5 КОМПЬЮТЕРНЫЕ СИСТЕМЫ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

УДК 004.423.4

ОСНОВНЫЕ ПРИНЦИПЫ РЕШЕНИЯ ЗАДАЧИ ПРЕОБРАЗОВАНИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО КОДА В ФОРМАТ RDF СРЕДСТВАМИ СЕМАНТИЧЕСКОГО АНАЛИЗА

А.В. Зараковский, С.В. Клименков, Н.И. Ткаченко, А.Е. Харитонова

Описано решение задачи преобразования исходного кода на объектно-ориентированном языке в формат RDF средствами семантического анализа. Рассмотрены основные принципы семантического анализа объектно-ориентированного кода и предложен архитектурный прототип программного продукта, осуществляющего преобразование кода в RDF.

Ключевые слова: код, RDF, семантический анализ, машинный анализ, триплет, грамматика, синтаксическое дерево, лексер, парсер, Java, ANTLR.

Введение

Одним из обязательных условий успешной разработки и сопровождения любого программного продукта является удобочитаемость его исходного кода. Стремясь добиться этого, программисты стараются делать код более логичным и понятным, именуя должным образом переменные и методы, используя правильно подобранные типы и классы, снабжая код комментариями. Все это говорит о том, что код является отличным источником информации о самом себе.

Программист, читая чужую программу с подробными комментариями, без труда поймет, что делает каждый метод, для чего предназначена любая переменная или же любой объект. Человек способен анализировать текст программы, распознавая в нем целостные куски информации. На самом деле в этот момент он осуществляет разбор кода на информационные блоки и сопоставление получаемых фактов друг с другом, так что, по сути, он строит зависимости в виде триплетов «объект – предикат – субъект» и оперирует уже ими. В качестве примеров можно привести обычные мысли вроде «переменная представляет целое число» или «метод запускается без параметров». В случае машинного анализа все не так просто. Машина не обладает разумом и способностью к анализу, а значит, не сможет самостоятельно провести разбор кода, как это делает человек. Таким образом, для успешной реализации машинного анализа кода необходимо выбрать формат представления данных и разработать специальную программу, которая осуществит разбор кода и приведет его к нужному программисту виду.

Язык описания ресурсов

В рамках данной работы в качестве формата, в который осуществляется преобразование программного кода, выбран Resource Description Framework (RDF). RDF – это разработанная консорциумом Всемирной паутины модель для представления данных, входящая в концепцию семантической паутины, а в особенности – метаданных. Метаданные – это информация о данных. Зачастую это структурированные данные, представляющие собой характеристики описываемых сущностей для целей их идентификации, поиска, оценки и управления ими. Важной особенностью формата RDF является то, что он описывает ресурсы в виде, пригодном (и достаточно удобном) для машинной обработки.

Ресурсом в RDF может быть любая сущность – информационная (сайт, программа, изображение) или неинформационная (человек, город, любое абстрактное понятие). Утверждение о ресурсе строится в виде триплетов и имеет вид «субъект – предикат – объект». Множество RDF-утверждений образует ориентированный граф, в котором вершинами являются субъекты и объекты, а предикаты являются ребрами.

RDF сам по себе представляется не каким-либо форматом файла, а абстрактной моделью. Для работы с RDF в основном используются несколько форматов, в число которых входят RDF/XML, RDFa, N3. Все они уже имеют определенные правила записи и структуру [1].

Семантический анализ программного кода

Основные принципы. Определившись с форматом хранения данных, можно перейти к основной задаче – собственно семантическому анализу программного кода. Для начала рассмотрим основные аспекты семантики любого объектно-ориентированного языка программирования.

Контекстно-свободная грамматика. Любой язык программирования имеет свой алфавит, который целиком состоит из терминальных и нетерминальных символов. Терминальный символ – это любой

имеющий конкретное известное значение символ, как, например, цифра или буква. Нетерминальный символ – это элемент конструкции языка, не имеющий заранее известного значения, как, например, формула или команда. Нетерминальные символы состоят из множества терминальных и в итоге к ним сводятся.

Язык программирования описывается набором правил, выделяющих некоторое подмножество известных символов из множества слов конечного алфавита языка. Такой набор правил называется грамматикой языка и может как задавать правила определения правильности построения слова языка, так и позволять построить любое новое слово языка. Первые являются распознающими (или аналитическими) грамматиками, вторые же являются порождающими.

Грамматика, левая часть правил которой состоит целиком из нетерминальных символов, называется контекстно-свободной [2].

Расширенная форма Бэкуса-Наура. Расширенная форма Бэкуса-Наура (РБНФ) – это формальная система определения синтаксиса, в которой одни синтаксические зависимости определяются через другие. Используется для описания контекстно-свободных грамматик.

Правило в РБНФ имеет вид «идентификатор = выражение.», где идентификатор – это имя нетерминального символа, а выражение – это соответствующая правилам РБНФ комбинация терминальных и нетерминальных символов. Точка в конце – это специальный символ, указывающий на завершение правила.

Набор возможных конструкций РБНФ очень невелик и состоит лишь из конкатенации символов, выбора, условного вхождения и повторения. Также при записи правила РБНФ можно использовать группирующие скобки для определения сложных правил [3].

Синтаксическое дерево. Объектно-ориентированный код, будучи правильно оформленным, сам по себе предоставляет отличные возможности для семантического анализа. При соблюдении правил именования сущностей в коде и его правильной структуры всегда существует возможность получить объектную модель, отражающую организацию кода. Каждый объект в программе можно рассматривать атомарно, не вдаваясь в подробности его реализации. Это называется абстрагированием. В свое же время каждый объект может включать в себя множество других объектов, которые также можно рассматривать как конечную атомарную сущность. Элементарные типы, которые помимо других объектов могут включаться в объект, всегда являются конечными и не имеют возможности инкапсулировать в себе что-либо. В данном случае при употреблении термина «объект» под объектом все же понимается не экземпляр класса, как можно было подумать, а некий атомарный блок информации – метод, класс, объявление переменной, блок комментариев или даже программа целиком. Каждый объект каким-то образом связан с другими, например, имеет некую вложенность в другой объект.

Рассмотрим следующий код:

```
public class Hello {
            public static void main(String[] args) {
                  String str = "Hello, world!";
                  System.out.println( str );
            }
}
```

Эта программа объявляет переменную str, присваивает ей значение, а затем вызывает метод вывода ее содержимого в поток out.

При анализе этого кода можно выделить класс Hello, в котором объявлен метод void main(String[] args), в котором объявлена переменная String str, которой присваивается значение «Hello, world!», и вызывается метод println() с входным параметром str объекта out класса System. Соответственно, это можно и нужно отразить не словами, а в виде некой формализованной модели, которая покажет вложенность кода, значения, типы и сами сущности. Наилучшим образом для этого подходит двоичное дерево, такое, что каждый его узел может быть законченным, т.е. отражать некое детерминированное значение или поле любого примитивного типа, или расходиться вниз, демонстрируя, что элемент не является заключительным в иерархии. Примерное такое дерево, построенное по вышеприведенному коду, можно увидеть на рис. 1.

Таким образом, при семантическом анализе объектно-ориентированного кода можно составить иерархию объектов и элементарных типов, а также провести зависимости и связи между ними. Такую иерархию представляет абстрактное синтаксическое дерево (АСД) или Abstract Syntax Tree (AST), листья которого представляют элементарные типы в коде, а различные ветви – объекты, представленные в коде. АСД – это конечное, помеченное, ориентированное дерево, в котором вершины сопоставлены с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются лишь пустыми операторами и представляют только переменные и константы. Количество ветвей, как и количество листьев, не ограничено, однако дерево всегда имеет одну вершину – объект, представляющий программу целиком.

Применение контекстно-свободной грамматики при разборе кода приводит к получению абстрактного синтаксического дерева, в котором элементы не определяются конкретной грамматикой разбираемого языка. Классическим примером являются ограничительные скобки в языковых конструкциях –

группировка операндов в АСД явно задается структурой дерева, а ограничивающие скобки вообще отсутствуют в разборе, так как не влияют на АСД. АСД со свободной грамматикой отличается от дерева разбора, т.е. дерева с конкретной грамматикой, тем, что в нем отсутствуют ребра и узлы для тех синтаксических правил, которые никак не влияют на семантику программы. Для языка с контекстно-свободной грамматикой составление АСД является достаточно тривиальной задачей. Большинство правил грамматики создают вершину, а символы в правиле становятся ребрами. Правила, которые ничего не привносят в АСД, такие, как, например, группирующие, просто заменяются в вершине одним из своих символов. Кроме того, анализатор может создать полное дерево разбора и затем пройти по нему, удаляя узлы и ребра, которые не используются в абстрактном синтаксисе, чтобы получить АСД.

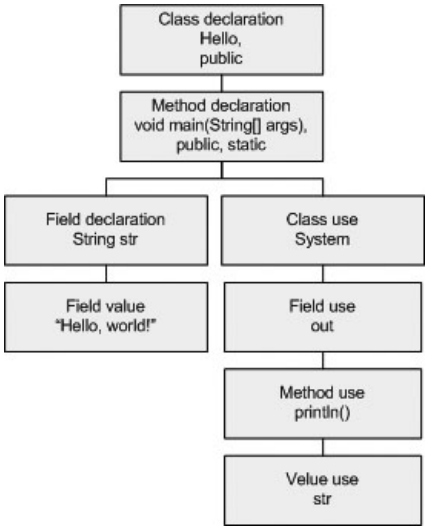

Рис. 1. Двоичное дерево разбора кода

В конечном итоге после разбора кода нас интересует, само собой, не АСД, а структура программы, откуда удалены все ненужные звенья дерева [4].

Решение задачи разбора кода

Построить семантическую модель кода и дерево разбора должны программа-лексер и программа-парсер, понимающие грамматику выбранного языка программирования, проходя от вершины к конечным операндам. Лексер – это программный модуль, осуществляющий разбор текста по заданным грамматическим правилам и генерирующий поток токенов, т.е. нетерминальных символов языка программирования. Например, токенами языка Java для полной грамматики языка будут являться класс, метод, блок комментариев, блок кода. Парсер – это программный модуль, который, основываясь на поступающих из потока лексера токенах, по заданным правилам строит абстрактное дерево разбора. Лексер должен выделять в коде следующие сущности, которые описываются лексическими правилами или грамматикой:

- функции, методы и процедуры, а также их вызовы;
- объекты и их вложенность;
- поля элементарных типов;
- комментарии, условные комментарии и документирующие комментарии;
- директивы;
- блоки кода и управляющие конструкции;
- различные атрибуты любой из перечисленных сущностей.

Дальнейший разбор дерева будет заключаться в выделении необходимых и важных для поставленной задачи узлов и листьев АСД. К примеру, не имеет смысла переводить в вид RDF различные терминальные символы, если они находятся в левой части грамматического правила. Таким образом, не имеет значения все то, что не укладывается в контекстно-свободную грамматику. В связи с тем, что RDF является языком описания ресурсов, а терминал вряд ли можно к таковым отнести, то парсер АСД в RDF должен такие листья дерева исключать из разбора.

На вход парсера в RDF поступают АСД и правила разбора в RDF. Работа парсера заключается в переводе поступающей информации в вид триплетов «субъект – предикат – объект», если эта информация имеет ценность для поставленной задачи. Генерируемый парсером RDF поток триплетов поступает на вход модуля, записывающего триплеты в выбранном формате.

Архитектурная модель создаваемой системы представлена на рис. 2.

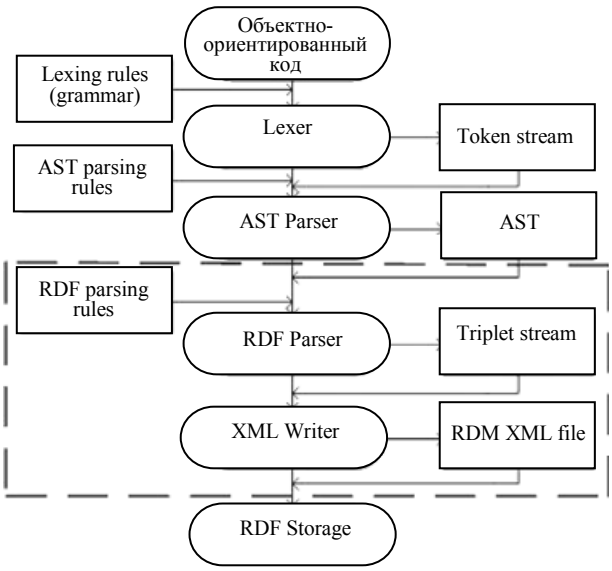

Рис. 2. Архитектура системы разбора кода в RDF

Программная реализация разбора кода

Существует два метода разбора кода – нисходящий и восходящий. Нисходящий разбор начинается с корня дерева, осуществляя рекурсивный проход до листьев. Восходящий же, наоборот, начинает проход с листьев, а заканчивает корнем. Алгоритмы восходящего метода разбора – более сложные, а в совокупности с тем фактом, что избыточность RDF не повлечет каких-то серьезных последствий, можно утверждать, что разумнее в задаче перевода кода в RDF использовать более простой и быстрый метод нисходящего разбора.

Существует множество программных библиотек, помогающих разобрать текст программы на контекстно-свободном языке программирования и получить его абстрактное представление в виде двоичного дерева. Каждое такое средство разбора оперирует грамматикой выбранного языка, написанной с соблюдением единой РБНФ, проводя лексический и синтаксический разборы.

Одно из подобных программных средств – это Another Tool for Language Recognition (ANTLR). ANTLR – это генератор парсеров на языке Java, который принимает на вход файл грамматики, написанный по единой форме Бэкуса–Наура, и файл с исходным кодом разбираемой программы, а на выходе создает абстрактное синтаксическое дерево. На самом деле провайдер ANTLR предоставляет утилиту ANTLRWorks, позволяющую описать грамматику и генерирующую классы лексера и парсера. Помимо утилиты предоставляется библиотека Java времени выполнения ANTLR, которая как раз и запускает в работу лексер и парсер, получает от них результат и строит дерево.

Готовых решений для перевода дерева в RDF не существует. Соответственно, встает задача разработки подобного решения. Поскольку к этому моменту уже имеется формальная форма представления кода, которая является результатом работы ANTLR, перед парсером RDF ставятся только следующие задачи:

- выборка важных и необходимых узлов АСД;
- сопоставление объектов, субъектов и предикатов на основе анализа ветвей АСД;
- формирование триплетов RDF;
- создание и запись XML-сущностей (или других, что зависит от выбранной формы записи RDF), отражающих суть каждого триплета.

Решение должно состоять из двух модулей – RDF Parser и XML Writer.

RDF Parser должен проходить по всем узлам дерева разбора, проверять каждый на соответствие правилам разбора в RDF и, если узел необходим для решения задачи, генерировать новый триплет в потоке триплетов и запоминать узел как проверенный.

XML Writer, получая триплеты из потока, генерирует XML-код, который описывает триплеты в нужном формате.

Алгоритмы работы RDF Parser и XML Writer представлены на рис. 3, а, б.

Весь программный комплекс, как «черный ящик», должен выглядеть так: на входе – исходный код программы, на выходе – требуемая форма записи RDF и, при необходимости, – отчет об ошибках. Отчет об ошибках служит для дополнительного анализа, если, к примеру, исходный код программы был написан с нарушениями общепринятых стандартов записи кода (например, соглашение о записи кода на языке Java).

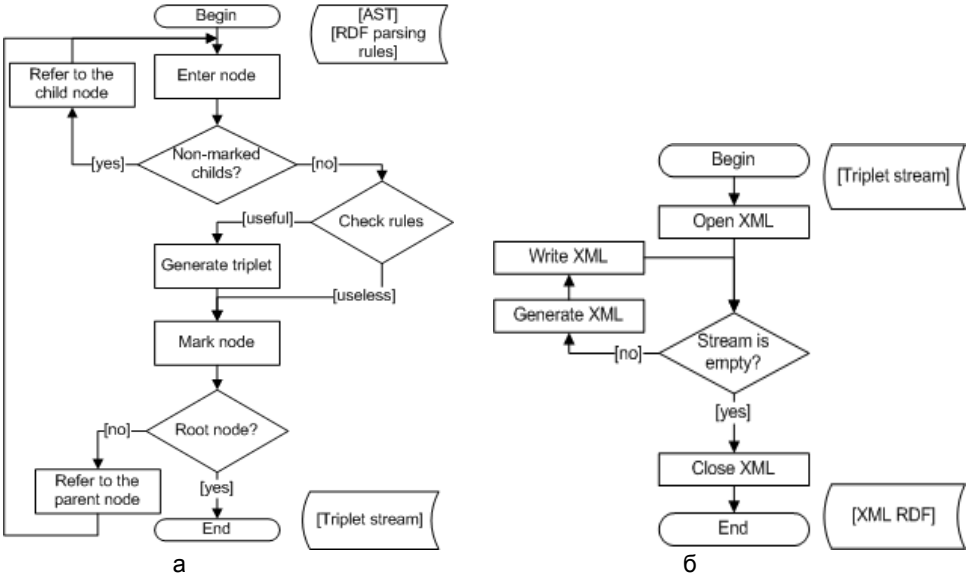

Рис. 3. Алгоритм RDF Parser (а); алгоритм XML Writer (б)

Заключение

В работе рассмотрены основные принципы решения задачи преобразования исходного кода на объектно-ориентированном языке программирования в формат RDF и предложен прототип архитектуры программного продукта решающего эту задачу. Программный продукт, основанный на предложенном архитектурном прототипе, может найти достаточно широкое применение при решении задач анализа кода, в экспертных системах и САПР.

Литература

1. Eric Miller. RDF Primer: W3C Recommendation // W3C [Электронный ресурс]. – Режим доступа: http://www.w3.org/TR/rdf-primer/ (дата обращения: 20.01.10).
2. Context-free grammar // Wikipedia [Электронный ресурс]. – Режим доступа: http://en.wikipedia.org/wiki/Context-free_grammar (дата обращения: 04.02.10).
3. Extended Backus-Naur Form // Wikipedia [Электронный ресурс]. – Режим доступа: http://en.wikipedia.org/wiki/EBNF (дата обращения: 04.02.10).
4. Parr T. The definitive ANTLR reference: Building domain-specific languages / Terence Parr. – San Francisco; The pragmatic programmers, LLC, 2007. – 384 p.

Зараковский Алексей Владимирович – Санкт-Петербургский государственный университет информационных технологий, механики и оптики, студент, dbiryukov@list.ru

Клименков Сергей Викторович – Санкт-Петербургский государственный университет информационных технологий, механики и оптики, ассистент, Serge.Klimenkov@Servicom.Ru

Ткаченко Никита Иванович – Санкт-Петербургский государственный университет информационных технологий, механики и оптики, студент, n.i.tkachenko@gmail.com

Харитонова Анастасия Евгеньевна – Санкт-Петербургский государственный университет информационных технологий, механики и оптики, ассистент, Anastassia.Kharitonova@Elcom.SPb.Ru