УДК 621.9.048.7

ИССЛЕДОВАНИЕ КОМПИЛЯТОРОВ И СТАТИЧЕСКИЙ АНАЛИЗ КОДА

Михаленя А.Н.

ООО «ТУТ БАЙ МЕДИА», Минск, Беларусь, mikhalenia.a@gmail.com

Объектом исследования данной работы является веб - ориентированный язык PHP версии 5.3. Предметом исследования является семантика и синтаксис этого языка. Целью данной работы является разработка программного обеспечения статического анализа PHP кода. В данной статье рассмотрены основные этапы трансляции программы, формальное представление грамматик и программы генераторы сканера и парсера на примере языка PHP.

Для начала следует разобраться, что такое статический анализ и зачем он нужен. Статический анализ кода – анализ программного обеспечения, производимый (в отличие от динамического анализа) без реального выполнения исследуемых программ. В большинстве случаев анализ производится над какой-либо версией исходного кода, хотя иногда анализу подвергается какой-нибудь вид объектного кода, например P-код или код на MSIL. Термин обычно применяют к анализу, производимому специальным ПО. В зависимости от используемого инструмента глубина анализа может варьироваться от определения поведения отдельных операторов до анализа, включающего весь имеющийся исходный код. Способы использования полученной в ходе анализа информации также различны – от выявления мест, возможно содержащих ошибки, до формальных методов, позволяющих математически доказать какие-либо свойства программы (например, соответствие поведения спецификации). Некоторые люди считают программные метрики и обратное проектирование формами статического анализа. Получение метрик и статический анализ часто совмещаются, особенно при создании встраиваемых систем. В последнее время статический анализ все больше используется в верификации свойств ПО, используемого в компьютерных системах высокой надежности, особенно критичных для жизни. Также применяется для поиска кода, потенциально содержащего уязвимости.

Более систематично – анализ кода это возможность программы прочитать код анализируемой программы в какой-либо форме, «понять» его и выдать какую-то информацию. Соответственно, практически все анализаторы кода можно представить себе как поиск в определенном представлении программы (возможно с преобразованиями) определенных паттернов и дальнейший подробный анализ найденных участков.

Статический анализ постоянно применяется в следующих областях:

– ПО для медицинских устройств;

– ПО для ядерных станций и систем защиты реактора;

– ПО для авиации (в комбинации с динамическим анализом).

Большинство компиляторов (например, GNU C Compiler) выводят на экран «предупреждения» (warnings) – сообщения о том, что код, будучи синтаксически правильным, скорее всего, содержит ошибку. Например:

```
int x;
int y=x+2; // Переменная x не инициализирована
```

Это простейший статический анализ. У компилятора есть много других немаловажных характеристик – в первую очередь скорость работы и качество машинного кода, поэтому компиляторы проверяют код лишь на очевидные ошибки. Статические анализаторы предназначены для более детального исследования кода.

Типы ошибок, обнаруживаемых статическими анализаторами:

– Неопределенное поведение – неинициализированные переменные, обращение к NULL-указателям. О простейших случаях сигнализируют и компиляторы.

– Нарушение блок-схемы пользования библиотекой. Например, для каждого fopen ну-

жен fclose. И если файловая переменная теряется раньше, чем файл закрывается, анализатор может сообщить об ошибке.

– Типичные сценарии, приводящие к недокументированному поведению. Стандартная библиотека языка Си известна большим количеством неудачных технических решений. Некоторые функции, например, gets, в принципе небезопасны. sprintf и strcpy безопасны лишь при определенных условиях.

– Переполнение буфера – когда компьютерная программа записывает данные за пределами выделенного в памяти буфера.

```
void doSomething(const char* x)
{
   char s[40];
   sprintf(s, "[%s]", x);    // sprintf в локальный буфер, возможно переполнение
   ....
}
```

– Типичные сценарии, мешающие кроссплатформенности.

```
Object *p = getObject();
int pNum = reinterpret_cast<int>(p);    // на x86-32 верно, на x64 часть указателя будет потеряна; нужен size_t
```

Ошибки в повторяющемся коде. Многие программы исполняют несколько раз одно и то же с разными аргументами. Обычно повторяющиеся фрагменты не пишут с нуля, а размножают и исправляют.

```
dest.x = src.x + dx;
dest.y = src.y + dx;  // Ошибка, надо dy!
```

– Ошибки форматных строк – в функциях наподобие printf могут быть ошибки с несоответствием форматной строки реальному типу параметров.

```
std::wstring s;
printf ("s is %s", s);
```

Сами по себе трансляторы очень сложные программы, поэтому нельзя рассматривать процесс трансляции в один этап. Существует аналитическая фаза – в которой программа анализирует синтаксическую и семантическую ограничения источника. За этим следует синтетическая фаза – генерируется объектный код целевого языка. Компоненты транслятора можно разделить на 2 части -Front end и Back end, причем Front end не зависит от целевой машины, а Back end очень сильно от нее зависит.

Основные компоненты транслятора изображены на рисунке 1.

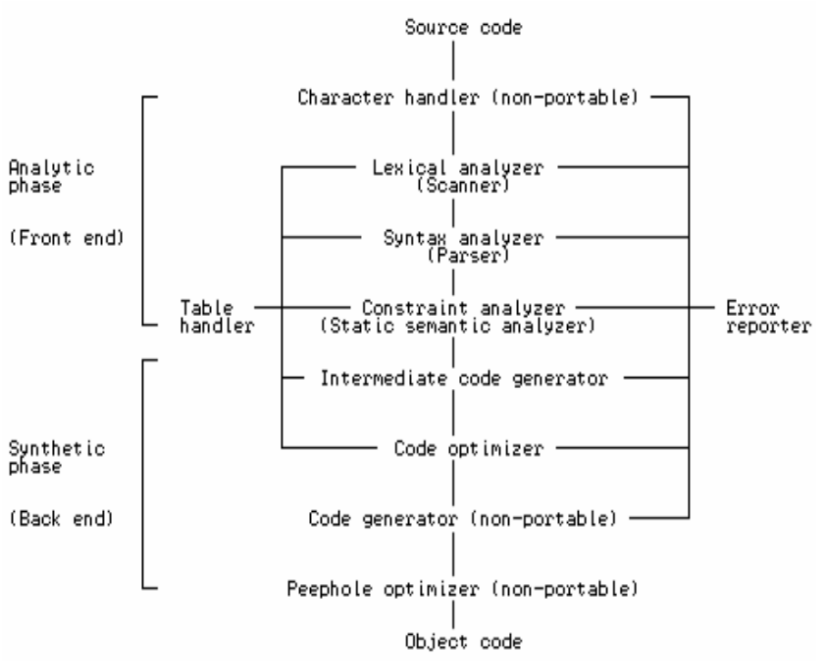

Рисунок 1 – Структура и фазы транслятора

Набор символов для обработки варьируется от операционной системы. Лексический анализатор или сканер делит исходный текст программы на группы, которые представляют логические лексемы языка, такие как строки, константы, ключевые слова, такие как while if, операторы - <= и так далее.

Некоторые из них получаются очень просто на выходе из сканера, некоторые должны быть связаны с различными свойствами, такими как значения. Лексический анализ происходит иногда очень просто, а иногда нет.

Разумно думать, что данное выражение синтаксически корректно, однако ни одна программа действительно не имеет смысла до ее динамического выполнения.

Семантика – это термин, который используется для описания смысла. Анализатор часто называют статический семантический анализатор или просто – семантический анализатор. Результат синтаксического и семантического анализа часто удобно представлять в виде абстрактного синтаксического дерева – AST.

Это очень удобно, так как позволяет оптимизировать некоторые участки на более позднем этапе генерации объектного кода, так же язык может иметь очень много ключевых слов, однако AST показывает только ключевые и значимые компоненты, отражающие смысл программы.

AST-дерево лишено семантических подробностей, важно отобразить суть.

Семантический анализатор имеет задачу идентифицироваться тип и другую контекстную информацию для различных узлов [1].

Разберем выражение:

WHILE (1 < P) AND (P < 9) DO P := P + Q END

Получившееся AST-дерево изображено на рисунке 2.

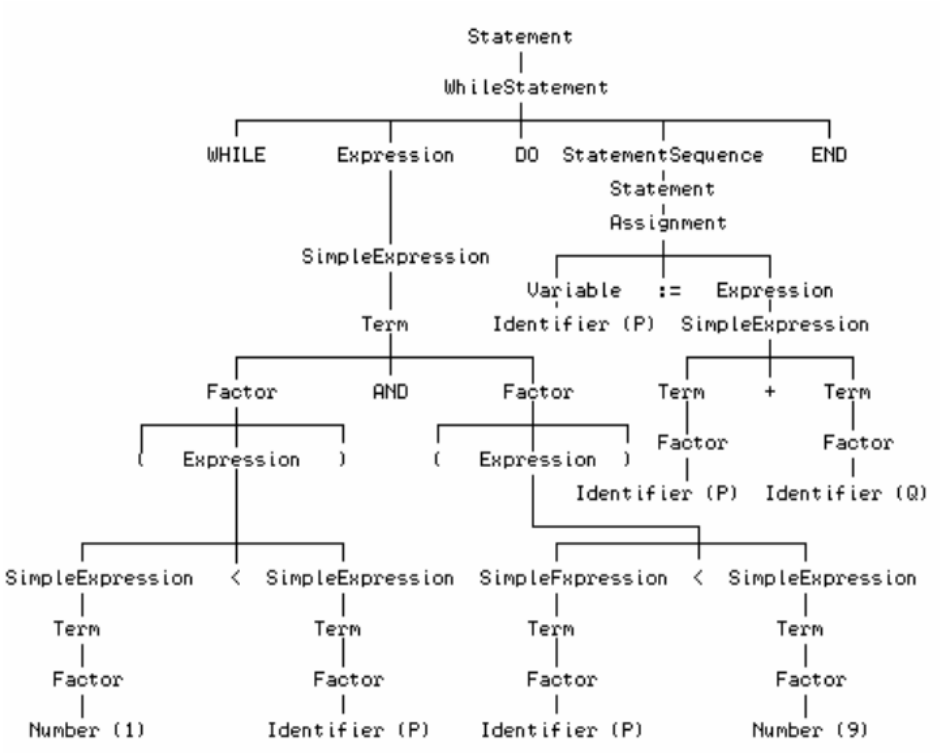

Рисунок 2 – AST-дерево

Фазы, которые были перечислены выше – носят аналитический характер. Так же есть те, которые носят более синтетический характер. Первой из них может быть промежуточная генерация кода. В некоторых случаях, эта фаза частично выполняется на более ранних этапах, или вовсе отсутствуют в простых трансляторах. Здесь используется структура, полученная на ранних этапах для создания формы кода, в виде скелета или набора макросов или ассемблер или даже высоко-уровневый код для обработки внешним компилятором. Основное отличие данного промежуточного кода от фактического машинного это то, что промежуточный код не должен описывать подробно такие вещи, как точная машина, например точные адреса и так далее [1].

Следующим этапом может быть оптимизация кода. Транслятор может быть снабжен этим механизмом в попытке улучшить промежуточный код в интересах производительности. Например, найти зоны, которые никогда не используются и удалить их. Однако это не безопасно.

Наиболее важным этапом в Back end – является обязательная генерация объектного кода. Эта фаза принимает входные данные от предыдущих фаз и производит объектный код, выделяя память для данных, выбор регистров для расчета промежуточных результатов, индексации и т. д. Этот этап требует отдельного пристального внимания [1].

Как известно, пользователи склонны допускать множество ошибок. Поэтому важно предусмотреть систему обработки ошибок, особенно на ранних этапах. Обнаружение ошибок во время компиляции в исходном коде не следует путать с обнаружением ошибок во время выполнения объектного кода.

Общая структура анализатора представлена на рисунке 3.

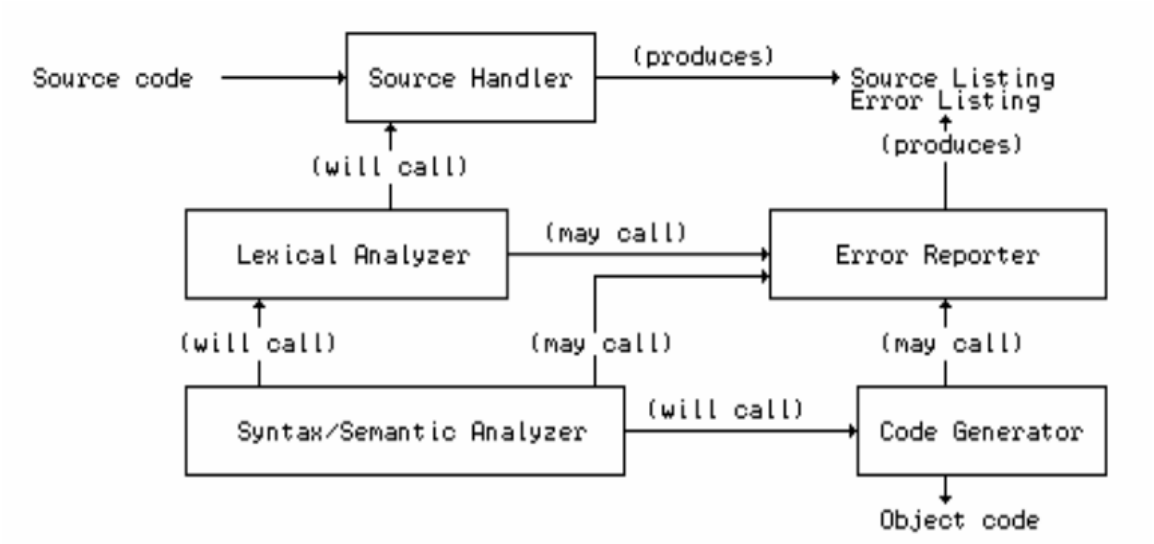

Рисунок 3 – Структура анализатора

Существует множество способов указания правил, однако для описания более реалистичных грамматик, таких, которые могут быть использованы в определении языков программирования наиболее распространенным способом для описания правил на протяжении многих лет являлась форма Бэкуса-Наура (англ. Backus-Normal-Form) – BNF. Впервые эта форма была использована в презентации языка Algol 60.

В классическом BNF, не-терминалу обычно дается подробное имя, и написано в угловые скобки, чтобы отличить его от терминального символа. Правила описываются так: leftside->definition

Здесь «->» можно интерпретировать как «определяется как» или «означает». В таких формах, как эта, левая сторона и определение состоит из строки, сцепленной с одним или более терминалами и нетерминалами. Для обозначения множества альтернатива используется символ «|»

Рассмотрим пример небольшого подмножества Английского языка:

G = {N , T , S , P}

N = {<sentence> , <qualified noun> , <noun> , <pronoun> , <verb> , <adjective> }

T = { the , man , girl , boy , lecturer , he , she , drinks , sleeps , mystifies , tall , thin , thirsty }

S = <sentence>

P = { <sentence> → the <qualified noun> <verb> (1)
| <pronoun> <verb> (2)
<qualified noun> → <adjective> <noun> (3)
<noun> → man | girl | boy | lecturer (4, 5, 6, 7)
<pronoun> → he | she (8, 9)
<verb> → talks | listens | mystifies (10, 11, 12)
<adjective> → tall | thin | sleepy (13, 14, 15)
}

Множество правил определяет нетерминал <sentence> как состоящий из любого терминала «the», а затем <qualified noun> перед <verb>, или <pronoun> за которым следует <verb>.<qualified noun> представляет <adjective> последующим <noun> и <noun> является одним из терминальных символов «man» или «girl» или «boy» или "lecturer".<pronoun> - нетерминал «he» или «she», а <verb> это либо «talks» или «listens» или «mystifies». Здесь <sentence>, <noun>, <qualified noun>, <pronoun>, <adjective> и <verb> являются нетерминалы. Они не появляются в любом предложении языка.

Из грамматики, один нетерминал выделяется в качестве так называемой цели или стар-

тового символа. Если мы хотим, сгенерировать произвольное предложение мы начинаем с символа цели и последовательно заменяем каждый нетерминал основываясь на правилах, определяющие, что этот нетерминал означает, пока все нетерминалы не будут удалены. В приведенном выше примере символ <sentence> можно обозначить как символ цели. Так, например, мы могли бы начать с <sentence> и от этого вывести сентенциальную форму:

the <qualified noun> <verb>.

С точки зрения определений в последнем разделе мы говорим, что <sentence> непосредственно определяет «the <qualified noun> <verb>». Если теперь применить правило 3 (<qualified noun> → <adjective> <noun>) получим сентенциальную форму:

the <adjective> <noun> <verb>

С точки зрения определений в последнем разделе, «the <qualified noun> <verb>» непосредственно определяет «the <adjective> <noun> <verb>», в то время как <sentence> определил эту сентенциальную форму нетривиальным путем. Если теперь следовать этому, применяя правило 14 (<adjective> →thin) мы получаем форму:

the thin <noun> <verb>

Применяем правило 10(<verb> → talks) получаем:

the thin <noun> talks

И наконец, применяем правило 6(<noun> → boy) получаем предложение: the thin boy talks.

Конечный результат удобно представлять в виде структурированного дерева разбора или дерево разбора, как на рисунке 4.

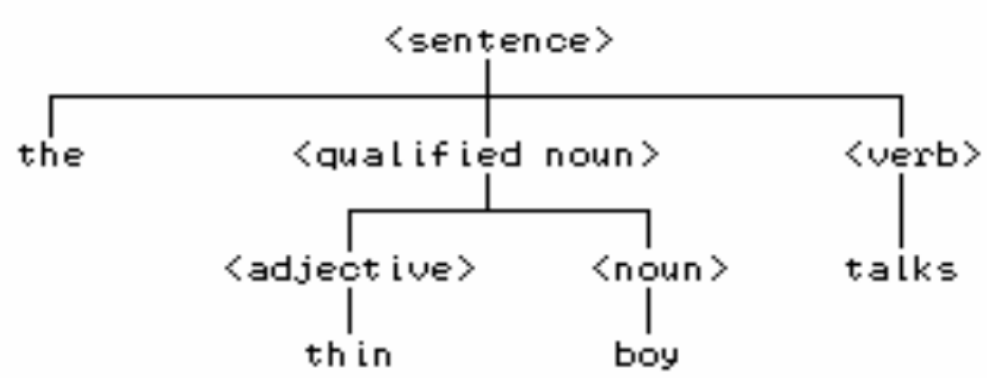

Рисунок 4 – Дерево разбора для предложения «the thin boy talks»

В этом представлении порядок, в котором были использованы правила не очевидны, однако становится ясно, почему выражаются «терминал» и «нетерминал». В теории формальных языков, листья такого дерева – терминалы грамматики, внутренние узлы – нетерминалы.

Существует множество возможных путей вывода из символа цели до окончательного предложения, в зависимости от порядка, в котором применяются правила. Левым каноническим выбором – называется порядок выбора правила, в котором выбирается самый левый нетерминал. Так же существует правый канонический выбор. Это очень важно не только для генерации, но и при анализе типа – является ли существующий символ частью языка. Когда простое признание сопровождается определение базовой структуры дерева, мы говорим о разборе.

Существуют различные расширения описания включающие в себя BNF, например для описания рекурсий и похожих правил, могут использоваться специальные метасимволы, обозначающие использование конструкции множество раз. Расширения, введенные для упрощения таких конструкций известны как EBNF (Extended BNF). Одной из популярных ворм EBNF является EBNF Вирта.

Изучение синтаксиса и семантики языков программирования могут быть сделаны на многих уровнях, и это является важной частью современной вычислительной техники. Люди используют языки, чтобы общаться. В обычной речи они используют естественные языки, как английский или французский язык; для более специализированных приложений, используются технические языки.

Полезные языки программирования должны удовлетворять описанию и реализации, однако трудно найти языки, которые удовлетворяют обоим требованиям. Языки низкого уровня более эффективны, в то время как языки высокого уровня более восприимчивы для понимания. В последние годы много усилий было потрачено на формализации языков программирования, и в поиске путей для их описания и определения. Конечно, формальный язык программирования должен быть описан с помощью другого языка. Этот язык описания называется - метаязык. Ранние языки программирования были описаны с использованием английского языка как метаязыка. Точная спецификация требует, чтобы метаязык был полностью однозначным, и это не сильная черта английского языка.

Натуральные языки, технические языки и языки программирования похожи по нескольким принципам. В каждом случае предложения состоят из последовательности символов или токенов или слов, и построение этих предложений регулируется путем применения двух наборов правил:

– Синтаксические правила - описывают форму предложений в языке. Например, в английском языке, предложение "They can fish" синтаксически правильно, в то время как предложение "Can fish they" не соответствует действительности. Возьмем другой пример, язык двоичных цифр используются только символы 0 и 1, расположенных в строках, образованных путем объединения: 101 - синтаксически правильно, в то время как предложение 1110211 - синтаксически некорректно.

– Семантические правила - определяют значение синтаксически правильных предложений в языке. Сам по себе код 101 не имеет смысла без добавления семантических правил о том, что это должно быть интерпретировано как представление некоторого числа. Предложение "They can fish" более интересно, так как оно может иметь два возможных значения; набор семантических правил будет еще труднее сформулировать.

ЛИТЕРАТУРА

[1] Wirth, N. Compiler Construction: Addison-Wesley, Wokingham, England, 1996. – 133 p.

[2] Wirth, N. The programming language Oberon: Software - Practice and Experience, 1988. 671 – 690 p.

[3]) Aho A.V., Sethi R., Ullman J.D.: Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, MA, 1986. – 295 p.

[4] Backhouse, R.C: Syntax of Programming Languages: Theory and Practice, Prentice-Hall, Hemel Hempstead, England, 1979. – 320 p.

[5] Burns A., Davies G.: Concurrent Programming, Addison-Wesley, Wokingham, England, 1993. – 170 p.

[6] Elder J. Compiler Construction: a recursive descent model, Prentice-Hall, Hemel, Hempstead, England, 1994.

[7] Grosch, J. Efficient and comfortable error recovery in recursive descent parsers, Structured Programming, 1990. – 129 – 140 p.

[8] Grune D., Jacobs C.J.H.: A programmer-friendly LL(1) parser generator, Software, Practice and Experience, 1988. – 29 – 38 p.

[9] Holmes J.: Object-Oriented Compiler Construction, Prentice-Hall, Englewood Cliffs, NJ, 1995. – 30p.

[10] Parr T.J., Quong R.W.: ANTLR: A predicated-LL(k) parser generator, Software - Practice and Experience, 1995. – 789-810 p. [11] Parr, T.J.: Language translation using PCCTS and C++ (a Reference Guide), Automata Publishing, San Jose, CA, 1996. – 10 p.