МИНОБРНАУКИ РОССИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет прикладной математики, информатики и механники
Кафедра программного обеспечения и администрирования
информационных систем
Профиль «информационные системы и базы данных»

Разработка лексического и синтаксического анализаторов с целью подсветки синтаксиса и автодополнения исходного кода для предоставленного языка

Бакалаврская работа
Направление 02.03.03. Математическое обеспечение и администрирование информационных систем

Зав. кафедрой _____ д. ф-м. н. проф. М. А. Артёмов
Обучающийся _____ А. С. Пахомов
Руководитель _____ преп. Н. В. Огаркова

Воронеж, 2019

Содержание

Введение

В последнюю очередь.

Глава 1. Аналитическая часть

1.1. Основные понятия теории компиляции

1.1.1. Компилятор

Компилятор — это программа, которая принимает текст, написанный на одном языке — *исходном*, и транслирует (переводит) его в эквивалентный текст на другом языке — *целевом*. Одна из важных ролей компилятора состоит в сообщении об ошибках в исходной программе, обнаруженных в процессе трансляции [1].

Процесс компиляции можно разделить на две большие части. Первая часть — *анализ* (выполняется несколькими анализаторами) разбивает исходную программу на составные части и накладывает на них грамматическую структуру. Затем эта структура используется для промежуточного представления программы. В процессе анализа пользователю сообщается об ошибках, собирается информация об исходной программе и сохраняется в *таблицу символов*. Таблица символов — структура данных, которая используются компилятором для хранения информации о конструкциях исходной программы. Информация накапливается инкрементно в фазе анализа компилятора и используется фазой синтеза для генерации целевого кода.

Вторая часть — *синтез* (выполняется генератором кода), строит требуемую целевую программу на основе промежуточного представления и информации из таблицы символов.

Если рассмотреть процесс компиляции более детально, то можно увидеть, что он представляет собой последовательность фаз, каждая из которых преобразует одно представление программы в другое:

- *Лексический анализ* или *сканирование*. Лексический анализатор читает поток символов, составляющих исходную программу, группирует их в значащие последовательности (лексемы), формирует для каждой лексемы специальные структуры данных, называемые токенами, и передает их синтаксическому анализатору.
- *Синтаксический анализ* или *разбор*. Синтаксический анализатор использует информацию из предыдущей фазы и на её основе строит синтаксическое дерево, в котором каждый внутренний

узел представляет операцию, а дочерние узлы — аргументы этой операции.

– *Семантический анализ*. Эта фаза компиляции использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Важной частью семантического анализа является проверка и приведение типов.
– *Генерация промежуточного кода.* На этой фазе компилятор генерирует низкоуровневое промежуточное представление кода исходной программы, которое можно рассматривать как программу для абстрактной вычислительной машины.
– *Машинно-независимая оптимизация кода.* Фаза машинно-независимой оптимизации кода пытается улучшить промежуточный код.
– *Генерация кода.* В качестве исходных данных генератор кода получает промежуточное представление исходной программы и отображает его в целевой язык, как правило в ассемблерный код.
– *Машинно-зависимая оптимизация кода.* Фаза машинно-зависимой оптимизации кода улучшает код целевой машины, учитывая особенности архитектуры процессора.

Фазы компиляции, которым соответствуют одноименные исполнители, представлены на рис. 1.1.

Процесс анализа компиляции разделен на лексический, синтаксический и семантический анализ по ряду причин:

1. Упрощение разработки. Отделение лексического анализа от синтаксического позволяет упростить как минимум одну из фаз анализа. Например, удаление комментариев и пробельных символов лексическим анализатором значительно проще, чем включение работы с ними в синтаксический анализатор.
2. Увеличение эффективности компилятора. Отдельный лексический анализатор позволяет применять более специализированные методики, предназначенные исключительно для решения лексических задач.

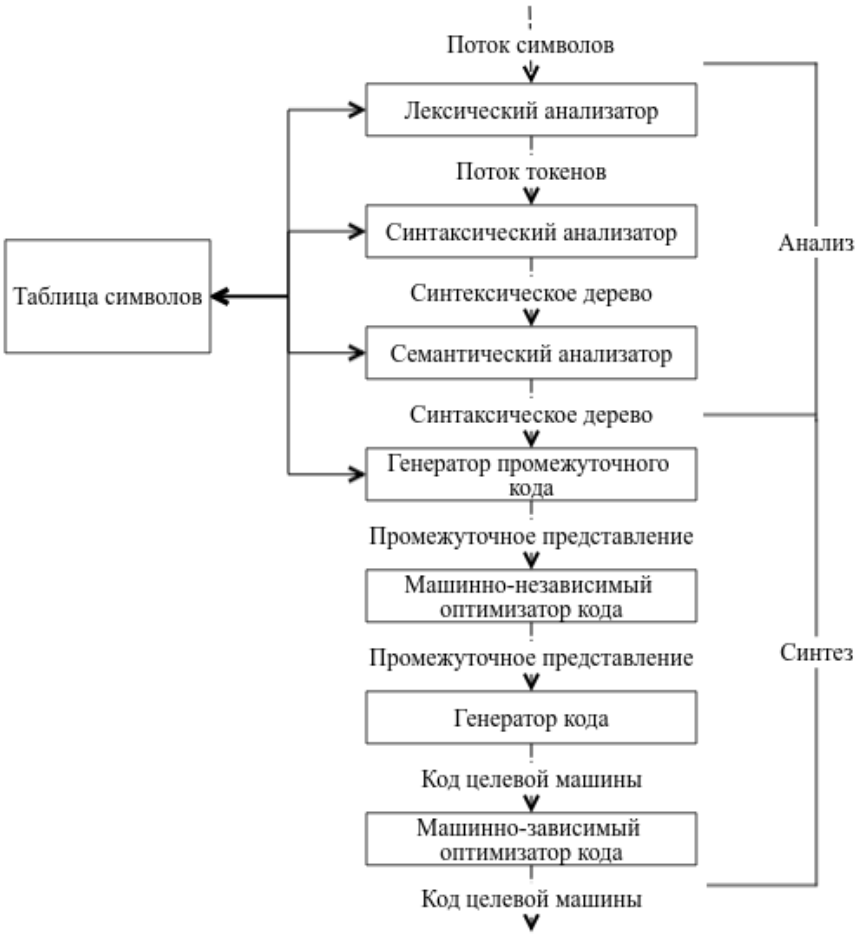

Рис. 1.1. Схема взаимодействия фаз компилятора

3. Увеличение переносимости компилятора. Особенности устройств ввода могут быть ограниченны лексическим анализатором.
4. Разделение зон ответственности. Ошибки типизации будут выявлены на стадии семантического анализа, что позволит сформулировать сообщение об ошибках типизации в более полной мере.

1.1.2. Лексический анализатор

При рассмотрении лексического анализа используются четыре связанных, но различных термина:

– *Токен* (от англ. *token* — знак, символ) представляет собой пару, состоящую из имени токена и необязательного атрибута. Имя токена — абстрактный символ, представляющий тип лексической единицы, например, конкретное ключевое слово или последовательность символов, составляющих идентификатор. Имена токенов являются входными символами, обрабатываемыми

синтаксическим анализатором. Примеры токенов приведены в табл. 1.1.

Таблица 1.1. Примеры токенов

Токен	**Неформальное описание**	**Примеры лексем**
if	Символы `i`, `f`	`if`
else	Символы `e`, `l`, `s` `e`	`else`
comparison	<, >, <=, >=, ==, !=	<=, !=
id	Буква, за которой следуют буквы и цифры	`pi`, `score`, `D2`
number	Любая числовая константа	`3.14159`, `0`
literal	Все, кроме ", заключенное в двойные кавычки	`"core dumped"`

– *Атрибут токена* — строка или структура, объединяющая несколько блоков информации, которая содержит значение числа, в случае если токен соответствует шаблону числа, или описание лексемы (строчка кода, значение), которая представляет токен. Например, выражение на языке Fortran `E=M*2` будет представленно в виде последовательности
⟨**id**, Указатель на запись в таблице символов для E⟩
⟨**assign_op**⟩
⟨**id**, Указатель на запись в таблице символов для M⟩
⟨**mult_op**⟩
⟨**number**, Целое значение 2⟩
– *Шаблон* — это описание вида, который может принимать лексема токена. В случае ключевого слова шаблон представляет собой последовательность символов, образующая это ключевое слово. Для некоторых токенов шаблон представляет более сложную структуру.
– *Лексема* — последовательность символов исходной программы, которая соответствует шаблону токена и идентифицируется лексическим анализатором как экземпляр токена.

Основная задача лексического анализатора состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора. Также

лексический анализатор может удалять комментарии, пробельные символы, синхронизировать сообщения об ошибках или раскрывать макросы.

1.1.3. Формальное определение контекстно-свободной грамматики

Существует множество грамматик (грамматики типа 3, контекстно-свободные грамматики, контекстно-зависимые грамматики и грамматики без ограничений), с помощью которых можно описать формальные языки. Для описания языков программирования используются контекстно-свободные грамматики (далее КС-грамматики).

КС-грамматика используется для определения синтаксиса языка программирования. КС-грамматика естественным образом описывает иерархическую структуру множества конструкций языка программирования. Ниже приведён пример КС-грамматики, которая описывает арифметические выражения:

$$
\begin{aligned}
expression &\to expression + term \\
expression &\to expression - term \\
expression &\to term \\
term &\to term * factor \\
term &\to term / factor \\
term &\to factor \\
factor &\to (expression) \\
factor &\to \textbf{id}
\end{aligned}
$$

КС-грамматика состоит из четырёх частей:

1. *Терминалы* — базовые символы, формирующие строки. Термин «имя токена» является синонимом слова «терминал». Пример терминала: `if`, `else`, `(`, `)`.
2. *Нетерминалы* — синтаксические переменные, которые обозначают множества строк. В примере, приведённом выше, $statement$ и $expression$ являются нетерминалами. Эти множества строк, обозначаемые нетерминалами, помогают определить язык, порождаемый КС-грамматикой. Нетерминалы также налагают на язык иерархическую структуру, облегчающую синтаксический анализ и трансляцию.

3. *Стартовый символ* — один из нетерминалов, который обозначает множество строк, являющиеся языком, определяемым КС-грамматикой. По соглашению, продукции стартового символа указываются первыми.
4. *Продукция* — способ, которым терминалы и нетерминалы объединяются в строки. Каждая продукция состоит из заголовка (левая часть), символа $\rightarrow$ и тела (правая часть), которое состоит из нуля или некоторого количества терминалов и нетерминалов.

1.1.4. Синтаксический анализатор

В классической модели компилятора синтаксический анализатор получает строку токенов от лексического анализатора и проверяет, может ли эта строка имен токенов порождаться грамматикой исходного языка. Также от синтаксического анализатора ожидаются сообщения обо всех выявленных ошибках и умение продолжать работу с оставшейся частью программы. В случае корректной программы синтаксический анализатор строит дерево разбора и передает его следующей части компилятора для дальнейшей обработки.

Имеется три основных типа синтаксических анализаторов грамматик:

– *Универсальные методы разбора*: например, алгоритмы Кока-Янгера-Касами (Cocke-Younger-Kasami) и Эрли (Earley) [2], которые могут работать с любой грамматикой. Однако эти обобщённые методы слишком неэффективны для использования в промышленных компиляторах.
– *Восходящие методы разбора* (bottom-up): построение дерева разбора происходит снизу (от листьев) вверх (к корню). Поток токенов сканируется слева направо.
– *Нисходящие методы разбора* (top-down): дерево разбора строится сверху (от корня) вниз (к листьям). Входной поток синтаксического анализатора, как и в восходящих методах, сканируется посимвольно слева направо.

Пример разбора для выражения $-(\mathbf{id} + \mathbf{id})$ показан на рис. 1.2

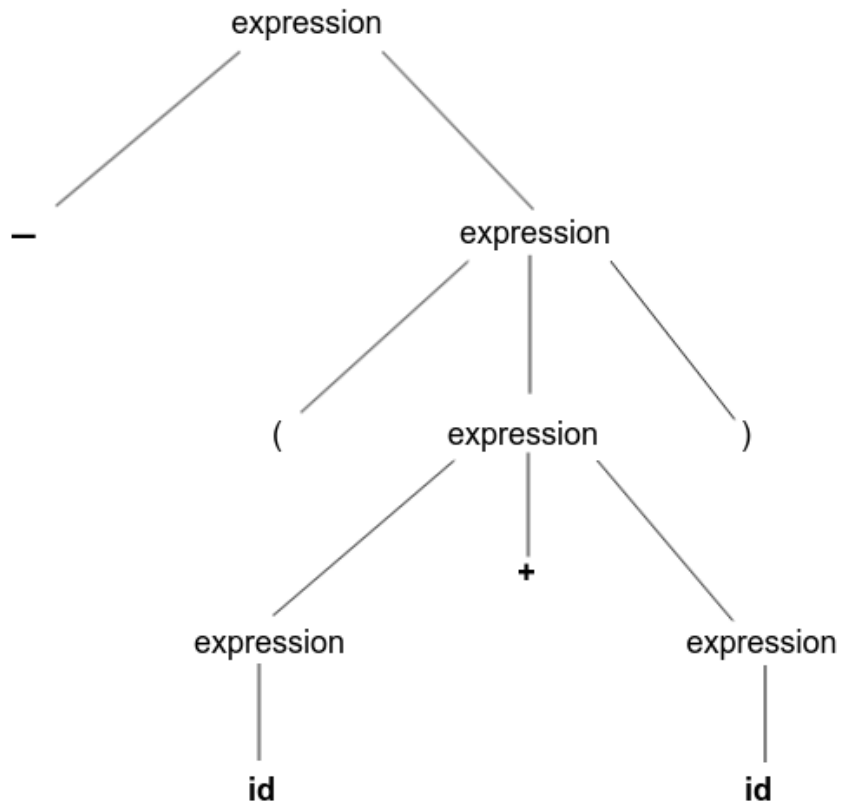

Рис. 1.2. Дерево разбора для выражения $-(\mathbf{id} + \mathbf{id})$

1.1.5. Семантический анализатор

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице символов для последующего использования в процессе генерации промежуточного кода.

Важной частью семантического анализа является проверка типов, когда компилятор проверяет, имеет ли каждый оператор операнды соответствующего типа. Например, многие определения языков программирования требуют, чтобы индекс массива был целым числом. Компилятор должен сообщить об ошибке, если в качестве индекса массива используется число с плавающей точкой.

1.1.6. Генератор лексических анализаторов Lex

Lex (в более поздних реализациях *Flex*) — программный инструмент, который позволяет определить лексический анализатор, указывая

регулярные выражения для описания шаблонов токенов. Входные обозначения для Lex обычно называют *языком Lex*, а сам инструмент — *компилятором Lex*. Компилятор Lex преобразует входные шаблоны в диаграмму переходов [3].

На рис. 1.3 показанна схема использования Lex. Входной файл `lex.l` написан на языке Lex и описывает генерируемый лексический анализатор. Компилятор Lex преобразует `lex.l` в программу на языке программирования C в файле с именем `lex.yy.c`. Этот файл компилируется компилятором C в файл с названием `a.out`. Результат работы компилятора C представляет собой работающий лексический анализатор, который может получать поток входных символов и выдавать поток токенов [1].

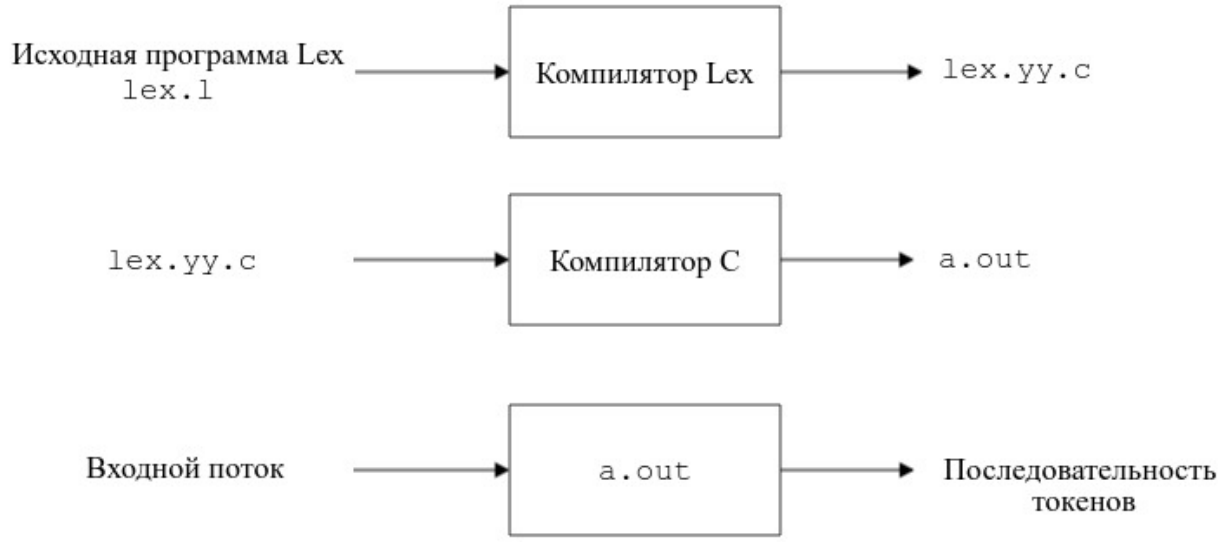

Рис. 1.3. Создание лексического анализатора с помощью Lex

1.1.7. Генератор синтаксических анализаторов Yacc

Yacc — компьютерная программа, служащая стандартным генератором синтаксических анализаторов в Unix-системах. Название является акронимом *«Yet Another Compiler Compiler»* («ещё один компилятор компиляторов») [4]. Yacc генерирует синтаксический анализатор на основе аналитической грамматики, описанной в нотации Бэкуса-Наура или контекстно-свободной грамматики. На выходе Yacc выдаётся код на языке программирования C.

Поскольку синтаксический анализатор, генерируемый с помощью Yacc, требует использования лексического анализатора, то часто он используется совместно с генератором лексических анализаторов, в большинстве случаев это Lex либо Flex.

Создание транслятора с использованием Yacc схематично показано на рис. 1.4. Вначале создаётся файл, например, `translate.y`, содержащий Yacc-спецификацию разрабатываемого транслятора. Затем этот файл преобразуется в программу `t.tab.c` на языке C, которая является синтаксическим анализатором. На вход скомпилированной программе `a.out` поступает поток токенов, а результатом работы становится синтаксическое дерево разбора [1].

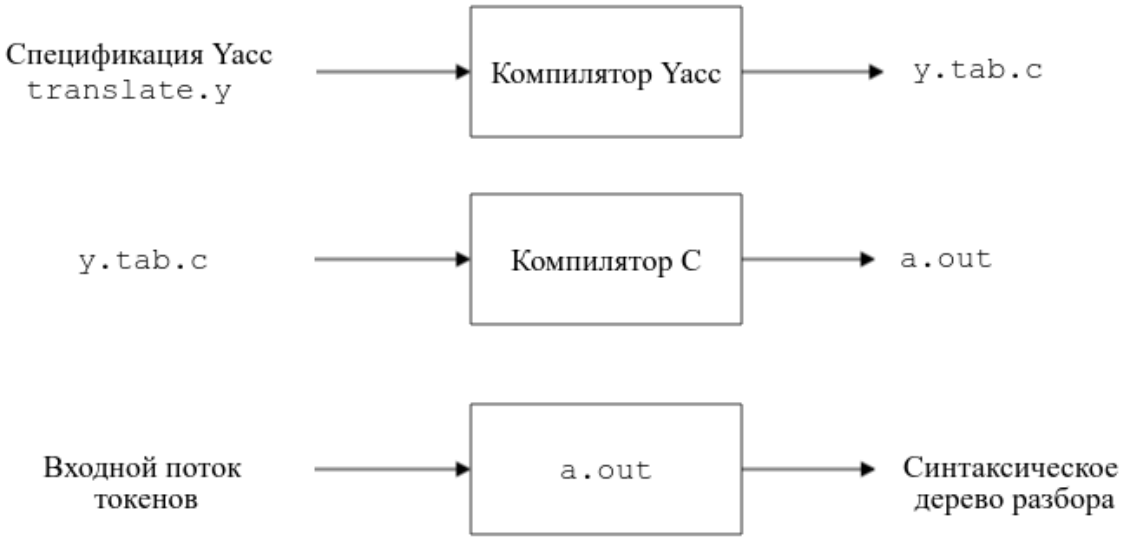

Рис. 1.4. Создание транслятора с помощью Yacc

1.2. Предметно-ориентированные языки программирования

1.2.1. Определение предметно-ориентированного языка программирования

Предметно-ориентированный язык программирования (domain-specific language, DSL) — язык программирования или исполняемая спецификация, которые предлагают выразительное и мощное решение конкретной предметной проблемы с помощью высокоуровневых абстракций и специализированных нотаций. Чаще всего такой язык программирования не является большим и предоставляет только те конструкции, которые необходимы для решения предметной задачи (например, язык Yacc).

Альтернативой предметно-ориентированному языку является использование объектно-ориентированного языка программирования вместе с библиотекой типов и функций, которые отвечают предметным потребностям. [5]

1.2.2. Виды предметно-ориентированных языков программирования

Глобально предметно-ориентированные языки программирования делятся на 2 группы:

– *внешние*
– *внутренние*

Разработка внешних языков состоит из трёх шагов:

– определение семантической модели
– определение синтаксической модели (абстрактный и конкретный синтаксис)
– определение правил трансформации (правила, по которым абстрактное представление транслируется в исполнимое

Для генерации лексического и синтаксического анализатора внешних языков существуют готовые средства, например, связка программ Lex + Yacc, входящая в стандарт POSIX.

Плюсом внешних DSL является узкая специализация, что облегчает процесс решения предметных задач, а так же гибкая базовая грамматика. Но у внешних языков существует и ряд недостатков. Например, среду разработки, которая поддерживала и облегчала бы написание сценариев на внешнем DSL, обычно разрабатывают либо с нуля, либо как дополнение к уже существующей современной интегрированной среде разработки (integrated development environment, IDE). Также, наряду с лёгкостью решения предметных задач, внешние DSL практически никогда не подходят для решения задач в смежных областях.

Альтернативой внешним DSL являются внутренние. Внутренний DSL (embedded language) — это подмножество других языков программирования широкого применения [6]. Такой подход позволяет совместить выразительность и мощь предметно-ориентированного языка вместе с возможностями языков широкого применения (Groovy, Scala, Kotlin, Ruby, Python, C#, F#, Haskell).

Выбирая современный язык программирования общего назначения как основу для создания внутреннего DSL, мы сразу получаем готовый набор

средств поддержки разработки — современные IDE, которые поддерживают базовый язык [7]. Но не во всех случаях такая поддержка является полноценной какая-то подводочка к проблеме плохой подцветки Groovy.

Схема внутренних и внешних DSL

Глава 2. Постановка задачи

Изучить существующие подходы к лексическому и синтаксическому анализу языков программирования с контекстно-свободной грамматикой и динамической компиляцией.

Разработать лексический и синтаксический анализаторы с целью подсветки синтаксиса и авто-дополнения исходного кода для предоставленного языка (предметно ориентированного).

Заключение

Основные результаты работы заключаются в следующем.

1. На основе анализа …
2. Численные исследования показали, что …
3. Математическое моделирование показало …
4. Для выполнения поставленных задач был создан …

И какая-нибудь заключающая фраза.

Список литературы

1. Aho, A. V. Compilers: principles, techniques and tools (for Anna University), 2/e / A. V. Aho. — Pearson Education India, 2003.
2. Earley, J. An efficient context-free parsing algorithm / J. Earley // Communications of the ACM. — 1983. — Т. 26, № 1. — С. 57—61.
3. Lex & yacc / J. R. Levine [и др.]. — 1992.
4. Yacc: Yet another compiler-compiler / S. C. Johnson [и др.]. — 1975.
5. Deursen, A. V. Little languages: Little maintenance? / A. V. Deursen, P. Klint // Journal of Software Maintenance: Research and Practice. — 1998. — Т. 10, № 2. — С. 75—92.
6. Van Deursen, A. Domain-specific languages: An annotated bibliography / A. Van Deursen, P. Klint, J. Visser // ACM Sigplan Notices. — 2000. — Т. 35, № 6. — С. 26—36.
7. Ботов, Д. Обзор современных средств создания и поддержки предметно-ориентированных языков программирования / Д. Ботов // Вестник ЮУрГУ. Серия «Компьютерные технологии, управление, радиоэлектроника». — 2013.