Impact Factor:			
ISRA (India) = 1.344	SIS (USA) = 0.912	ICV (Poland) = 6.630
ISI (Dubai, UAE) = 0.829	РИНЦ (Russia) = 0.207	PIF (India) = 1.940
GIF (Australia) = 0.564	ESJI (KZ) = 4.102	IBI (India) = 4.260
JIF = 1.500	SJIF (Morocco) = 2.031	

SOI: 1.1/TAS DOI: 10.15863/TAS

International Scientific Journal

Theoretical & Applied Science

p-ISSN: 2308-4944 (print) **e-ISSN:** 2409-0085 (online)

Year: 2018 **Issue:** 05 **Volume:** 61

Published: 30.05.2018 **http://T-Science.org**

Andrey Andreevich Kadomskij
Researcher
Peter the Great St. Petersburg Polytechnic University
Saint-Petersburg, Russia

Oleg Yurievich Sabinin
PhD in Computer Science
Peter the Great St. Petersburg Polytechnic University
Saint-Petersburg, Russia

SECTION 4. Computer science, computer engineering and automation.

STUDY OF THE POSSIBILITY OF CREATING A UNIVERSAL LANGUAGE OF HIGH-LEVEL PROGRAMMING

Abstract: *The purpose of the work is to create a lexical-syntactic analyzer, which can serve as a basis for creating a universal and practical high-level programming course. The peculiarity of such a programming language is that with the help of lexical-syntactic analysis the source code in this language can be converted into the source code in any other programming language and vice versa.*

Key words: *lexical analysis, syntactic analysis, parser, compiler*

Language: *Russian*

Citation: Kadomskij AA, Sabinin OY (2018) STUDY OF THE POSSIBILITY OF CREATING A UNIVERSAL LANGUAGE OF HIGH-LEVEL PROGRAMMING. ISJ Theoretical & Applied Science, 05 (61): 84-90.

Soi: http://s-o-i.org/1.1/TAS-05-61-17 ***Doi***: crossref https://dx.doi.org/10.15863/TAS.2018.05.61.17

ИССЛЕДОВАНИЕ ВОЗМОЖНОСТИ СОЗДАНИЯ УНИВЕРСАЛЬНОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ

Аннотация: *Целью работы является создание лексико-синтаксического анализатора, который может послужить основой для создания универсального и практичного языка программирования высокого уровня. Особенностью такого языка программирования является то, что при помощи лексическо-синтаксического анализа исходный код на этом языке может быть преобразован в исходный код на любом другом языке программирования и обратно.*

Ключевые слова: *лексический анализ, синтаксический анализ, парсер, компилятор*

Введение

До середины 60-х компьютеры были слишком дорогими устройствами, которые использовались только для проведения специфических научных вычислений. Скорость выполнения задач была в эти годы приоритетным фактором.

Однако со временем цены на компьютеры стали падать, а скорость вычислений увеличиваться. Наступило время, когда создатели языков начали задумываться над удобством языка. Однако на тот момент уже было написано большое количество программ, для которых совместимость стала главной задачей, а скорость выполнения начала отходить на второй план.

С тех пор тянется шлейф устаревших подходов и методов, которые необходимо поддерживать.

В итоге имеются десятки языков, со своими достоинствами и недостатками, но один единственный, в который можно перенести любую программу, это язык ассемблера. Современный ассемблер едва ли сильно отличается от самого первого языка ассемблера. Однако же он сложен для понимания, размеры исходников велики, и вряд ли кто-то возьмется разрабатывать на нем большие бизнес-приложения, веб-страницы или игры. Необходимость в таком языке, исходный код которого можно перевести в исходный код любого другого языка существует и по сей день. И наша задача выяснить, возможно ли это как этого добиться и каких путей придерживаться.

Постановка задачи

На сегодняшний день существует множество инструментов, решающих

аналогичные задачи, однако, часто их функционала недостаточно.

Среди таких инструментов существуют программные продукты, способные проводить хороший анализ кода для определенного языка (например, Splint, PREFast для C), но, как правило, их трудно переносить на другие языки.

Так же существуют более универсальные программные инструменты, такие как FxCop, способный проводить анализ любого кода под .NET, но не предоставляет возможности автоматического исправления. Для языка не под .NET он уже не предназначен, кроме того, не может работать с некорректным входом, поскольку анализирует уже откомпилированные программы.

Есть коммерческие продукты (например, DMS), возможности которых велики, однако, они не подходят для использования в open-source проектах.

Так же, есть инструменты, занимающиеся автоматическим форматированием текста (например, Artistic Style, Indent), но эта функциональность весьма ограничена, и для ее реализации часто требуется минимальная часть синтаксиса языка, а для ее расширения на статический анализ требуется много работы.

Как видно из приведенных примеров, большинство инструментов недостаточно универсальны, что и послужило причиной для создания собственного инструмента.

Целью нашей работы является создание лексического и синтаксического анализатора, который в дальнейшем послужит основой для создания языка программирования высокого уровня, особенностью которого является его универсальность и практичность, позволяющая преобразовывать исходный код на нашем языке программирования в исходный код на любом другом языке программирования и обратно.

Преимуществом данного технологического решения является то, что достаточно знать всего один язык программирования для того, чтобы использовать все многообразие языков программирования. Так же, благодаря такому подходу мы получим возможность использования любых библиотек независимо от того, на каком языке они были написаны. Помимо этого, открывается возможность ускорить процесс разработки и перенесения программного продукта между различными платформами и архитектурами, так как уменьшаются временные затраты в процессе разработки.

Поэтому, результатом научного проекта должна стать программа, выполняющая лексический и синтаксический анализ. Данная программа должна разбивать входной текст на лексемы, создавать структуру для их хранения, и передавать поток лексем для последующей их обработки синтаксическим анализатором.

Теоретические основы работы

Главная задача лексического анализатора заключается в чтении поступающих символов исходного текста программы, группировании их в лексемы, а также в выводе последовательности токенов для всех лексем исходной программы.

Лексемами любого языка программирования являются идентификаторы, константы, ключевые слова, знаки операций и разделители. Состав возможных лексем для каждого конкретного языка программирования определяется синтаксисом этого языка. Для реализации лексического анализатора стоит начать с построения диаграммы или другого описания лексемы каждого токена. Затем необходимо написать программу, которая будет производить идентификацию встреченных ей лексем, и на основе анализа возвращать информацию об обнаруженном токене.

Далее поток токенов пересылается синтаксическому анализатору для разбора. В большинстве компиляторов лексический и синтаксический анализаторы — это взаимосвязанные части. Благодаря чему лексический разбор исходного текста выполняется поэтапно так, что синтаксический анализатор, выполнив разбор очередной конструкции языка, обращается к лексическому анализатору за следующей лексемой. Обычно при работе лексический анализатор так же взаимодействует с таблицей символов, в которую помещает лексемы. Описанное взаимодействие представлено на (рис. 1)

Рисунок 1 - Взаимодействие лексического и синтаксического анализатора

Однако, можно упростить задачу, определив шаблоны лексем. Данный подход упрощает внесение изменений в лексический анализатор, поскольку для этого необходимо переписать лишь измененные шаблоны, но не весь код. Так же подобный подход упрощает процесс реализации лексического анализатора, поскольку программист взаимодействует только с высокоуровневыми шаблонами, абстрагируясь от работы над детальным кодом, который в свою очередь, является результатом работы генераторов лексических анализаторов. Среди таких инструментов можно выделить:

ANTLR — генератор парсеров написанный на Java, который позволяет создавать лексические и синтаксические анализаторы на различных языках (Java,C,C++,Python, C#,ActionScript,JavaScript,PHP) на основе грамматик.

Coco/R — программный продукт генерации компиляторов или интерпретаторов языка.

Bison — развитие инструмента YACC. Все грамматики, созданные для оригинального YACC, будут работать и в Bison.

И другие, среди которых можно выделать JavaCC, SLK, Menhire, ASF+SDF, Elkhound. В (табл.1) приведены данные о поддержке языков различными генераторами.

Таблица 1.

Языки инструментов

Название инструмента	Языки программирования					
	C	C++	C#	Java	OCaml	Python
ANTLR	−	+	+	+	−	+
ASF+SDF	+	−	−	−	−	−
Bison	+	−	−	−	−	−
Coco/R	+	+	+	+	−	−
Elkhound	−	+	−	−	+	−
JavaCC	−	−	−	+	−	−
Menhir	−	−	−	−	+	−
SLK	+	+	+	+	−	−

Нами же для дальнейшей разработки был выбран программный инструмент Lex, который позволяет определить лексический анализатор, указывая регулярные выражения для описания шаблонов токенов. На (рис.2) приведена схема работы генератора лексических анализаторов Lex. Из данной схемы видно, что входной файл lex.l описывает генерируемый лексический анализатор, а компилятор Lex преобразует файл lex.l в программу на языке программирования C. Данный файл lex.yy.c компилируется уже компилятором C в a.out файл, он и представляет собой работающий лексический анализатор, который может получить поток входных символов, и выдать на основе их поток токенов.

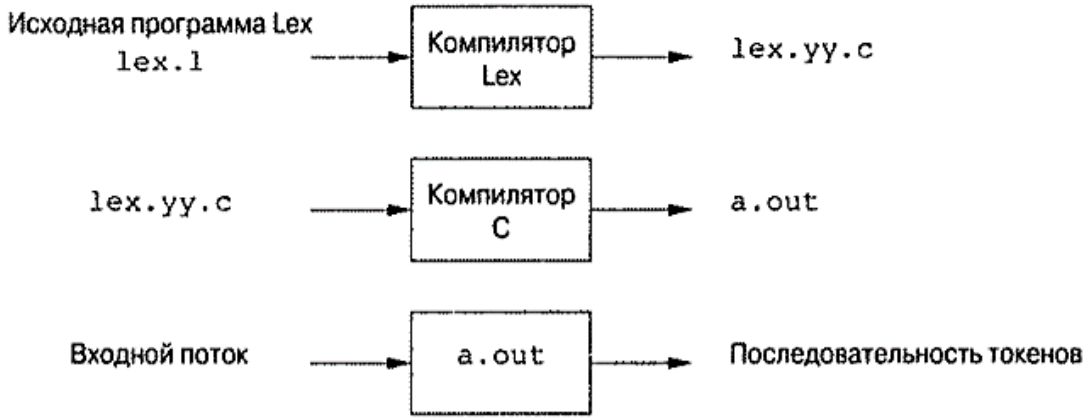

Рисунок 2 - Создание лексического анализатора при помощи Lex

В задачу же синтаксического анализатора входит найти и выделить основные синтаксические конструкции в тексте входной программы, установить тип и проверить правильность каждой синтаксической конструкции и, представить синтаксические конструкции в виде, удобном для дальнейшей генерации текста результирующей программы.

Синтаксический анализатор получает строку токенов от лексического анализатора, и проверяет, может ли эта строка токенов порождаться грамматикой входного языка. Ещё одной функцией синтаксического анализатора является генерация сообщений обо всех выявленных ошибках, причём достаточно внятных и полных, а кроме того, синтаксический анализатор должен уметь обрабатывать обычные, часто встречающиеся ошибки и продолжать работу с оставшейся частью программы. В случае корректной программы синтаксический анализатор строит дерево разбора и передаёт его следующей части компилятора для дальнейшей обработки. В дальнейшем для упрощения процесса разработки нами предполагается использовать автоматически сгенерированный синтаксический анализатор, полученный при помощи программного инструмента Bison, который представляет собой генератор синтаксических анализаторов на основе LALR-грамматик. Любой синтаксический анализатор, сгенерированный с использованием Bison, формирует дерево разбора по мере анализа поступающих токенов. На (рис.3) приведена схема совместной работы лексического и синтаксического анализатора.

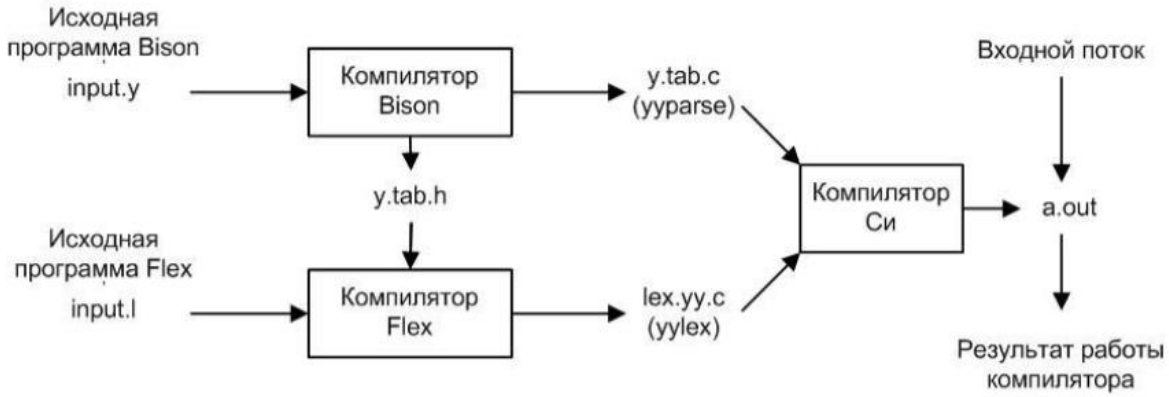

Рисунок 3 - Совместная работа Flex и Bison

Bison работает с грамматикой, которая определяется во входном файле, и генерирует синтаксический анализатор, распознающий «предложения», соответствующие этой грамматике. Однако, синтаксически правильная программа необязательно является семантически верной. Например, для языка Си присваивание строкового значения целочисленной переменной типа int – семантически неправильно, но удовлетворяет синтаксическим правилам языка. Bison проверяет лишь правильность синтаксиса.

Описание грамматики на языке Bison и его соответствие форме Бэкуса-Наура приведено в (табл.2). Вертикальная черта (|) показывает, что есть две возможности задания одного и того же нетерминального символа или что несколько правил могут иметь идентичную левую часть. Символы в левой части правила – нетерминалы. Символы, возвращаемые лексическим анализатором, – терминалы или токены.

Таблица 2.

Описание грамматики на языке Bison

Пример грамматики (Bison)	Форма Бэкуса - Наура
statement: NAME'='expression expression: NUMBER'+'NUMBER | NUMBER'–'NUMBER	statement ➜ NAME'='expression expression ➜ NUMBER'+'NUMBER | NUMBER'–'NUMBER

Построим дерево разбора для выражения fred = 12 + 13.

В данном примере 12 + 13 соответствует нетерминалу expression, а fred = expression формирует statement. Любая грамматика содержит начальный символ, который выступает в качестве корня дерева разбора. В данной грамматике statement является таким символом. Дерево разбора для данного примера представленно на (рис.4).

В синтаксическом дереве внутренние узлы (вершины) представляют собой операции, а листья соответствуют операндам. Как правило, листья синтаксического дерева соответствуют записям в таблице идентификаторов. Структура синтаксического дерева отражает синтаксис языка программирования, на котором написана исходная программа. Синтаксические деревья содержат информацию о действиях, которые необходимо выполнить компилятору над соответствующими элементами.

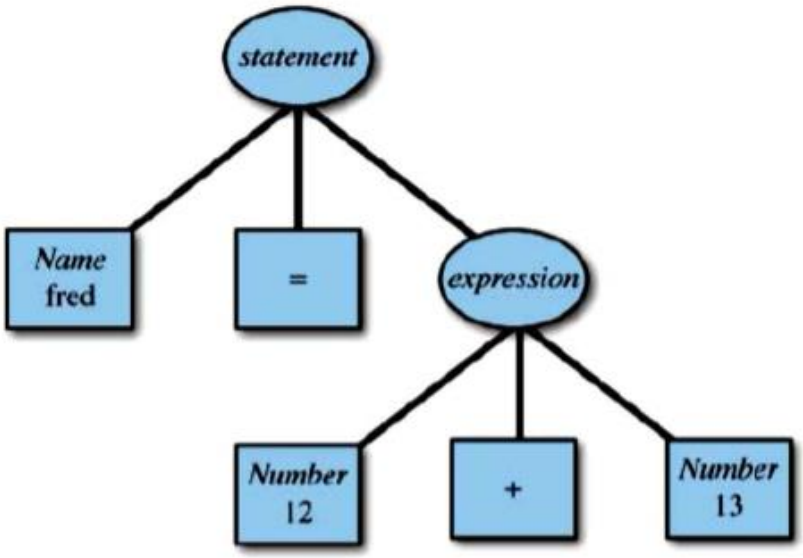

Рисунок 4 - Дерево разбора для fred = 12 + 13

Таким образом, благодаря использованию Bison мы имеем возможность получать автоматически сгенерированный синтаксический анализатор, который планируется использовать при дальнейшей разработке.

Разработка и исследование программы

В процессе исследования нами была разработана программа, которая проводит лексический и синтаксический анализ исходного текста.

Данная программа содержит в себе два класса. Класс LoadSaveFile и класс ParserComment. Конструктор класса LoadSaveFile принимает переменную типа string в качестве параметра, в которой содержится путь до файла с исходным кодом, и копирует ее содержимое в переменную file типа string. Что является начальным действием в нашей программе. Функция void SaveFile(string path, string file) является завершающим действием. Она служит для внесения изменений и сохранения файлов.

Класс же ParserComment отвечает только за парсинг кода. Функция void ParserComment(vector<string> cpp) принимающая в себя в качестве параметра код программы, проводит отделение комментарии от исходного кода и помещает их в векторную структуру vector<string> keyWords.

```
void ParseComment(vector<string> cpp) {
for (unsigned int i = 0; i <= cpp.size() - 1; i++) {
                    string str = cpp[i];
                    string split("/*");
                    size_t prev = 0;
                    size_t next;
                size_t delta = split.length();
while ((next = str.find(split, prev)) != string::npos) {
keyWords.push_back(str.substr(next + 2, str.length()
- 1));
                    prev = next + delta;
                    }
                }
        }
```

Далее функция void replace() анализирует структуру keyWords, содержащую в себе пары комментариев наподобие “using=использовать”, и разделяет их на две новые отдельные векторные структуры vector<string> whatToReplace и vector<string> whenToReplace, в одну из которых помещает помещает “using”, а в другую “использовать”.

```
void replace() {
                for (unsigned int i = 0; i <=
keyWords.size() - 1; i++) {
                    string str = keyWords[i];
                    string split(";");
                    size_t prev = 0;
                    size_t next;
                size_t delta = split.length();

while ((next = str.find(split, prev)) != string::npos) {
string substring = str.substr(prev, next - prev);
                    bool nextKey = false;
                        string IN = "";
                        string TO = "";
for (unsigned int j = 0; j <= substring.length() - 1;
j++) {
                    if (substring[j] == '=') {
                    nextKey = true;
                                        }
                    else if (nextKey == false)
{

                    IN += substring[j];
                                        }
                    else if (nextKey == true) {

                    TO += substring[j];
                                        }
                                    }

                    whatToReplace.push_back(IN);

                    whenToReplace.push_back(TO);
                    prev = next + delta;
                    }
                }
        }
```

В завершении функция void Subs(vector<string> cpp) проводит парсинг исходной программы, написанной на языке программирования C++ и при помощи whenToReplace преобразует ее исходный код в исходный код на нашем языке программирования.

```
    void Subs(vector<string> cpp) {
for (unsigned int i = 0; i <= cpp.size() - 1; i++) {
                    newFile += cpp[i] + '\n';
                }
for (unsigned int i = 0; i < whenToReplace.size();
i++) {
                    std::tr1::regex
rx(whatToReplace[i]);
                    std::string replacement =
whenToReplace[i];

                    std::string temp =
tr1::regex_replace(newFile, rx, replacement);
                    newFile = temp;
                }
        }
```

В будущем нами предполагается автоматизировать анализ файлов при помощи генератора лексических и синтаксических анализаторов Flex и Bison. В результате это позволит ускорить процесс разработки собственного универсального языка программирования.

Заключение

В результате выполнения работы нами была разработана программа, выполняющая лексический и синтаксический анализ входного текста. На этапе лексического анализа входной текст разбивается на лексемы, создается структура для их хранения. Лексемы передаются синтаксическому анализатору для дальнейшей обработки. В результате выполнения программы мы получаем исходный код, преобразованный в наш язык программирования.

Так же разработанный продукт в дальнейшем послужит основой для создания нового универсального языка программирования высокого уровня, особенностью которого является его практичность, заключающаяся в легко преобразуемой форме исходного кода на нашем языке программирования в исходный код на любом другом языке и наоборот.

Данное исследование показало, что поставленная задача реализуема. При детальном изучении структуры выходных файлов, полученных в результате их преобразования с помощью разработанной нами тестовой программы, были сделаны выводы о сходствах различных языков программирования. Однако, имеются и существенные различия, из-за семантических и архитектурных особенностей каждого языка программирования, которые ставят существенные ограничения по времени на разработку данного программного продукта.

References:

1. Karpov, YU.G. (2005) Teoriya i tekhnologiya programmirovaniya. Osnovy postroeniya translyatorov: uch. posobie. – SPb.: BHV-Peterburg, 2005. – 272 p.
2. Aho A., Seti R., Ul'man D. (2003) Kompilyatory: principy, tekhnologii i instrumenty. – M.: Vil'yams, 2003. - 768.
3. Virt N. (2010) Postroenie kompilyatorov (Klassika programmirovaniya). — M.: DMK-Press,2010. — 192 p.
4. Grigor'ev C.V. (2015) Sintaksicheskij analiz dinamicheski formiruemyh programm : Diss… kandidata nauk / C. V. Grigor'ev ; Sankt-Peterburgskij gosudarstvennyj universitet. — 2015.
5. (2018) Bison – GNU parser generator. Rezhim dostupa: http://www.gnu.org/software/bison/. (data obrashcheniya: 03.04.2018).
6. Niemann T. (2018) A Compact Guide To Lex & Yacc. Rezhim dostupa: http://epaperpress.com/lexandyacc/download/lexyacc.pdf. (data obrashcheniya: 06.04.2018).
7. Hanter R. (2002) Osnovnye koncepcii kompilyatorov. :Per.s angl. — M.::Izdatel'skij dom «Vil'yams», 2002 — 256p.
8. (2018) Available: http://astyle.sourceforge.net/
9. (2018) Available: https://docs.microsoft.com/en-us/previous-versions/windows/embedded/ms933794(v=msdn.10)
10. Ulitin K.A. (2010) Razrabotka arhitektury dlya generatora sintaksicheskih analizatorov, 2010. 15 s. (http://recursiveascent.googlecode.com/files/KonstantinUlitin_CompilerCompilerArchitecture.pdf)