Comparison
$compExArray = array();  //record all the expressions separated by Comparison
$markCompare = "compare";  //mark the last compare operator, initially "+="
$dBraceArray=findDynamicBracePair($exStart, $exEnd);

while($tempVar <= $exEnd)
{
    if(($tokens[$tempVar][0] == T_IS_EQUAL) ||  //find a comparison sign
        ($tokens[$tempVar][0] == T_IS_NOT_EQUAL) ||
        ($tokens[$tempVar][0] == T_IS_IDENTICAL) ||
        ($tokens[$tempVar][0] == T_IS_NOT_IDENTICAL))
    {
        if(inBraces($tempVar,$exStart, $exEnd, $dBraceArray));  //if the comparison is
            between a pair of "(" and ")", do nothing but to find the next Comparison
        else if(inBrackets($tempVar,$exStart, $exEnd, $dBracketArray));  //if the
            comparison is between a pair of "[" and "]", do nothing but to find the next comparison
        else
        {
            $compNum++;
            $lcomp = $tempVar - 1;  //left side of the comparison
            $rcomp = $tempVar + 1;  //right side of the comparison
            $compEnd = $lcomp;  //end of the left expression separated by the
            Comparison
            $exArray[0]=$compStart;
            $exArray[1]=$compEnd;
            if($tokens[$tempVar][0] == T_IS_EQUAL)$exArray[2]="is_equal";
            if($tokens[$tempVar][0] == T_IS_NOT_EQUAL)$exArray[2]="not_equal";
            if($tokens[$tempVar][0] == T_IS_IDENTICAL)$exArray[2]="is_identical";
        }
    }
}

Page 195
if ($tokens[$tempVar][0] == T_IS_NOT_IDENTICAL) $exArray[2] = "not_identical";
array_push($compExArray, $exArray); // add the array in the expression
array list
$compStart = $rcomp; // next search start is the right side of the Comparison
$markCompare = $exArray[2]; // mark the assignment operator for the last item
}
} // end else
} // end if
$tempVar++;
} // end of while looking for Comparison
$exArray[0] = $rcomp;
$exArray[1] = $exEnd;
$exArray[2] = $markCompare;
array_push($compExArray, $exArray); // the last expression, or the entire expression has no Comparison
return $compExArray; // return the found array of strings (marked with start and end location) separated by Comparison
} // end of findCompare

// transform Comparison and find high level comparisons
function transformCompare($compExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;

    $hicompareExpressionArray = array(); // an array that hold items (start and end) separated by Ternary

    $markCompOp = array(); // record location where appear a different operator
    $markNum = 0; // record number of different operators
    $markCompOp[$markNum] = 0; // the start position, for the first op

    $sizeCompExArray = sizeof($compExArray);
if ($sizeCompExArray > 1)
{
    // search the positions that appear a different operator
    for ($j = 1; $j < $sizeCompExArray; $j++)
    {
        if ($compExArray[$j][2] <> $compExArray[$j - 1][2])
        {
            $markNum++;
            $markCompOp[$markNum] = $j;
        }
    }
}
for ($i = 0; $i < $sizeCompExArray; $i++)
{
    $hicOMPareExpressionArray = findHicOMPare($compExArray[$i]); // left item of an assignment op; identify Ternary
    if ($sizeCompExArray <= 1)
        transformHicOMPare($hicOMPareExpressionArray); // transform ternary then boolean or
    else // more than one item
    {
        // echo "enter transformCompare else <br />";
        if (in_array($i, $markCompOp)) // recorded op that be different from the previous op
            $upmlText[$countSection] .= "<".$compExArray[$i][2]."\n"; // start a new op
        transformHicOMPare($hicOMPareExpressionArray); // transform ternary then boolean or
        if ($i == ($sizeCompExArray - 1)) // the last item
            for ($k = $markNum; $k >= 0; $k--) // end each of the marked operators
                $upmlText[$countSection] .= "</".$compExArray[$markCompOp[$k]][2]."\n";
        Page 197
//find expressions separated by high level comparisons
function findHicompare($hicompareExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray, $bracketArray, $dBraceArray, $dBracketArray;

    $tempVar = $hicompareExArray[0];

    $lhicomp = $tempVar; //left side of a Comparison
    $rhicomp = $tempVar; //right side of a Comparison
    $hicompNum = -1; //count number of Comparison, this number is used as array subscription

    $exStart = $hicompareExArray[0]; //record the start of an expression
    $exEnd = $hicompareExArray[1]; //record the end of an expression
    $hicompStart = $exStart; //start of an expression separated by Comparison
    $hicompEnd = $exStart; //end of an expression separated by Comparison
    $exArray = array(); //record an expression separated by a Comparison
    $hicompExArray = array(); //record all the expressions separated by Comparison
    $markHicompare = "compare"; //mark the assignment operator for the last item
    $dBraceArray = findDynamicBracePair($exStart, $exEnd);

    while ($tempVar <= $exEnd) {
        if (($tokens[$tempVar][0] == "<") || //find a comparison sign
            ($tokens[$tempVar][0] == T_IS_SMALLER_OR_EQUAL) ||
            ($tokens[$tempVar][0] == ">") ||
            ($tokens[$tempVar][0] == T_IS_GREATER_OR_EQUAL)) {
            if (inBraces($tempVar, $exStart, $exEnd, $dBraceArray)); //if the or is between a pair of "(" and ")", do nothing but to find the next Comparison
else if (inBrackets($tempVar,$exStart, $exEnd, $dBracketArray)); //if the boolean OR is between a pair of "[" and "]", do nothing but to find the next Smaller_equal

else
{
    $hicompNum++;
    $1hicomp = $tempVar - 1; //left side of the comparison
    $rhicomp = $tempVar + 1; //right side of the comparison
    $hicompEnd = $1hicomp; //end of the left expression separated by the Comparison
    $exArray[0]=$hicompStart;
    $exArray[1]=$hicompEnd;
    if ($tokens[$tempVar][0] == "<")$exArray[2]="smaller";
    if ($tokens[$tempVar][0] == T_IS_SMALLER_OR_EQUAL)$exArray[2]="smaller_equal";
    if ($tokens[$tempVar][0] == ">")$exArray[2]="greater";
    if ($tokens[$tempVar][0] == T_IS_GREATER_OR_EQUAL)$exArray[2]="greater_equal";

    array_push($hicompExArray, $exArray); //add the array in the expression array list
    $hicompStart = $rhicomp; //next search start is the right side of the Comparison
    $markHicompare = $exArray[2]; //mark the assignment operator for the last item
}

}//end else

}//end if

$tempVar++; //end of while looking for Comparison
$exArray[0]=$rhicomp;
$exArray[1]=$exEnd;
$exArray[2]=$markHicompare;
array_push($hicompExArray, $exArray); //the last expression, or the entire expression has no Comparison

return $hicompExArray; //return the found array of strings (marked with start and end location) separated by Comparison
} //end of findHicompare

//transform higher level Comparisons and find shift
function transformHicompare($hicompExArray)
{
  global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
  $currentClassName, $countSection;
  global $braceArray;

  $shiftExpressionArray = array(); //an array that hold items (start and end) separated by Ternary

  $markHicompOp=array(); //record location where appear a different operator
  $markNum = 0; //record number of different operators
  $markHicompOp[$markNum]=0; //the start position, for the first op

  $sizeHicompExArray = sizeof($hicompExArray);
  //echo "sizeHicompExArray=":$sizeHicompExArray."<br />";

  if($sizeHicompExArray>1)
  {
    //search the positions that appear a different operator
    for($j=1; $j<$sizeHicompExArray; $j++)
    {
      if($hicompExArray[$j][2] <> $hicompExArray[$j-1][2])
      {
        $markNum++;
        $markHicompOp[$markNum]=$j;
        //echo "markNum=":$markNum."<br />
        markHicompOp=".$markHicompOp[$markNum]."<br />";
      }
    }
    for($i=0; $i<$sizeHicompExArray; $i++)
    {
      $shiftExpressionArray = findShift($hicompExArray[$i]); //left item of an assignment op; identify Ternary
    }
  }
if(sizeof($hicompExArray)<=1)
    transformShift($shiftExpressionArray);//transform shifts then add, sub, concat

else//more than one item
{
    //echo "enter transformHicompare else <br />
    if(in_array($i, $markHicompOp))//recorded op that be different from
    the previous op
        $upmlText[$countSection] .= "<".$hicompExArray[$i][2]."\n";//start a new op

    transformShift($shiftExpressionArray);//transform ternary then boolean or

    if($i==($sizeHicompExArray-1))//the last item
        for($k=$markNum; $k>=0; $k--)//end each of the marked operators
        {"$upmlText[$countSection] .= "</".$hicompExArray[$markHicompOp[$k]][2]."\n";
        }
    }//end for loop
}//end of transformHicompare

//find expressions separated by shifts
function findShift($shiftExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray, $bracketArray, $dBraceArray, $dBracketArray;
    $tempVar = $shiftExArray[0];

    $lshift = $tempVar; //left side of an Comparison
    $rshift = $tempVar; //right side of an Comparison
$shiftNum = -1; //count number of Comparison, this number is used as array subscription
$exStart = $shiftExArray[0]; //record the start of an expression
$exEnd = $shiftExArray[1]; //record the end of an expression
$shiftStart = $exStart; //start of an expression separated by Comparison
$shiftEnd = $exStart; //end of an expression separated by Comparison
$exArray = array(); //record an expression separated by a Comparison
$shiftExArray = array(); //record all the expressions separated by Comparison
$markShift = "shift"; //mark the assignment operator for the last item
$dBraceArray=findDynamicBracePair($exStart, $exEnd);

while($tempVar <= $exEnd)
{
    if((($tokens[$tempVar][0] == T_SL )|| ($tokens[$tempVar][0] == T_SR))
    {
        if(inBraces($tempVar,$exStart, $exEnd, $dBraceArray)); //if the shift is between a pair of "(" and ")", do nothing but to find the next shift
            else if(inBrackets($tempVar,$exStart, $exEnd, $dBracketArray)); //if the shift is between a pair of "[" and "]", do nothing but to find the next shift
                                        else
                                    {
                                        $shiftNum++;
                                        $lshift = $tempVar - 1; //left side of the shift
                                        $rshift = $tempVar + 1; //right side of the shift
                                        $shiftEnd = $lshift; //end of the left expression separated by the shift
                                        $exArray[0]=$shiftStart;
                                        $exArray[1]=$shiftEnd;
                                        if($tokens[$tempVar][0] == T_SL)$exArray[2]="shift_left";
                                        if($tokens[$tempVar][0] == T_SR)$exArray[2]="shift_right";
                                        array_push($shiftExArray, $exArray); //add the array in the expression array list
                                        $shiftStart = $rshift; //next search start is the right side of the shift
                                        $markShift = $exArray[2]; //mark the assignment operator for the last item
                                    }
    }
}
} // end else
} // end if
$tempVar++;
} // end of while looking for shift
$exArray[0]=$rshift;
$exArray[1]=$exEnd;
$exArray[2]=$markShift;
array_push($shiftExArray, $exArray); // the last expression, or the entire expression
has no shift
return $shiftExArray; // return the found array of strings (marked with start and end
location) separated by shift
} // end of findShift

// transform Shift and find add_sub_concat
function transformShift($shiftExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
$currentClassName, $countSection;
    global $braceArray, $dBraceArray, $dBracketArray;

    $addsubconcatExpressionArray = array(); // an array that hold items (start and end)
separated by add, sub, or concat

    $markShiftOp=array(); // record location where appear a different operator
    $markNum = 0; // record number of different operators
    $markShiftOp[$markNum]=0; // the start position, for the first op

    $sizeShiftExArray = sizeof($shiftExArray);
    if($sizeShiftExArray>1)
    {
        // search the positions that appear a different operator
        for($j=1; $j<$sizeShiftExArray; $j++)
        {
            if($shiftExArray[$j][2] <> $shiftExArray[$j-1][2])
            {
                $markNum++;
                $markShiftOp[$markNum]=$j;
        ...
for($i=0; $i<$sizeShiftExArray; $i++)
{
    $addsubconcatExpressionArray = findAddsubconcat($shiftExArray[$i]); // left item of an assignment op; identify Ternary
    if($sizeShiftExArray<=1)
        transformAddsubconcat($addsubconcatExpressionArray); // transform Add/Sub/Concatenation then mul/div/modulus
    else // more than one item
    {
        if(in_array($i, $markShiftOp)) // recorded op that be different from the previous op
            $upmlText[$countSection] .="<".$shiftExArray[$i][2].">
        transformAddsubconcat($addsubconcatExpressionArray); // transform add/sub/concatenation then mul/div/modulus
        if($i==($sizeShiftExArray-1)) // the last item
            for($k=$markNum; $k>=0; $k--) // end each of the marked operators
            {
                $upmlText[$countSection] .="</".$shiftExArray[$markShiftOp[$k]][2].">\n"
            }
    }
} // end for loop
} // end of transformShift

// find expressions separated by add, sub, or concatenate
function findAddsubconcat($compareExArray)
global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;

global $braceArray, $bracketArray; //, $dBraceArray, $dBracketArray;
$tempVar = $compareExArray[0];

$lcomp = $tempVar; //left side of an Comparison
$rcomp = $tempVar; //right side of an Comparison
$compNum = -1; //count number of Comparison, this number is used as array subscription
$exStart = $compareExArray[0]; //record the start of an expression
$exEnd = $compareExArray[1]; //record the end of an expression
$compStart = $exStart; //start of an expression separated by Comparison
$compEnd = $exStart; //end of an expression separated by Comparison
$exArray = array(); //record an expression separated by a Comparison
$compExArray = array(); //record all the ex expressions separated by Comparison
$markAddsubconcat = "add_sub_concat"; //mark the assignment operator for the last item
$dBraceArray=findDynamicBracePair($exStart, $exEnd);

while ($tempVar <= $exEnd)
{
    if (($tokens[$tempVar][0] == "+") || //find a add, sub, or concatenation sign
        ($tokens[$tempVar][0] == "-") ||
        ($tokens[$tempVar][0] == ".")
    )
    {
        if (inBraces($tempVar, $exStart, $exEnd, $dBraceArray)); //if the add, sub, or concat is between a pair of "(" and ")", do nothing but to find the next add, sub, or concat
            else if (inBrackets($tempVar, $exStart, $exEnd, $dBracketArray)); //if the add, sub, or concat is between a pair of "[" and "]"}, do nothing but to find the next add, sub, or concat
            else
            {
                $compNum++;
                $lcomp = $tempVar - 1; //left side of the add, sub, or concatenation
                $rcomp = $tempVar + 1; //right side of the add, sub, or concatenation
                $compEnd = $lcomp; //end of the left expression separated by the add, sub, or concatenation
                $exArray[0] = $compStart;
            }

Page 205
$exArray[1]=$compEnd;

if ($tokens[$tempVar][0] == "+") $exArray[2]="add";
else if ($tokens[$tempVar][0] == "+") $exArray[2]="add";
else if ($tokens[$tempVar][0] == "-") $exArray[2]="subtract";
else if ($tokens[$tempVar][0] == ".") $exArray[2]="concatenate";

array_push($compExArray, $exArray); // add the array in the expression array list
$compStart = $rcomp; // next search start is the right side of the add, sub, or concatenation
$markAddsubconcat = $exArray[2]; // mark the assignment operator for the last item

} // end else
} // end if
$compStart = $rcomp; // next search start is the right side of the add, sub, or concatenation
$markAddsubconcat = $exArray[2]; // mark the assignment operator for the last item

 }//end of while looking for add, sub, concat
$exArray[0]=$rcomp;
$exArray[1]=$exEnd;
$exArray[2]=$markAddsubconcat;
array_push($compExArray, $exArray); // the last expression, or the entire expression

return $compExArray; // return the found array of strings (marked with start and end location) separated by add, sub, or concatenation

} // end of findAddsubconcat

// transform Shift and find add_sub_concat
function transformAddsubconcat($ascExArray)
{

global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
$countClassName, $countSection;

global $braceArray;

$ascExpressionArray = array(); // an array that hold items (start and end) separated by
Ternary

$markAscOp=array(); // record location where appear a different operator
$markNum = 0; // record number of different operators
$markAscOp[$markNum]=0; // the start position, for the first op
$sizeAscExArray = sizeof($ascExArray);
//echo "sizeAscExArray="."$sizeAscExArray."<br /">;
if($sizeAscExArray>1)
{
    //search the positions that appear a different operator
    for($j=1; $j<$sizeAscExArray; $j++)
    {
        if($ascExArray[$j][2] <> $ascExArray[$j-1][2])
        {
            $markNum++;
            $markAscOp[$markNum]=$j;
        }
    }
}
for($i=0; $i<$sizeAscExArray; $i++)
{
    $muldivmodExpressionArray = findMuldivmod($ascExArray[$i]); //left item of an
    //assignment op; identify Ternary
    if($sizeAscExArray<=1)
        transformMuldivmod($muldivmodExpressionArray); //transform mul/div/mod
    else
    {
        //more than one item
        //echo "enter transformMuldivmod else <br />
        if(in_array($i, $markAscOp))
            //recorded op that be different from the
        a new op
            $upmlText[$countSection] .= "<"."$ascExArray[$i][2]."\\n"; //start
        boolean or
            transformMuldivmod($muldivmodExpressionArray); //transform ternary then
        if($i==($sizeAscExArray-1)) //the last item
            for($k=$markNum; $k>=0; $k--) //end each of the marked operators
            {
                $upmlText[$countSection] .=
            Page 207
"</".$ascExArray[$markAscOp[$k]][2].">\n";
}

}  //end for loop
}  //end of transformAddsubconcat

//find expressions separated by mul, div, and mod
function findMuldivmod($mdmExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;

    $braceArray, $bracketArray;//, $dBraceArray, $dBracketArray;

    $tempVar = $mdmExArray[0];

    $lmdm = $tempVar; //left side of a mul, or div, or mod
    $rmdm = $tempVar; //right side of a mul, or div, or mod
    $mdmNum = -1; //count number of mul, or div, or mod, this number is used as array

    $exStart = $mdmExArray[0];  //record the start of an expression
    $exEnd = $mdmExArray[1];    //record the end of an expression
    $mdmStart = $exStart;       //start of an expression separated by a mul, or div, or mod
    $mdmEnd = $exStart;         //end of an expression separated by a mul, or div, or mod
    $exArray = array();        //record an expression separated by a a mul, or div, or mod
    $mdmExArray = array();     //record all the expressions separated by a mul, or div, or mod
    $markMdm = "mul div mod";  //mark the mul/div/mod operator for the last item
    $dBraceArray = findDynamicBracePair($exStart, $exEnd);

    while($tempVar <= $exEnd)
    {
        if((($tokens[$tempVar][0] == "+") || // find a a mul, or div, or mod sign
            ($tokens[$tempVar][0] == "/") ||
            ($tokens[$tempVar][0] == "%"))
        {
            if(inBraces($tempVar,$exStart, $exEnd, $dBraceArray)); //if the mul, div, mod
                is between a pair of "(" and ")", do nothing but to find the next mul, div, mod
        }
    }
}
else if (inBrackets($tempVar, $exStart, $exEnd, $dBracketArray)); // if the mul, div, mod is between a pair of "[" and "]", do nothing but to find the next mul, div, mod
else
{
    $mdmNum++;  
    $lmdm = $tempVar - 1; // left side of the add, sub, or concatenation
    $rmdm = $tempVar + 1; // right side of the add, sub, or concatenation
    $mdmEnd = $lmdm;    // end of the left expression separated by the add, sub, or concatenation
    $exArray[0] = $mdmStart;
    $exArray[1] = $mdmEnd;
    if ($tokens[$tempVar][0] == "+") $exArray[2] = "multiply";
    else if ($tokens[$tempVar][0] == "/") $exArray[2] = "divide";
    else if ($tokens[$tempVar][0] == "%") $exArray[2] = "modulo";
    array_push($mdmExArray, $exArray); // add the array in the expression array list
    $mdmStart = $rmdm; // next search start is the right side of the mul, div, or mod
    $markMdm = $exArray[2]; // mark the mul/div/mod operator for the last item
} // end else
} // end if
$tempVar++;  
// end of while looking for mul, div, mod
$exArray[0] = $rmdm;
$exArray[1] = $exEnd;
$exArray[2] = $markMdm;
array_push($mdmExArray, $exArray); // the last expression, or the entire expression
return $mdmExArray; // return the found array of strings (marked with start and end location) separated by mul, div, and mod
} // end of findMuldivmod

// transform mul, div, mod and find logical NOT (!)
function transformMuldivmod($mdmExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
$currentClassName, $countSection;

    global $braceArray;

    $mdmExpressionArray = array();  // an array that hold items (start and end) separated by

Ternary

$markMdmOp = array();  // record location where appear a different operator
$markNum = 0;  // record number of different operators
$markMdmOp[$markNum] = 0;  // the start position, for the first op

$sizeMdmExArray = sizeof($mdmExArray);
if ($sizeMdmExArray > 1) {
    // search the positions that appear a different operator
    for ($j = 1; $j < $sizeMdmExArray; $j++) {
        if ($mdmExArray[$j][2] <> $mdmExArray[$j - 1][2]) {
            $markNum++;
            $markMdmOp[$markNum] = $j;
        }
    }
}
for ($i = 0; $i < $sizeMdmExArray; $i++) {
    $notExpressionArray = findNot($mdmExArray[$i]);  // left item of an assignment

    op; identify Ternary

    if ($sizeMdmExArray <= 1)
        transformNot($notExpressionArray);  // transform Not then instanceof

    else // more than one item
    {
        if (in_array($i, $markMdmOp))  // recorded op that be different from the
            previous op
        $upmlText[$countSection] .= "<".$mdmExArray[$i][2]."/>
";  // start a new

    op

    Page 210
transformNot($notExpressionArray);//transform Not then instanceof

if($i==($sizeMdmExArray-1)) //the last item
    for($k=$markNum; $k>=0; $k--) //end each of the marked operators
        {
            $upmlText[$countSection] .= "</".$mdmExArray[$markMdmOp[$k]][2].">
        }
    } //end for loop
} //end of transformMuldivmod

function findNot($notExpArray)//expression with possibl Not
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
             $currentClassName, $countSection;
    global $braceArray;

    $notExpressionArray = array();

    $tempVar = $notExpArray[0];

    if($tokens[$tempVar] == "!")
        {
            $notExpressionArray[0]=$tempVar + 1; //start of not expression
            $notExpressionArray[1]=$notExpArray[1]; //end of not expression
            $notExpressionArray[2]="logical_not"; //operator not
        }
    else
        {
            $notExpressionArray[0]=$tempVar; //start of not expression
            $notExpressionArray[1]=$notExpArray[1]; //end of not expression
            $notExpressionArray[2]="_"; //no not
        }
}
return $notExpressionArray; //return the found array of strings (marked with start and end location) separated by not
} //end of findNot

//transform expressions negated not (!) then instance of
function transformNot($notExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;
    $
    $instanceofExpressionArray = array(); //an array that hold items (start and end)
    $instanceofExpressionArray = findInstanceof($notExArray); //identify instanceof
    
    if($notExArray[2] == "logical_not")
        $upmlText[$countSection] .= "<logical_not>
    transformInstanceof($instanceofExpressionArray);
    if($notExArray[2] == "logical_not")
        $upmlText[$countSection] .= "</logical_not>\n"
} //end of transformNot

function findInstanceof($instanceofExpArray) //expression with possible ORs. the OR array
consists of start and end location
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray, $bracketArray; //, $dBraceArray, $dBracketArray;
    
    $tempVar = $instanceofExpArray[0];
    $linst = $tempVar; //left side of abit Or
    $rinst = $tempVar; //right side of a bit Or
    $instNum = -1; //count number of bit Or, this number is used as array subscription
    $exStart = $instanceofExpArray[0]; //record the start of an expression
    $exEnd = $instanceofExpArray[1]; //record the end of an expression
$instStart = $exStart;  // start of an expression separated by bit Or
$instEnd = $exStart;  // end of an expression separated by bit Or
$exArray = array();  // record an expression separated by a bit Or
$instExpressionArray = array();  // record all the expressions separated by bit Or
$dBraceArray = findDynamicBracePair($exStart, $exEnd);

while ($tempVar <= $exEnd) {
    if ($tokens[$tempVar][0] == T_INSTANCEOF)  // find an instanceof
    {
        if (inBraces($tempVar, $exStart, $exEnd, $dBraceArray));  // if the bit Or is
            // between a pair of "(" and ")", do nothing but to find the next bit Or
        else if (inBrackets($tempVar, $exStart, $exEnd, $dBracketArray));  // if the bit
            // OR is between a pair of "[" and "]", do nothing but to find the bit OR
        else
        {
            $instNum++;  
            $linst = $tempVar - 1;  // left side of the boolean inst
            $rinst = $tempVar + 1;  // right side of the boolean inst
            $instEnd = $linst;  // end of the left expression separated by the inst
            $exArray[0] = $instStart;
            $exArray[1] = $instEnd;
            array_push($instExpressionArray, $exArray);  // add the array in the
            // expression array list
        }
    }
    $tempVar++;
}  // end of while looking for instanceof

$exArray[0] = $rinst;
$exArray[1] = $exEnd;
array_push($instExpressionArray, $exArray);  // the last expression, or the entire
expression is no bit Or

return $instExpressionArray;  // return the found array of strings (marked with start and end location) separated by bit Or
function transformInstanceOf($instanceofExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;

    $idcExpressionArray = array(); //an array that hold items (start and end) prepended by
    inc, dec, or cast

    $sizeofInstanceExArray = sizeof($instanceofExArray);
    if ($sizeofInstanceExArray>1 ) //if there are more than 1 item, which means there is at
    least one instanceof
        $upmlText[$countSection] .= "<instanceof>
    
    for($i=0; $i<$sizeofInstanceExArray; $i++)
    {
        $idcExpressionArray = findIncDecCast($instanceofExArray[$i]); //identify inc, dec, cast
        transformIncDecCast($idcExpressionArray); //transform commas then separate bit Xors
    } //end for loop
    if ($sizeofInstanceExArray>1) //if there are more than 1 item, which means there is at
    least one bit Xors between
        $upmlText[$countSection] .= "</instanceof>
    
    }//end of transformInstanceOf

    //find expressions prepended with ++ -- ~ and type casts
    function findIncDecCast($idcExArray)
    {
        global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
        $currentClassName, $countSection;
        global $braceArray, $bracketArray; //, $dBraceArray, $dBracketArray;

Page 214
$tempVar = $idcExArray[0];

$exStart = $idcExArray[0]; //record the start of an expression
$exEnd = $idcExArray[1]; //record the end of an expression
$exArray = array(); //record an expression separated by a Inc, Dec, and Type casts
$idcExpressionArray = array(); //record all the expressions separated by Inc, Dec, and
Type casts

$dBraceArray=findDynamicBracePair($exStart, $exEnd);
if (($tokens[$tempVar][0] == T_INC )|| //find a inc sign
($tokens[$idcExArray[1]][0] == T_INC )|| //find a post inc sign
($tokens[$tempVar][0] == T_DEC)|| //find a dec sign
($tokens[$idcExArray[1]][0] == T_DEC)|| //find a post dec sign
($tokens[$tempVar][0] == "~") || //find a negation sign
($tokens[$tempVar][0] == T_INT_CAST) || //find a int cast (int)
($tokens[$tempVar][0] == T_DOUBLE_CAST) || //find a real cast (real), (double),
(float)
($tokens[$tempVar][0] == T_STRING_CAST) || //find a string cast (string)
($tokens[$tempVar][0] == T_ARRAY_CAST) || //find an array cast (array)
($tokens[$tempVar][0] == T_OBJECT_CAST) || //find an object cast (object)
($tokens[$tempVar][0] == T_BOOL_CAST) || //find a bool cast (bool) or (boolean)
($tokens[$tempVar][0] == "@")) //find an error control operator
{
    if (inBraces($tempVar,$exStart, $exEnd, $dBraceArray)); //if the inc, dec, cast is between a pair of "(" and ")", do nothing but to find the next inc, dec, or cast
    else if (inBrackets($tempVar,$exStart, $exEnd, $dBracketArray)); //if the inc, dec, cast is between a pair of "[" and "]", do nothing but to find the next inc, dec, or cast
    else
    {
        if ($tokens[$idcExArray[1]][0] == T_INC || $tokens[$idcExArray[1]][0] == T_DEC)
        {
            $exArray[0]=$exStart;
            $exArray[1]=$exEnd - 1;
        }
        else
        {
            $exArray[0]=$exStart + 1;
        }
$exArray[1]=$exEnd;

if ($tokens[$tempVar][0] == T_INC) $exArray[2]="pre_increment">
```plaintext
} else if ($tokens[$idcExArray[1]][0] == T_DEC) $exArray[2]="post_decrement">
```plaintext
else if ($tokens[$tempVar][0] == T_DEC) $exArray[2]="pre_decrement">
```plaintext
else if ($tokens[$tempVar][0] == "~") $exArray[2]="negate">
```plaintext
else if ($tokens[$tempVar][0] == T_INT_CAST) $exArray[2]="cast_to_integer">
```plaintext
else if ($tokens[$tempVar][0] == T_DOUBLE_CAST) $exArray[2]="cast_to_double">
```plaintext
else if ($tokens[$tempVar][0] == T_STRING_CAST) $exArray[2]="cast_to_string">
```plaintext
else if ($tokens[$tempVar][0] == T_ARRAY_CAST) $exArray[2]="cast_to_array">
```plaintext
else if ($tokens[$tempVar][0] == T_OBJECT_CAST) $exArray[2]="cast_to_object">
```plaintext
else if ($tokens[$tempVar][0] == T_BOOL_CAST) $exArray[2]="cast_to_bool">
```plaintext
else if ($tokens[$tempVar][0] == "@") $exArray[2]="control_error">
```plaintext
else $exArray[2]="_">
```plaintext
array_push($idcExpressionArray, $exArray); //add the array in the expression array list
```plaintext
```
return $idcExpressionArray;  //return the found array of strings (marked with start and end location) separated by Comparison
}//end of findIncDecCast

//transform Inc, Dec, or Cast
function transformIncDecCast($idcExArray)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;

    $arrayExpressionArray = array();  //an array that hold items (start and end) separated by Ternary

    $markIdcOp = array();  //record location where appear a different operator

    if ($idcExArray[0][2]<>"_")
        $upmlText[$countSection] .= "<".$idcExArray[0][2].">\\n";
    findArrayElement($idcExArray);  //identify array subscription
    if ($idcExArray[0][2]<>"_")
        $upmlText[$countSection] .= "</".$idcExArray[0][2].">\\n";

    return;
}//end of transformIncDecCast

//find expressions of array subscription
function findArrayElement($arrayExArray)
{
    global $countFunction, $levelFunction, $contents, $tokens, $upmlText, $countClass,
    $levelClass, $currentClassName, $countSection;
    global $braceArray;

    $tempVar = $arrayExArray[0][0];
    $arrayExRemaining = $inputStr;  //the remaining array expression is initially set to the input string

    }
$pos = $arrayExArray[0]; //count the number of subscription of an array
$arrayName = "";
$arrayName = array(); //array name could be a string, or an array like obj->myArray[

$numSubLeft = 0; //count the number of "[
$numSubRight = 0; //count the number of "]
$subStart = $arrayExArray[0][0];
$subEnd = $arrayExArray[0][0];
$subScriptions = array(); //store the pairs of start and end of subscriptions

$sub = array(); //array that holds pairs of start and end of subscriptions

//search all subscriptions of the array
for ($i = $arrayExArray[0][0]; $i <= $arrayExArray[0][1]; $i++)
{
    if ($tokens[$i][0]=="[")
    {
        if ($numSubLeft==0)//the first "[" found
        {
            $arrayName = $tokens[$arrayExArray[0][0]][1]; //the first item should be the array name
            $upmlText[$countSection] .= "<array>
            $upmlText[$countSection] .= "<name>
            //echo "$arrayExArray00=$arrayExArray[0][0]." i-1=".($i-1)."<br />
            identifyPrimitive($arrayExArray[0][0], $i-1); //identify the array name:
            a string or a obj->myArray[..]
            //SupmlText[$countSection] .= $arrayName;
            $upmlText[$countSection] .= "</name>
        }
        $numSubLeft++;
        $subStart = $i + 1;
        $subScriptions[0] = $subStart;
    }
    else if ($tokens[$i][0]=="]")
    {
        $numSubRight++;
        $subEnd = $i - 1; //one position before "]"
        $subScriptions[1] = $subEnd;
    }
}
array_push($sub, $subScriptions);
}
}//end for searching subscriptions

if($numSubLeft <> $numSubRight)
{
    echo "there may have error for array subscription<br />
    "$upmlText[$countSection] .= "<warning>There may have error for array subscription</warning><br/>
    "$upmlText[$countSection] .= "</array><br/>

} else
{
if($numSubLeft == 0)identifyCloneNew($arrayExArray); //if no array subscription
else
{
    for($i = 0; $i<$numSubLeft; $i++)
    {
        $upmlText[$countSection] .= "<subscription><br/>
        identifyExpression($sub[$i][0],$sub[$i][1]); //identify a subscription
        $upmlText[$countSection] .= "</subscription><br/>
    }
    $upmlText[$countSection] .= "</array><br/>
}
}
return;
}//end of findArrayElement

//find expressions of object clone or new
function identifyCloneNew($clonenewExArray)
{
    global $countFunction, $levelFunction, $contents, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
    global $braceArray;

    if($tokens[$clonenewExArray[0][0]][0] == T_CLONE )
function identifyPrimitive($start, $end) {
    //1. if the entire expression is inside a pair of braces, identify the inside part
    //2. a single item is an array expression array(...)
    //3. array association: $a => $b;
    //4. a single item is a function expression
    //5. object operator: a -> $b;
    //6. double colon: a::b;
    //7. static: T_STATIC
    //8. a variable
    //9. a literal number: T_DNUMBER, T_LNUMBER,
    //10. a string: parent, true, etc.
    //11. string constant: T_CONSTANT_ENCAPSED_STRING for "abc"
       //11.1 T_ENCAPSED_AND_WHITESPACE - "xxx $a yyy" has "xxx" and "yyy", $a is
//1. if the entire expression is inside a pair of braces, identify the inside part
//echo "start=".$start.";tokens[start]=".$tokens[start][0]."<br />
//echo "end=".$end.";tokens[end]=".$tokens[end][0]."<br />

if($tokens[$start]=="(" && $tokens[$end]==")")
    identifyExpression($start, $end); //identifyExpression() will remove "(" and ")" if needed

//2. a single item is an array expression array(...)
else if($tokens[$start][0]==T_ARRAY)
{
    $upmlText[$countSection] .= "<array_define>
    if($start <> $end -2)//no content in (), like $a = array();
        identifyExpression($start+1, $end);
    $upmlText[$countSection] .= "</array_define>

    //3. array association: $a => $b;
else if (dblArwExists($start, $end) <> false)
{
    $dblArrow = dblArwExists($start, $end);
    $upmlText[$countSection] .= "<array_item>
        identifyExpression($start, $dblArrow -1);
    $upmlText[$countSection] .= "</array_item>
    $upmlText[$countSection] .= "<array_item_value>
        identifyExpression($dblArrow + 1, $end);
    $upmlText[$countSection] .= "</array_item_value>

    //4. a single item is a function expression myFun(...)
else if($tokens[$start][0]==T_STRING && $tokens[$start+1]=="(")
{
    $upmlText[$countSection] .= "<function_expression><name>
        $upmlText[$countSection] .= $tokens[$start][1]."\n"
        $upmlText[$countSection] .= "</name>\n"
        $upmlText[$countSection] .= "<arguments>\n"
    identifyExpression($start+2, $end - 1);//further process the arguments of the
function

$upmlText[$countSection] .= "</arguments></function_expression>
"

//5. object operator: a -> $b;
else if (arrowExists($start, $end) <> false)
{
    $objOp = arrowExists($start, $end);
    $upmlText[$countSection] .= "<object_name>
";
    identifyExpression($start, $objOp-1);
    $upmlText[$countSection] .= "</object_name>
";
    $upmlText[$countSection] .= "<object_member>
";
    identifyExpression($objOp+1, $end);
    $upmlText[$countSection] .= "</object_member>
";
}

//6. double colon: a::b;
else if (dblCollonExists($start, $end)<> false)
{
    $dblCollon = dblCollonExists($start, $end);
    $upmlText[$countSection] .= "<owner>
"; //a class or other item that owns the
    identifyExpression($start, $dblCollon-1);
    $upmlText[$countSection] .= "</owner>
";
    $upmlText[$countSection] .= "<own>
";
    identifyExpression($dblCollon+1, $end);
    $upmlText[$countSection] .= "</own>
";
}

//7. a reference variable
else if ($tokens[$start] == "&" && $tokens[$end][0]==T_VARIABLE)
{
    $upmlText[$countSection] .= "<reference>
";
    $upmlText[$countSection] .= $tokens[$end][1];
    $upmlText[$countSection] .= "</reference>
";
//8. a variable
else if(($start == $end) && ($tokens[$start][0] == T_VARIABLE))
{
    $varScope = "";
    if($tokens[$start -1][0]==T_PUBLIC)$varScope="public";
    else if($tokens[$start -1][0]==T_PROTECTED)$varScope="protected";
    else if($tokens[$start -1][0]==T_PRIVATE)$varScope = "private";
    if($varScope<>"")
        $upmlText[$countSection] .= "<variable scope=""."$varScope.""/>\n"
    else
        $upmlText[$countSection] .= "<variable/>\n"
    $upmlText[$countSection] .= $tokens[$start][1];
    $upmlText[$countSection] .= "</variable>\n";
}

//9. a literal number: T_DNUMBER, T_LNUMBER,
else if(($start == $end) && ($tokens[$start][0]== T_DNUMBER || $tokens[$start][0]==
T_LNUMBER))
{
    $upmlText[$countSection] .= "<literal_value/>\n"
    $upmlText[$countSection] .= $tokens[$start][1];
    $upmlText[$countSection] .= "</literal_value>\n"
}

//10. a string: parent, true, label, etc.
else if(($start == $end) && ($tokens[$start][0]== T_STRING))
{
    //if($tokens[$start][1]=="true")
    //    $upmlText[$countSection] .= "<bool_true />";
    //else if($tokens[$start][1]=="false")
    //    $upmlText[$countSection] .= "<bool_false />";
    //else
    //    $upmlText[$countSection] .= "<label>$tokens[$start][1]."</label>";
    //    $upmlText[$countSection] .= $tokens[$start][1];
    }
//11. string constant: T_CONSTANT_ENCAPSED_STRING for "abc"
else if(($start == $end) && ($tokens[$start][0] == T_CONSTANT_ENCAPSED_STRING))
{
    $upmlText[$countSection] .= "<literal_string>\n";
    $upmlText[$countSection] .= $tokens[$start][1];
    $upmlText[$countSection] .= "</literal_string>\n";
}

//11.1 T_ENCAPSED_AND_WHITESPACE - "xxx $a yyy" has "xxx" and "yyy", $a is T_VARIABLE
else if($tokens[$start][0] == T_ENCAPSED_AND_WHITESPACE)
{
    $upmlText[$countSection] .= "<literal_string>\n";
    $upmlText[$countSection] .= $tokens[$start][1];
    $upmlText[$countSection] .= "<variable>\n";
    $upmlText[$countSection] .= $tokens[$start+1][1];
    $upmlText[$countSection] .= "</variable>\n";
    $upmlText[$countSection] .= $tokens[$start+2][1];
    $upmlText[$countSection] .= "</literal_string>\n";
}

return;
} //end identifyPrimitive($start, $end)

//looking for "=>"
function dblArwExists($start, $end)
{
    global $tokens;

    for($i=$start; $i<$end; $i++)
    {
        if($tokens[$i][0] == T_DOUBLE_ARROW) return $i;
    }
    return false;
}
//looking for "->", object operator
function arrowExists($start, $end) {
    global $tokens;
    for($i=$start; $i<$end; $i++) {
        if($tokens[$i][0] == T_OBJECT_OPERATOR) return $i;
    }
    return false;
}

//looking for "::", class member operator
function dblCollonExists($start, $end) {
    global $tokens;
    for($i=$start; $i<$end; $i++) {
        if($tokens[$i][0] == T_DOUBLE_COLON || $tokens[$i][0] == T_PAAMAYIM_NEKUDOTAYIM) return $i;
    }
    return false;
} //end of function dblCollonExists

//find pairs of "(" and ")"
function findBracePair($toks) {
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
    global $braceArray;
    $bracePair = array(); //a pair of positions of neighboring "(" and ")", including $a(..), $fun(..)
    $tempVar = 0;
    $lb = $tempVar;
    $rb = $lb;
    $exEnd = sizeof($toks);
while($tempVar < $exEnd)
{
    if($tokens[$tempVar] == "(")
    {
        $lb = $tempVar;
        $rb = findRightBracePair($lb, $exEnd); //find the right brace ")" to pair
        //echo "lb=\$lb. rb=\$rb."<br />
        if($rb > $lb && $rb <= $exEnd)
        {
            $bracePair[0]=$lb;
            $bracePair[1]=$rb;
            array_push($braceArray, $bracePair); //put the pair in the pair array
        }
    }
    $tempVar++;
} //end of while looking for pair of "(" and ")"
//1.1. identify whether a pair of "(" and ")" is for a function or a variable
// like $a(...) or fun(...) for($i = 0; $i<=sizeof($braceArray); $i++)
    if($tokens[$braceArray[$i][0]-1][0]==T_VARIABLE || $tokens[$braceArray[$i][0]-1][0]==T_STRING)
        $braceArray[$i][0]--;
    //the pair of array retrace one token back to contain the variable as its group
return $braceArray;
} //end of findBracePair($exStart, $exEnd, $brcArray)

//find ")" that match the found "(
function findRightBracePair($vSub, $exRightEnd)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray;
    $tempVar = $vSub+1; //from the next of "(" to start the search
    $countLeft = 1; //count "(" so that ")" matches the number of "("
    while($tempVar <= $exRightEnd)
    {

Page 226
if($tokens[$tempVar]=="(")$countLeft++;  
else if($tokens[$tempVar]==")")$countLeft--;  
if($countLeft == 0) return $tempVar;  
else $tempVar++;  
}  
return $vSub+1;  //no ")" found, return to next to the start search point.  
}  

//separate an expression into smaller expression (strings) by commas

//find pairs of "(" and ")"
function findBracketPair($toks)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;  
global $braceArray, $bracketArray;  
$bracketPair = array(); //a pair of positions of neighboring "(" and ")", including
    $a(..), fun(..)
    $tempVar = 0;  
$lb = $tempVar;  
$rb = $lb;  
$exEnd = sizeof($toks);  
while($tempVar < $exEnd)
    {
        if($tokens[$tempVar] == "[")
        {
            $lb = $tempVar;  
            $rb = findRightBracketPair($lb, $exEnd); //find the right brace ")" to pair the found "("  
            if($rb > $lb & & $rb <= $exEnd)  
            {
                $bracketPair[0] =$lb;  
                $bracketPair[1]=$rb;  
                array_push($bracketArray, $bracketPair); //put the pair in the pair array
            }
        }
    }
function findRightBracketPair($vSub, $exRightEnd)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
    global $braceArray, $bracketArray;
    $tempVar = $vSub+1; //from the next of "[" to start the search
    $countLeft = 1; //count "," so that "]" matches the number of "[
    while($tempVar <= $exRightEnd)
    {
        if($tokens[$tempVar]=="[")$countLeft++;
        else if($tokens[$tempVar]=="]")$countLeft--;
        if($countLeft == 0)return $tempVar;
        else $tempVar++;
    //echo "";
    }
    return $vSub+1; //no "]" found, return to next to the start search point.
}

function findDynamicBracePair($exStart, $exEnd)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass, $currentClassName, $countSection;
    global $braceArray, $bracketArray, $dBraceArray, $dBracketArray;
    $dBracePair = array(); //a pair of positions of neighboring "(" and ""); including
    $tempVar = $exStart; //a pair of positions of neighboring "(" and ""); including
    $lb = $tempVar;
    $rb = $lb;
    //findDynamicBracePair($exStart, $exEnd)
$countDBracePair = 0; //record found pairs of braces
//echo "in findDynamicBracePair: exStart=".$exStart." exEnd=".$exEnd."<br />
unset($dBraceArray);
$dBraceArray=array();
while($tempVar < $exEnd)
{
    if($tokens[$tempVar] == "(")
    {
        $lb = $tempVar;
        $rb = findRightBracePair($lb, $exEnd); //find the right brace ")" to pair
        //echo "lb=".$lb." rb=".$rb."<br />
        if($rb > $lb && $rb <= $exEnd)
        {
            $dBracePair[0]=$lb;
            $dBracePair[1]=$rb;
            array_push($dBraceArray, $dBracePair); //put the pair in the pair
        }
        $countDBracePair++;
    }
    //end of while looking for pair of "(" and ")"
}
$countDBracePair++; //end of while looking for pair of "(" and ")"
//1.1. identify whether a pair of "(" and ")" is for a function or a variable
    like $a(...) or fun(...) //echo "countDBracePair=".$countDBracePair."<br /"; //end of while looking for pair of "(" and ")"
if($countDBracePair==0)//no pair of braces
{
    $dBracePair[0]=0;
    $dBracePair[1]=0;
    array_push($dBraceArray, $dBracePair);
}
else
{
    for($i = 0; $i<=sizeof($dBraceArray); $i++)
    {
        if($tokens[$dBraceArray[$i][0]-1][0]==T_VARIABLE ||
function findDynamicBracketPair($exStart, $exEnd)
{
    global $countFunction, $levelFunction, $tokens, $upmlText, $countClass, $levelClass,
    $currentClassName, $countSection;
    global $braceArray, $bracketArray, $dBraceArray, $dBracketArray;
    $dBracketPair = array(); // a pair of positions of neighboring "(" and ")", including $a(..), fun(..)
    $tempVar = $exStart;
    $lb = $tempVar;
    $rb = $lb;
    while($tempVar < $exEnd)
    {
        if($tokens[$tempVar] == "[")
        {
            $lb = $tempVar;
            $rb = findRightBracketPair($lb, $exEnd); // find the right brace ")" to pair the found "("
            // echo "lb=".$lb." rb=".$rb."<br />");
            if($rb > $lb && $rb <= $exEnd)
            {
                $dBracketPair[0]=$lb;
                $dBracketPair[1]=$rb;
                array_push($dBracketArray, $dBracketPair); // put the pair in the pair array
            }
        }
        $tempVar++;
    } // end of while looking for pair of "(" and ")"
}
//1.1. identify whether a pair of "(" and ")" is for a function or a variable
// like $a(...) or fun(...)
for($i = 0; $i <= sizeof($dBracketArray); $i++)
if($tokens[$dBracketArray[$i][0]-1][0]==T_VARIABLE || $tokens[$dBracketArray[$i][0]-1][0]==T_STRING)
    $dBracketArray[$i][0]--;
//the pair of array retrace one token back to contain the variable as its group
return $dBracketArray;
}//end of findDynamicBracketPair

//identifyBodyStart
function identifyBodyStart($sub)
{
    global $tokens, $countSection;
    $tempVar = $sub;
    do
    {
        $tempVar++;
    }while($tokens[$tempVar]<>"{");
    return $tempVar;
}

//detect if a location is between a pair of braces ("(" and ")") or not
function inBraces($loc,$locStart, $locEnd, $brcArray)
{
    for($i = 0; $i <= sizeof($brcArray); $i++)
    {
        if(($loc > $brcArray[$i][0])&& ($loc < $brcArray[$i][1]))
        {//&&($brcArray[$i][0]>=$locStart)&&(brcArray[$i][1]<=$locEnd))
            return true;
        }
    return false;
}

//detect if a location is between a pair of brackets ("[" and "]") or not
function inBrackets($loc,$locStart, $locEnd, $brckArray)
{
    
Page 231
for($i = 0; $i <= sizeof($brckArray); $i++)
{
    if((($loc > $brckArray[$i][0]) && ($loc < $brckArray[$i][1]))
        //&&($brcArray[$i][0]>$locStart)&&(brcArray[$i][1]<=$locEnd))
        return true;
    }
    return false;
}

//find return data type
function findReturnType($funStart, $funEnd)
{
    global $tokens;
    for($i=$funStart; $i<$funEnd; $i++)
    {
        if($tokens[$i][0]==T_RETURN)//find keyword return
        {
            if($tokens[$i+1]==";") return null;
            else
            {
                $dataType = gettype($tokens[$i+1][1]);
                return $dataType;
            }
        }
        //this can be problematic
    }
    return null; //not find keyword return
}

//make a string from tokens elements
function makeString($startLoc, $endLoc)
{
    global $tokens;
    $myStr = "";
    for($i=$startLoc; $i<=$endLoc; $i++)
{  
    if (is_string($tokens[$i])) $myStr .= $tokens[$i];  
    else $myStr .= $tokens[$i][1];  
}  

return $myStr;
}

//identify the position of ";", the current statement

//identify a statement that starts with a variable
function identifyStateStartVar($stateStartSub, $stateEndSub)  
{  
    echo "identifyStateStartVar\n";
}

function printArr($a)  
{  
    echo "<br />";
    foreach($a as $aele)  
    {  
        $i = 0;
        while ($i < sizeof($aele))  
        {  
            echo $aele[$i]."<br />";  
            $i++;
        }
        echo "<br />";
    }
    echo "<br />";
    return;
}

function printStr($a)  
{  
    echo "<br />";
    foreach($a as $aele)  
    {  
        echo "$aele[$i]."<br />";
    }  

/$i = 0;
  if(is_string($aele)) echo $aele;
  else echo $aele[1];
}

 echo "<br />";
 return;
}

//*****End of identifying PHP*****

//*****Start of identifying JavaScript*****
function identifyJS($uploadFile)
{
  echo "this is function identifyJS()";
  return;
}
//*****End of identifying JavaScript*****

//*****Start of identifying Java*****
function identifyJava($uploadFile)
{
  echo "this is function identifyJava()";
  return;
}
//*****End of identifying Java*****

//*****Start of identifying C*****
function identifyC($uploadFile)
{
  echo "this is function identifyC()";
  return;
}
//*****End of identifying C*****

//*****Start of identifying C+++*****
function identifyCPP($uploadFile)
{    echo "this is function identifyCPP()";
    return;
}  
//*****End of identifying C++***** 

//*****Start of identifying Python*****  
function identifyPy($uploadFile)
{
    echo "this is function identifyPy()";
    return;
}
//*****End of identifying Python***** 

?>
</body>
</html>
## Appendix C. PHP Tokens (from php.net)

<table>
<thead>
<tr>
<th>Token</th>
<th>Syntax</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_ABSTRACT</td>
<td>abstract</td>
<td><a href="#">Class Abstraction</a> (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_AND_EQUAL</td>
<td>&amp;=</td>
<td><a href="#">assignment operators</a></td>
</tr>
<tr>
<td>T_ARRAY</td>
<td>array()</td>
<td><a href="#">array(), array syntax</a></td>
</tr>
<tr>
<td>T_ARRAY_CAST</td>
<td>(array)</td>
<td><a href="#">type-casting</a></td>
</tr>
<tr>
<td>T_AS</td>
<td>as</td>
<td><a href="#">foreach</a></td>
</tr>
<tr>
<td>T_BAD_CHARACTER</td>
<td></td>
<td>anything below ASCII 32 except \t (0x09), \n (0x0a) and \r (0x0d)</td>
</tr>
<tr>
<td>T_BOOLEAN_AND</td>
<td>&amp;&amp;</td>
<td><a href="#">logical operators</a></td>
</tr>
<tr>
<td>T_BOOLEAN_OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_BOOL_CAST</td>
<td>(bool) or (boolean)</td>
<td><a href="#">type-casting</a></td>
</tr>
<tr>
<td>T_BREAK</td>
<td>break</td>
<td><a href="#">break</a></td>
</tr>
<tr>
<td>T_CALLABLE</td>
<td>callable</td>
<td><a href="#">callable</a></td>
</tr>
<tr>
<td>T_CASE</td>
<td>case</td>
<td><a href="#">switch</a></td>
</tr>
<tr>
<td>T_CATCH</td>
<td>catch</td>
<td><a href="#">Exceptions</a> (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>Token</td>
<td>Syntax</td>
<td>Reference</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>T_CHARACTER</td>
<td></td>
<td>not used anymore</td>
</tr>
<tr>
<td>T_CLASS</td>
<td>class</td>
<td>classes and objects</td>
</tr>
<tr>
<td>T_CLASS_C</td>
<td><strong>CLASS</strong></td>
<td>magic constants (available since PHP 4.3.0)</td>
</tr>
<tr>
<td>T.Clone</td>
<td>clone</td>
<td>classes and objects (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_CLOSE_TAG</td>
<td>?&gt; or %&gt;</td>
<td>escaping from HTML</td>
</tr>
<tr>
<td>T_COMMENT</td>
<td>// or #, and /* */ in PHP 5</td>
<td>comments</td>
</tr>
<tr>
<td>T_CONCAT_EQUAL</td>
<td>.=</td>
<td>assignment operators</td>
</tr>
<tr>
<td>T_CONST</td>
<td>const</td>
<td>class constants</td>
</tr>
<tr>
<td>T_CONSTANT_ENCAPSED_STRING</td>
<td>&quot;foo&quot; or 'bar'</td>
<td>string syntax</td>
</tr>
<tr>
<td>T_CONTINUE</td>
<td>continue</td>
<td>continue</td>
</tr>
<tr>
<td>T CURLY_OPEN</td>
<td>{$</td>
<td>complex variable parsed syntax</td>
</tr>
<tr>
<td>T_DEC</td>
<td>--</td>
<td>incrementing/decrementing operators</td>
</tr>
<tr>
<td>T_DECLARE</td>
<td>declare</td>
<td>declare</td>
</tr>
<tr>
<td>Token</td>
<td>Syntax</td>
<td>Reference</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>T_DEFAULT</td>
<td>default</td>
<td>switch</td>
</tr>
<tr>
<td>T_DIR</td>
<td><strong>DIR</strong></td>
<td><a href="https://www.php.net/manual/en/language.types.constants.php#120915">magic constants</a> (available since PHP 5.3.0)</td>
</tr>
<tr>
<td>T_DIV_EQUAL</td>
<td>/=</td>
<td><a href="https://www.php.net/manual/en/language.operators.assignment.php">assignment operators</a></td>
</tr>
<tr>
<td>T_DNUMBER</td>
<td>0.12, etc</td>
<td><a href="https://www.php.net/manual/en/language.types.float.php">floating point numbers</a></td>
</tr>
<tr>
<td>T_DOC_COMMENT</td>
<td>/** */</td>
<td><a href="https://www.php.net/manual/en/function.phpdoc.php">PHPDoc style comments</a>(available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_DO</td>
<td>do</td>
<td>do..while</td>
</tr>
<tr>
<td>T_DOUBLE_CAST</td>
<td>(real), (double) or (float)</td>
<td><a href="https://www.php.net/manual/en/language.types.casting.php">type-casting</a></td>
</tr>
<tr>
<td>T_DOUBLE_COLON</td>
<td>::</td>
<td>see T_PAAMAYIM_NEKUDOTAYIMbelow</td>
</tr>
<tr>
<td>T_ECHO</td>
<td>echo</td>
<td>echo</td>
</tr>
<tr>
<td>T_ELSE</td>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>T_ELSEIF</td>
<td>elseif</td>
<td>elseif</td>
</tr>
<tr>
<td>Token</td>
<td>Syntax</td>
<td>Reference</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>T_EMPTY</td>
<td>empty</td>
<td>empty()</td>
</tr>
<tr>
<td>T_ENCAPSED_AND_WHITESPACE</td>
<td>&quot; $a&quot;</td>
<td>constant part of string with variables</td>
</tr>
<tr>
<td>T_ENDDOUBLE</td>
<td>enddeclare</td>
<td>declare, alternative syntax</td>
</tr>
<tr>
<td>T_ENDFOR</td>
<td>endfor</td>
<td>for, alternative syntax</td>
</tr>
<tr>
<td>T_ENDFOREACH</td>
<td>endforeach</td>
<td>foreach, alternative syntax</td>
</tr>
<tr>
<td>T_ENDIF</td>
<td>endif</td>
<td>if, alternative syntax</td>
</tr>
<tr>
<td>T_ENDSWITCH</td>
<td>endswitch</td>
<td>switch, alternative syntax</td>
</tr>
<tr>
<td>T_ENDWHILE</td>
<td>endwhile</td>
<td>while, alternative syntax</td>
</tr>
<tr>
<td>T_END_HEREDOC</td>
<td></td>
<td>heredoc syntax</td>
</tr>
<tr>
<td>T_EVAL</td>
<td>eval()</td>
<td>eval()</td>
</tr>
<tr>
<td>T_EXIT</td>
<td>exit or die</td>
<td>exit(), die()</td>
</tr>
<tr>
<td>T_EXTENDS</td>
<td>extends</td>
<td>extends, classes and objects</td>
</tr>
<tr>
<td>T_FILE</td>
<td><strong>FILE</strong></td>
<td>magic constants</td>
</tr>
<tr>
<td>T_FINAL</td>
<td>final</td>
<td>Final Keyword (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>Token</td>
<td>Syntax</td>
<td>Reference</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>T_FINALLY</td>
<td>finally</td>
<td>Exceptions (available since PHP 5.5.0)</td>
</tr>
<tr>
<td>T_FOR</td>
<td>for</td>
<td>for</td>
</tr>
<tr>
<td>T_FOREACH</td>
<td>foreach</td>
<td>foreach</td>
</tr>
<tr>
<td>T_FUNCTION</td>
<td>function or cfunction</td>
<td>functions</td>
</tr>
<tr>
<td>T_FUNC_C</td>
<td><strong>FUNCTION</strong></td>
<td>magic constants (available since PHP 4.3.0)</td>
</tr>
<tr>
<td>T_GLOBAL</td>
<td>global</td>
<td>variable scope</td>
</tr>
<tr>
<td>T_GOTO</td>
<td>goto</td>
<td>(available since PHP 5.3.0)</td>
</tr>
<tr>
<td>T_HALT_COMPILER</td>
<td>__halt_compiler()</td>
<td>__halt_compiler (available since PHP 5.1.0)</td>
</tr>
<tr>
<td>T_IF</td>
<td>if</td>
<td>if</td>
</tr>
<tr>
<td>T_IMPLMENTS</td>
<td>implements</td>
<td>Object Interfaces (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_INC</td>
<td>++</td>
<td>incrementing/decrementing operators</td>
</tr>
<tr>
<td>T_INCLUDE</td>
<td>include()</td>
<td>include</td>
</tr>
<tr>
<td>T_INCLUDE_ONCE</td>
<td>include_once()</td>
<td>include_once</td>
</tr>
<tr>
<td>T_INLINE_HTML</td>
<td></td>
<td>text outside PHP</td>
</tr>
<tr>
<td>Token</td>
<td>Syntax</td>
<td>Reference</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>T_INSTANCEOF</td>
<td>instanceof</td>
<td><a href="#">type operators</a> (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_INSTEADOF</td>
<td>insteadof</td>
<td><a href="#">Traits</a> (available since PHP 5.4.0)</td>
</tr>
<tr>
<td>T_INT_CAST</td>
<td>(int) or (integer)</td>
<td><a href="#">type-casting</a></td>
</tr>
<tr>
<td>T_INTERFACE</td>
<td>interface</td>
<td><a href="#">Object Interfaces</a> (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_ISSET</td>
<td>isset()</td>
<td>isset()</td>
</tr>
<tr>
<td>T_IS_EQUAL</td>
<td>==</td>
<td><a href="#">comparison operators</a></td>
</tr>
<tr>
<td>T_IS_GREATER_OR_EQUAL</td>
<td>&gt;=</td>
<td><a href="#">comparison operators</a></td>
</tr>
<tr>
<td>T_IS_IDENTICAL</td>
<td>===</td>
<td><a href="#">comparison operators</a></td>
</tr>
<tr>
<td>T_IS_NOT_EQUAL</td>
<td>!= or &lt;&gt;</td>
<td><a href="#">comparison operators</a></td>
</tr>
<tr>
<td>T_IS_NOT_IDENTICAL</td>
<td>!==</td>
<td><a href="#">comparison operators</a></td>
</tr>
<tr>
<td>T_IS_SMALLER_OR_EQUAL</td>
<td>&lt;=</td>
<td><a href="#">comparison operators</a></td>
</tr>
<tr>
<td>T_LINE</td>
<td><strong>LINE</strong></td>
<td><a href="#">magic constants</a></td>
</tr>
<tr>
<td>T_LIST</td>
<td>list()</td>
<td>list()</td>
</tr>
<tr>
<td>T_LNUMBER</td>
<td>123, 012, 0x1ac, etc</td>
<td><a href="#">integers</a></td>
</tr>
<tr>
<td>Token</td>
<td>Syntax</td>
<td>Reference</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>T_LOGICAL_AND</td>
<td>and</td>
<td>logical operators</td>
</tr>
<tr>
<td>T_LOGICAL_OR</td>
<td>or</td>
<td>logical operators</td>
</tr>
<tr>
<td>T_LOGICAL_XOR</td>
<td>xor</td>
<td>logical operators</td>
</tr>
<tr>
<td>T_METHOD_C</td>
<td><strong>METHOD</strong></td>
<td>magic constants (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_MINUS_EQUAL</td>
<td>-=</td>
<td>assignment operators</td>
</tr>
<tr>
<td>T_ML_COMMENT</td>
<td>/* and */</td>
<td>comments (PHP 4 only)</td>
</tr>
<tr>
<td>T_MOD_EQUAL</td>
<td>%=</td>
<td>assignment operators</td>
</tr>
<tr>
<td>T_MUL_EQUAL</td>
<td>*=</td>
<td>assignment operators</td>
</tr>
<tr>
<td>T_NAMESPACE</td>
<td>namespace</td>
<td>namespaces (available since PHP 5.3.0)</td>
</tr>
<tr>
<td>T_NS_C</td>
<td><strong>NAMESPACE</strong></td>
<td>namespaces (available since PHP 5.3.0)</td>
</tr>
<tr>
<td>T_NS_SEPARATOR</td>
<td>\</td>
<td>namespaces (available since PHP 5.3.0)</td>
</tr>
<tr>
<td>T_NEW</td>
<td>new</td>
<td>classes and objects</td>
</tr>
<tr>
<td>T_NUM_STRING</td>
<td>&quot;$a[0]&quot;</td>
<td>numeric array index inside string</td>
</tr>
<tr>
<td>T_OBJECT_CAST</td>
<td>(object)</td>
<td>type-casting</td>
</tr>
<tr>
<td>Token</td>
<td>Syntax</td>
<td>Reference</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>T_OBJECT_OPERATOR</td>
<td>-&gt;</td>
<td>classes and objects</td>
</tr>
<tr>
<td>T_OLD_FUNCTION</td>
<td>old_function</td>
<td>(PHP 4 only)</td>
</tr>
<tr>
<td>T_OPEN_TAG</td>
<td>&lt;?php, &lt;? or &lt;%</td>
<td>escaping from HTML</td>
</tr>
<tr>
<td>T_OPEN_TAG_WITH_ECHO</td>
<td>&lt;?= or &lt; %=</td>
<td>escaping from HTML</td>
</tr>
<tr>
<td>T_OR_EQUAL</td>
<td></td>
<td>=</td>
</tr>
<tr>
<td>T_PAAMAYIM_NEKUDOTAYIM</td>
<td>::</td>
<td>::. Also defined as T_DOUBLE_COLON.</td>
</tr>
<tr>
<td>T_PLUS_EQUAL</td>
<td>+=</td>
<td>assignment operators</td>
</tr>
<tr>
<td>T_PRINT</td>
<td>print()</td>
<td>print</td>
</tr>
<tr>
<td>T_PRIVATE</td>
<td>private</td>
<td>classes and objects (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_PUBLIC</td>
<td>public</td>
<td>classes and objects (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_PROTECTED</td>
<td>protected</td>
<td>classes and objects (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_REQUIRE</td>
<td>require()</td>
<td>require</td>
</tr>
<tr>
<td>T_REQUIRE_ONCE</td>
<td>require_once()</td>
<td>require_once</td>
</tr>
<tr>
<td>T_RETURN</td>
<td>return</td>
<td>returning values</td>
</tr>
<tr>
<td>Token</td>
<td>Syntax</td>
<td>Reference</td>
</tr>
<tr>
<td>------------------</td>
<td>--------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>T_SL</td>
<td>&lt;&lt;</td>
<td>bitwise operators</td>
</tr>
<tr>
<td>T_SL_EQUAL</td>
<td>&lt;&lt;=</td>
<td>assignment operators</td>
</tr>
<tr>
<td>T_SR</td>
<td>&gt;&gt;</td>
<td>bitwise operators</td>
</tr>
<tr>
<td>T_SR_EQUAL</td>
<td>&gt;&gt;=</td>
<td>assignment operators</td>
</tr>
<tr>
<td>T_START_HEREDOC</td>
<td>&lt;&lt;&lt;</td>
<td>heredoc syntax</td>
</tr>
<tr>
<td>T_STATIC</td>
<td>static</td>
<td>variable scope</td>
</tr>
<tr>
<td>T_STRING</td>
<td>parent, true etc.</td>
<td>identifiers, e.g. keywords like parent and self, function names, class names and more are matched. See also T_CONSTANT_ENCAPSED_STRING.</td>
</tr>
<tr>
<td>T_STRING_CAST</td>
<td>(string)</td>
<td>type-casting</td>
</tr>
<tr>
<td>T_STRING_VARNAME</td>
<td>&quot;${a</td>
<td>complex variable parsed syntax</td>
</tr>
<tr>
<td>T_SWITCH</td>
<td>switch</td>
<td>switch</td>
</tr>
<tr>
<td>T_THROW</td>
<td>throw</td>
<td>Exceptions (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_TRAIT</td>
<td>trait</td>
<td>Traits (available since PHP 5.4.0)</td>
</tr>
<tr>
<td>T_TRAIT_C</td>
<td><strong>TRAIT</strong></td>
<td><strong>TRAIT</strong> (available since PHP 5.4.0)</td>
</tr>
<tr>
<td>Token</td>
<td>Syntax</td>
<td>Reference</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>T_TRY</td>
<td>try</td>
<td>Exceptions (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_UNSET</td>
<td>unset()</td>
<td>unset()</td>
</tr>
<tr>
<td>T_UNSET_CAST</td>
<td>(unset)</td>
<td>type-casting (available since PHP 5.0.0)</td>
</tr>
<tr>
<td>T_USE</td>
<td>use</td>
<td>namespaces (available since PHP 5.3.0; reserved since PHP 4.0.0)</td>
</tr>
<tr>
<td>T_VAR</td>
<td>var</td>
<td>classes and objects</td>
</tr>
<tr>
<td>T_VARIABLE</td>
<td>$foo</td>
<td>variables</td>
</tr>
<tr>
<td>T_WHILE</td>
<td>while</td>
<td>while, do..while</td>
</tr>
<tr>
<td>T_WHITESPACE</td>
<td>\t \r\n</td>
<td></td>
</tr>
<tr>
<td>T_XOR_EQUAL</td>
<td>^=</td>
<td>assignment operators</td>
</tr>
<tr>
<td>T_YIELD</td>
<td>yield</td>
<td>generators (available since PHP 5.5.0)</td>
</tr>
</tbody>
</table>