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* [1]. Процесс компиляции можно разделить на две части.

Первая часть — *анализ* (выполняется в несколько фаз) разбивает исходную программу на составные части и накладывает на них грамматическую структуру. Затем эта структура используется для генерации промежуточного представления программы. Анализатор сообщает пользователю об ошибках, собирает информацию об исходной программе и сохраняет 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.

Анализ и синтез делятся на несколько шагов, которые представлены на рис. 1.1.

Анализ:

- *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.

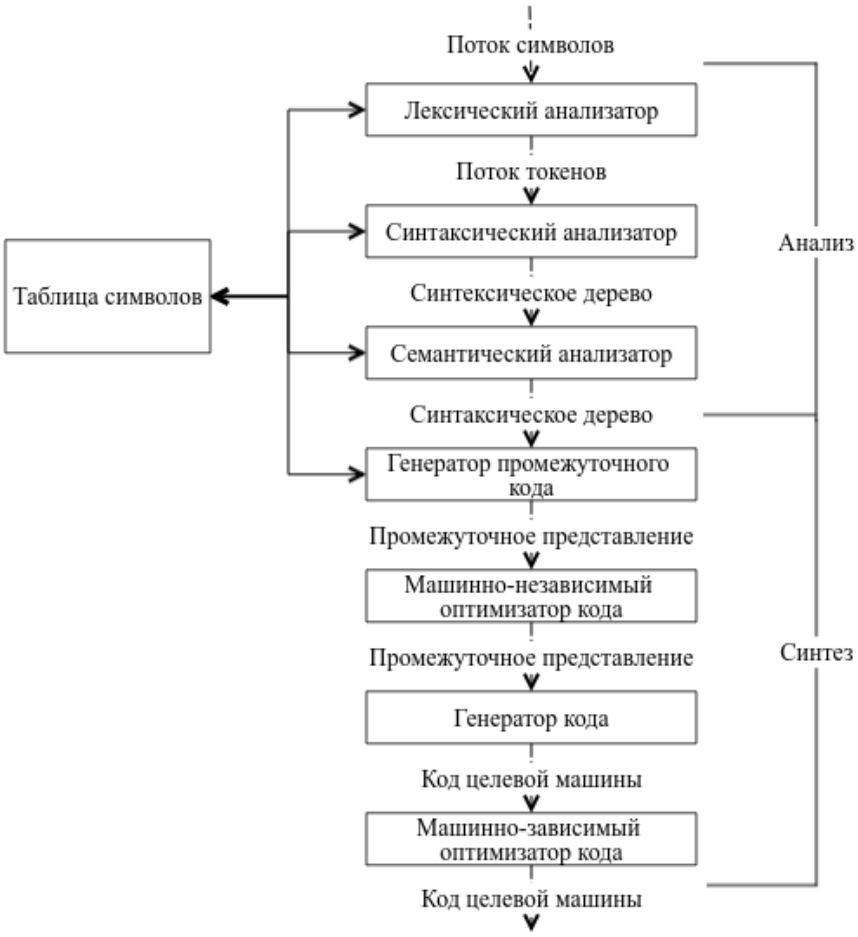

Рис. 1.1. Схема взаимодействия фаз компилятора

– *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.

Синтез:

– *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. Например, заменить вызов метода на непосредственное выполнение тела метода вместе вызова.

– *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.

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 переносимости компилятора. Например, для того, что бы компилятор работал на разных операционных системах достаточно реализовать только часть генерации кода, вместо написания второго компилятора.
3. Разделение зон ответственности. Каждый компонент отвечает только за определённую часть функциональности. Таким образом сокращается вероятность допущения ошибки, код каждого компонента становится легко модифицируемым.

1.1.2. Лексический анализатор

При рассмотрении лексического анализа используются четыре 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. Атрибут токена — строка или структура, объединяющая несколько блоков информации описание лексемы (строчка кода, значение), которая представляет токен.

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⟩

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"`

– *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 (регулярное выражение).
– *Лексема* — последовательность символов исходной программы, которая соответствует шаблону токена и идентифицируется лексическим анализатором как экземпляр токена.

Основная задача лексического анализатора — чтение входных символов исходной программы, их группирование в лексемы и вывод последовательностей токенов для всех лексем исходной программы. Поток токенов пересылается синтаксическому анализатору для разбора. Лексический анализатор удаляет комментарии, пробельные символы, синхронизирует сообщения об ошибках и раскрывает макросы.

1.1.3. Формальное определение контекстно-свободной грамматики

Для работы синтаксического анализатора требуется описание грамматики языка программирования, которое задаётся с помощью контекстно-свободных грамматик (далее КС-грамматики).

Существует множество типов грамматик (грамматики типа 3, контекстно-свободные грамматики, контекстно-зависимые грамматики и грамматики без ограничений), с помощью которых можно описать различные языки. Для описания языков программирования используются КС-грамматики, потому что их разбор наиболее быстрый.

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 выражение `switch` языка `Java`:

$$
\begin{aligned}
SwitchStatement &\rightarrow \textbf{switch (}\ Expression\ \textbf{)}\ SwitchBlock \\
SwitchBlock &\rightarrow \textbf{\{}\ \{\ SwitchBlockStatements\ \}\ \{\ SwitchLabel\ \}\ \textbf{\}} \\
SwitchBlockStatements &\rightarrow SwitchLabels\ BlockStatements \\
SwitchLabels &\rightarrow SwitchLabel\ \{\ SwitchLabel\ \} \\
SwitchLabel &\rightarrow \textbf{case}\ ConstantExpression\ \textbf{:} \\
SwitchLabel &\rightarrow \textbf{case}\ EnumConstantName\ \textbf{:} \\
SwitchLabel &\rightarrow \textbf{default :} \\
EnumConstantName &\rightarrow Identifier \\
ConstantExpression &\rightarrow Expression
\end{aligned}
$$

КС-грамматика состоит из четырёх частей:

1. *Терминалы* — базовые символы, формирующие строки. Термин «имя токена» является синонимом слова «терминал». Пример терминала: `switch`, `case`, `(`, `)`. Другими словами, это листья дерева разбора, рис. 1.2.
2. *Нетерминалы* — синтаксические переменные, которые обозначают множества строк. В примере, приведённом выше, $Expression$, $SwitchBlock$, $SwitchLabel$ и др. 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, определяемых КС-грамматикой. По соглашению, продукции стартового символа указываются первыми (примере это $SwitchStatement$).
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 процессе компиляции синтаксический анализатор получает строку токенов от лексического анализатора и проверяет, может ли эта строка имен токенов соответствовать грамматике исходного языка. Если программа написанна корректно, то синтаксический анализатор строит дерево разбора и передает его следующей части компилятора для дальнейшей обработки. В противном случае синтаксический анализатор сообщает обо всех выявленных ошибках и продолжает работу с оставшейся частью программы.

Имеется три основных типа синтаксических анализаторов грамматик:

– *Универсальные методы разбора*: например, алгоритмы Кока-Янгера-Касами (Cocke-Younger-Kasami) и Эрли (Earley) [2]. Плюсами данного подхода является возможность работать с любой грамматикой. Однако эти обобщённые методы слишком неэффективны для использования в промышленных компиляторах и для реализации подобных алгоритмов требуются тьюринг-полные языки программирования.
– *Восходящие методы разбора* (bottom-up): построение дерева разбора происходит снизу (от листьев) вверх (к корню). Поток токенов сканируется слева направо. Плюсом такого подхода является эффективный расход памяти (для реализации достаточно расширенного автомата с магазинной памятью) и близкая к линейной оценка работы.

– *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. Такие методы используется для полностью ручной реализации, так как являются наиболее естественными для восприятия человеком.

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

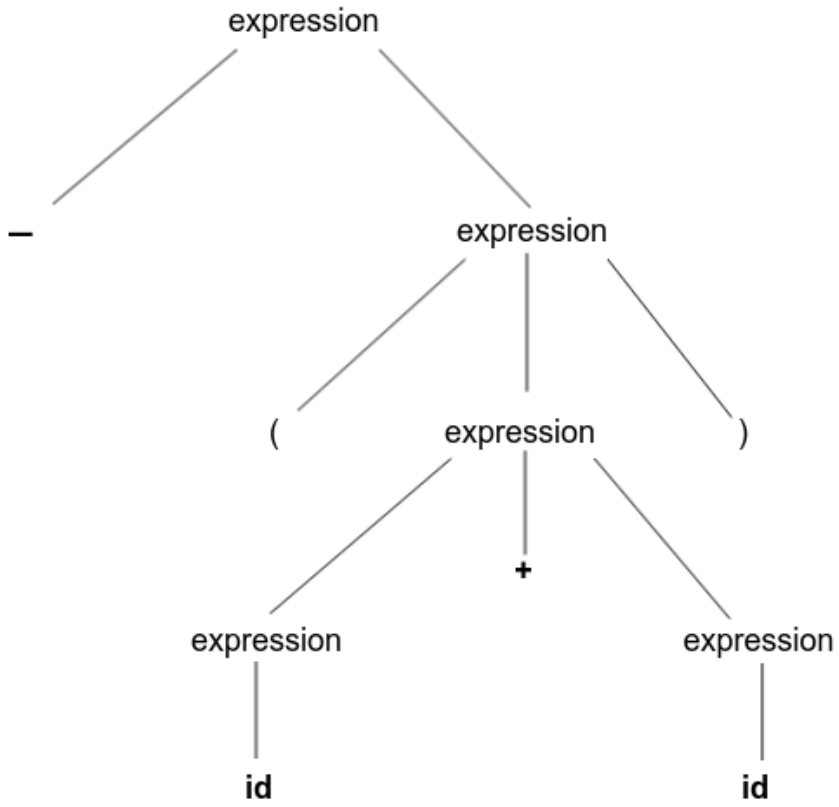

Рис. 1.2. Дерево разбора для выражения $-(\mathbf{id} + \mathbf{id})$

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 неотрицательным 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, реализующий восходящий анализ, который может получать поток входных символов и выдавать поток токенов [1].

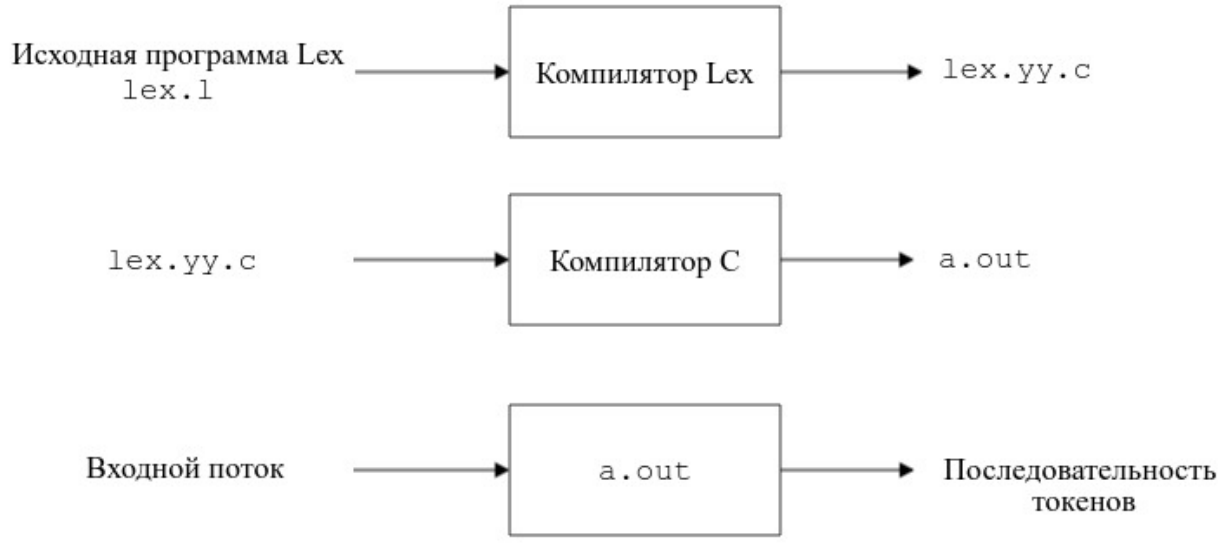

Рис. 1.3. Создание лексического анализатора с помощью Lex

Следующим поколением лексического генератора Lex стал Flex (fast lex). Flex практически полностью совместим с Lex. Отличия Flex:

– может компилироваться в C++, что позволяет использовать объектно-ориентированные конструкции C++;
– генерирует более производительные лексические анализаторы;
– не имеет ограничений по размеру для таблиц символов (в отличие от Lex);
– небольшие синтаксические различия [3].

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, который реализует восходящий тип разбора. На вход скомпилированной программе `a.out` поступает поток токенов, а результатом работы становится синтаксическое дерево разбора [1].

GNU(«GNU is Not Unix»)-альтернативой Yacc служит *Bison*, он практически полностью совместим с Yacc, за исключением небольших синтаксических дополнений и флагов запуска программы [3].

1.1.8. Универсальные генераторы анализаторов

Помимо инструментов, которые генерируют отдельно синтаксический и лексический анализаторы, существуют универсальные генераторы, способные автоматизировать создание лексического и синтаксического анализаторов одновременно.

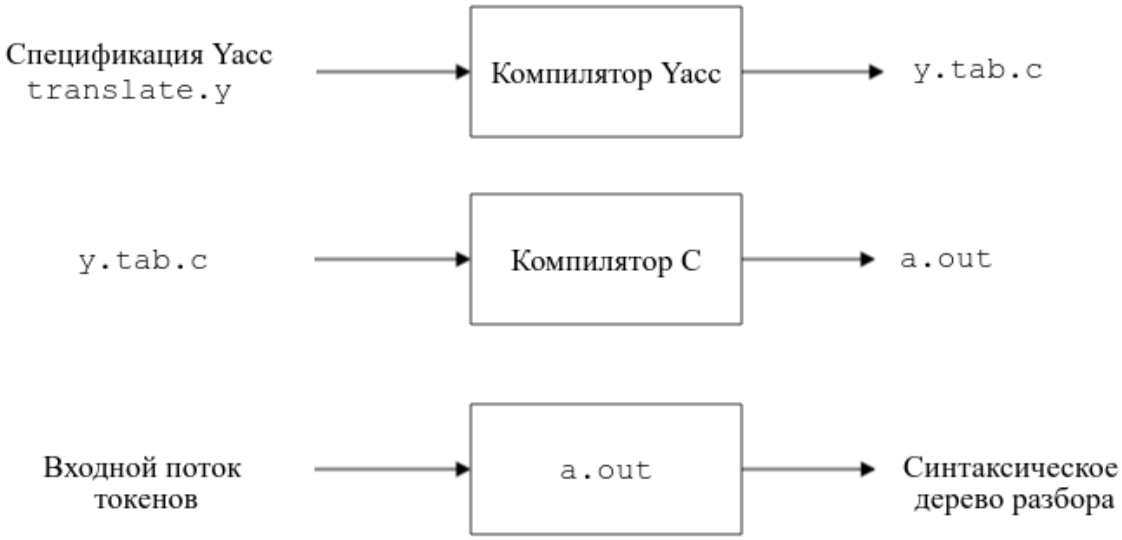

Рис. 1.4. Создание транслятора с помощью Yacc

ANTLR — генератор синтаксических и лексических анализаторов. Реализует алгоритмы нисходящего анализа. Обладает развитыми возможностями по оптимизации таблиц разбора. За счет чего достигается конкурентоспособность в быстродействии конечного продукта против решений построенных на реализации восходящих алгоритмов анализа. На вход принимается описание контекстно-свободной грамматики в Расширенной форме Бекуса-Наура. На выход выдается код синтаксического и лексического анализаторов на языках Java или C++ и других. Активно используется во многих продуктах, таких как среда разработки Eclipse, NetBeans, система баз данных Cassandra и другие.

Coco/R — генератор синтаксических и лексических анализаторов. Реализует алгоритмы нисходящего рекурсивного анализа. На вход принимается описание контекстно-свободной грамматики в Расширенной форме Бекуса-Наура. На выход выдается код синтаксического и лексического анализаторов на языках Java, C++ или других [5].

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 языком общего применения и предоставляет только те конструкции, которые необходимы для решения предметной задачи (например, язык Yacc).

Альтернативой предметно-ориентированному языку можеть стать использование языков программирования общего назначения (объектно-ориентированные) вместе с библиотекой типов и функций, которые отвечают предметным потребностям. [6]

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 [7]. 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 [8]. Подходы к созданию DSL, языки и инструментарий представлены на рис. 1.1.

При разработке внутренних DSL возникают ситуации, когда IDE не может определить некоторые конструкции языка. Это может быть часть DSL, которая будет интерпретированна только во время выполнения программы (Groovy DSL). Или допустимый домен атрибута (переменной) может зависеть от окружения в котором находится содержащий его файл, что значительно усложняет статический анализ, производимый IDE.

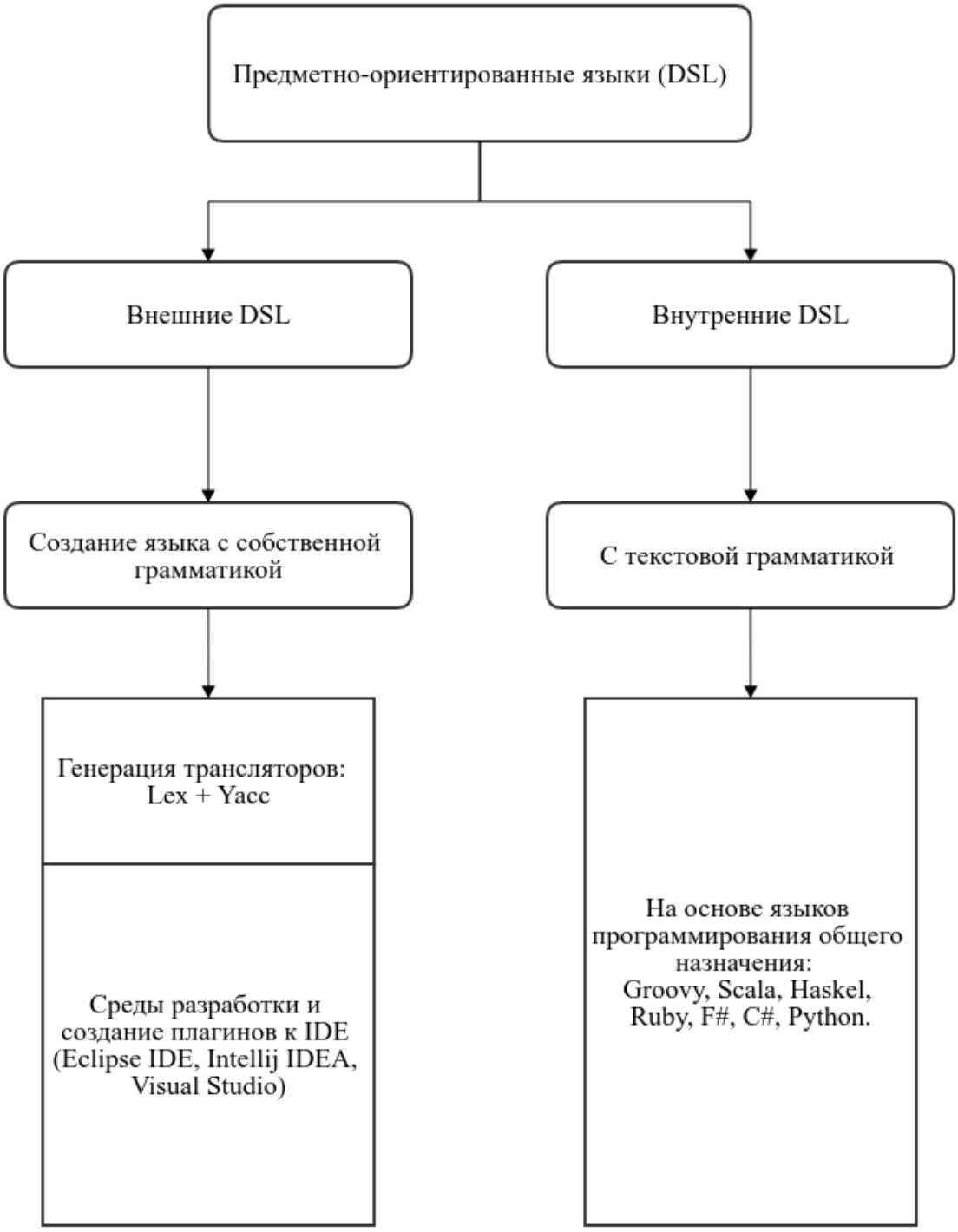

Рис. 1.5. Подходы к созданию внешних и внутренних DSL, языки и инструментарий создания и поддержки DSL

Глава 2. Практическая часть

2.1. Постановка задачи

Целью работы является обеспечение поддержки (подсветки синтаксиса и авто-дополнения исходного кода) предметно-ориентированного языка программирования в интерактивной среде разработки (IDE) Intellij IDEA.

Задачи работы:

1. Изучить существующие подходы к лексическому и синтаксическому анализу языков программирования с контекстно-свободной грамматикой и динамической компиляцией.
2. Разработать лексический и синтаксический анализаторы для подсветки синтаксиса и авто-дополнения исходного кода для предоставленного языка (предметно ориентированного).
3. Интегрировать лексический и синтаксический анализаторы с Intellij IDEA.

Областью исследования является лексический и синтаксический анализ, а также интеграция этих процессов с современной средой разработки.

Предмет исследования — подцветка синтаксиса Groovy DSL в Intellij IDEA.

Разработка полноценного плагина для IDE, самостоятельного компилятора и предметно-ориентированного языка выходят за рамки работы.

2.2. Средства реализации

2.3. Требования к программному и аппаратному обеспечению

Требования к аппаратному и программному обеспечению:

- RAM: 1 Гб минимум, 2 Гб рекомендовано;
- свободное место на диске: 300 Мб + не менее 1 Гб для кэша;
- минимальное разрешение экрана — 1024x768;
- JDK 1.6 и выше;
- Groovy 1.6 и выше;
- Intellij IDEA 9 и выше.

2.4. Реализация

2.5. Интерфейс пользователя

2.6. План тестирования

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. Андреевич, С. Б. СОЗДАНИЕ МЕТОДИКИ РАЗРАБОТКИ ТРАНСЛЯТОРОВ ПРЕДМЕТНО-ОРИЕНТИРОВАННЫХ ЯЗЫКОВ, НА ПРИМЕРЕ ПРОТОТИПА РАСПРЕДЕЛЕННОГО ТРАНСЛЯТОРА ЯЗЫКА ОПИСАНИЯ АРХИТЕКТУРЫ : дис. … канд. / Андреевич Спиридонов Борис. — Санкт-Петербургский политехнический университет Петра Великого, 2017.
6. 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.
7. 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.
8. Ботов, Д. Обзор современных средств создания и поддержки предметно-ориентированных языков программирования / Д. Ботов // Вестник ЮУрГУ. Серия «Компьютерные технологии, управление, радиоэлектроника». — 2013.