Курсовые работы по курсу «Конструирование компиляторов»

Коновалов А. В.

2 октября 2024 г.

Содержание

Темы курсовых работ **2**

1. Троянский конь в самоприменимом компиляторе . . . 2
2. Самоприменимый генератор компиляторов на основе анализа типа «перенос-свёртка» . . . 2
3. Самоприменимый генератор компиляторов методом рекурсивного спуска . 2
4. Самоприменимый компилятор языка Scheme . . . 3
5. Интерпретатор для компилятора P5, использующий JIT . . . 3
6. Суперкомпилятор модельного языка + исследования . . . 4
7. Учебное пособие для изучения КС-грамматик . . . 4
8. Компилятор ISO Паскаля в НУИЯП ~~или ANSI Си~~ . . . 5
9. Компилятор ОО НУИЯПа в реальный ассемблер . . . 5
10. Компилятор подмножества Java в ANSI C . . . 5
11. Утилита проверки расстановки отступов для C/C++ . . . 5
12. Стадия анализа для компилятора Рефала-5J (курсовая + ВКР) . . . 6
13. Компилятор функционального языка программирования . . . 6
14. Компилятор диалекта Бейсика в ассемблер . . . 7
15. Компилятор подмножества Фортрана в ассемблер . . . 7
16. Компилятор подмножества Паскаля в ассемблер MIX Кнута . . . 7
17. Перфектный суперкомпилятор лиспоподобного языка . . . 7
18. Самоприменимый генератор компиляторов для Java рекурсивным спуском 8
19. Компилятор языка L_1–L_4 в ассемблер Microsoft.NET или LLVM . . . 8
20. Компилятор Алгола-60 в ассемблер Microsoft.NET или LLVM . . . 8
21. Перенос компилятора ВТРС(64) на байткод . . . 8
22. Интерпретатор П-кода P5 на Си89 . . . 9
23. Перенос ВТРС64 на Apple ARM . . . 9
24. Перенос ВТРС(64) на JVM . . . 9
25. Модель универсального транслятора, использующего СУО . . . 9
26. Самоприменимый компилятор подмножества Haskell-98 в Java . . . 9

27. Самоприменимый транспилятор подмножества Haskell-98 в Scala 10
28. JIT-компилятор модельного ассемблера РАЯПа для реального процессора . 10
29. Курсовая для Ивана Токарева 10
30. Расширение возможностей библиотеки `parser_edsl` 10
31. Самоприменимый компилятор подмножества Оберона в ассемблер MS-DOS 11
32. Реализация эффективного алгоритма раскраски синтаксиса 11
33. Самоприменимый компилятор Scheme в C89 11
34. Языковой сервер для Рефала-5 12

Приложения **12**
Приложение А. Пример кода на языке из заданий 6 и 13 12
Приложение Б. Образец диалекта Бейсика для задания 14 14
Приложение В. Образец лиспоподобного языка для задания 17 17
Приложение Г. Пример спецификации грамматики для задания 18 18

Темы курсовых работ

1. Троянский конь в самоприменимом компиляторе

Цель работы: продемонстрировать внедрение троянского коня в компилятор BTPC(64)/P5 по выбору студента.

2. Самоприменимый генератор компиляторов на основе анализа типа «перенос-свёртка»

Курсовую работу по этой теме могут делать только те, кто сдал лабораторную работу 3.1!

Цель работы: сделать лабораторную работу 3.1 для распознавателя LR(1). Входной язык тот же, что и в лабораторной работе 3.1, язык реализации по выбору студента.

3. Самоприменимый генератор компиляторов методом рекурсивного спуска

Цель работы: реализовать генератор компиляторов, порождающий из описания грамматики в РБНФ синтаксический анализатор методом рекурсивного спуска.

Генератор должен порождать код на C89 (совместимый C++) и быть совместимым по API с генератором лексических анализаторов Flex (стыковаться со Flex'ом он должен точно также, как и Bison).

Входным языком генератора должно быть описание грамматики в той или иной разновидности РБНФ, содержащее также вставки кода на C/C++. Генератор должен уметь распознавать во входном коде ошибки вроде левой рекурсии или неоднозначного выбора альтернативы.

Целевой код должен генерироваться методом рекурсивного спуска.

4. Самоприменимый компилятор языка Scheme

Цель работы: написать компилятор Scheme в соответствии с описанием, данным в главе «Chapter 4. The stack-based model» диссертации P. Кента Дибвига:

https://www.cs.unm.edu/~williams/cs491/three-imp.pdf

В диссертации Дибвига описывается модель компилятора — компилятор порождает структуры данных на Scheme, которые затем другим кодом на Scheme интерпретируются.

Разрабатываемая система должна быть описана как пара из компилятора, порождающего некоторый промежуточный код, и интерпретатора, этот код выполняющего.

Компилятор:

- Входной язык: подмножество R^5RS Scheme.
- Целевой язык: некий промежуточный код.
- Язык реализации: совпадает со входным языком (самоприменимый).
- Платформа реализации: кроссплатформенное ПО.
- Целевая платформа: интерпретатор промежуточного кода.

Интерпретатор:

- Входной язык: промежуточный код.
- Язык реализации: любой.
- Платформа реализации: кроссплатформенное ПО.

Примечание. В диссертации Дибвига уже приводится исходный код ядра компилятора на каком-то диалекте Scheme. Если реализовать макросы для используемого им синтаксического сахара, то допустимо просто скопировать его код.

5. Интерпретатор для компилятора P5, использующий JIT

Цель работы: ускорить работу интерпретатора компилятора P5, используемого в лабораторных работах.

Входной язык: абстрактный ассемблер на выходе компилятора и на входе интерпретатора P5.

Интерпретатор должен в памяти порождать двоичный код x86-64 и передавать на него управление.

6. Суперкомпилятор модельного языка + исследования

Цель работы: написать модельный суперкомпилятор и провести на нём исследования по суперкомпиляции.

Входной язык: есть два варианта:

1. В качестве входного языка предлагается язык из лабораторной работы № 5 по курсу «Объектно-функциональное программирование»:

   https://hw.iu9.bmstu.ru/submission?task_id=5436

2. Можно реализовывать входной язык из приложения А, удалив из него поддержку целых чисел.

В лабораторной работе требуется только описать синтаксическое дерево в виде case-классов и реализовать некоторые алгоритмы суперкомпиляции. В курсовой работе также требуется реализовать и полноценную стадию анализа (front-end) суперкомпилятора.

Результатом работы суперкомпилятора должен быть граф суперкомпиляции (как минимум), а лучше дамп графа в виде остаточной программы на входном языке.

В работе предлагается сравнить следующие подходы суперкомпиляции:

- перестройка сверху vs перестройка снизу + «гиперцикл Абрамова»,
- проверка на зацикливание по отношению Хигмана-Крускала vs мешок тегов.

Сравнить — т.е. найти примеры программ, на которых разные подходы дают разный результат и проинтерпретировать результаты.

Хорошие методички по суперкомпиляции:

- Анд.В. Климов, С.А. Романенко. Суперкомпиляция: основные принципы и базовые понятия // Препринты ИПМ им. М.В.Келдыша. 2018. № 111. С. 1–36. doi:10.20948/prepr-2018-111 PDF
- С.А. Романенко. Суперкомпиляция: гомеоморфное вложение, вызов по имени, частичные вычисления // Препринты ИПМ им. М.В.Келдыша. 2018. № 209. С. 1–32. doi:10.20948/prepr-2018-209 PDF

7. Учебное пособие для изучения КС-грамматик

Цель работы: написать web-приложение, в котором можно (а) для грамматик и входных цепочек интерактивно строить деревья разбора и (б) производить анализ грамматик.

Для интерактивного построения деревьев разбора следует использовать алгоритм Эрли (поскольку на входе может быть произвольная грамматика), для отображения следует использовать Graphviz (например).

Анализ должен включать в себя проверку на принадлежность классам LL(1) и LR(1), а также проверку на неоднозначность методом грубой силы (генерация входных цепочек по грамматике с поиском равных цепочек с разными деревьями вывода).

8. Компилятор ISO Паскаля в НУИЯП ~~или ANSI Си~~

Цель работы: написать стадию анализа компилятора Паскаля, порождающую на выходе текст на си-подобном языке программирования (низкоуровневом императивном языке программирования, рассмотренном в курсе РАЯП ~~или ANSI Си~~).

Язык реализации может быть любым.

9. Компилятор ОО НУИЯПа в реальный ассемблер

Цель работы: написать стадию синтеза, преобразующую низкоуровневый императивный язык программирования, рассмотренный в курсе РАЯП, в ассемблерный листинг для процессора x86-32, x86-64 или ARM (по выбору студента).

НУИЯП должен содержать одно из объектно-ориентированных расширений из курса РАЯП (так интереснее).

Функции в сгенерированном коде должны придерживаться стандартных соглашений вызова для языка Си, т.е. объектные файлы, полученные из листингов ассемблера, должны компоноваться с кодом на Си.

Язык реализации может быть любым, но в случае языка, отличного от Рефала-5, придётся разрабатывать и стадию анализа (допустимо в этом случае кодировать НУИЯП при помощи JSON или XML).

10. Компилятор подмножества Java в ANSI C

Цель работы: научиться проектировать структуры данных для представления объектно-ориентированных концепций (классы, интерфейсы) в современных ОО-языках; реализовать компилятор подмножества Java в язык Си.

Во входном языке должны поддерживаться классы, интерфейсы, наследование (и классов, и интерфейсов), переопределение методов, проверка и приведение типов и сборка мусора.

Реализовывать вложенные, локальные и безымянные классы не обязательно.

Поддержка обобщённых типов на усмотрение студента.

11. Утилита проверки расстановки отступов для C/C++

Цель работы: написать инструмент, приучающий студентов первого курса к написанию отформатированного кода. Инструмент предполагается внедрить в сервер тестирования.

Стилей оформления кода для си-подобных языков существует много — утилита не должна привязываться к какому-то одному конкретному стилю, а, наоборот, допускать большинство хороших стилей.

Кодировать распознавание и поддержку разных стилей нецелесообразно, целесообразно проверять общие принципы, которые кладутся в основу хороших стилей программирования.

Предполагается, что будут проверяться следующие условия:

- вложенные конструкции имеют больший отступ, чем объемлющие;
- либо конструкция языка программирования (нетерминал грамматики) занимает несколько *целых* строчек текста, либо строчка текста содержит несколько *целых* конструкций;
- отступы в соседних строках должны либо совпадать, либо отступ одной из строк должен быть префиксом другой; отсюда следует, что в одной из строк отступ не может быть сделан пробелами, а в другой — знаками табуляции;
- форматирование не должно зависеть от размера табуляции — недопустимо, когда при одной величине табуляции (скажем, 3 пробела) код смотрится красиво, а при другом (скажем, 5 пробелов) — ровные столбики рассыпаются;
- в исходном тексте нет больших кусков закомментированного кода.

12. Стадия анализа для компилятора Рефала-5J (курсовая + ВКР)

Цель работы: написать реализацию Рефала-5, использующую массивное представление для объектных выражений и сравнить её производительность (асимптотически и абсолютно) с имеющейся реализацией (использующей двунаправленные списки для представления данных).

Спецификация Рефала-5J приложена отдельным файлом.

Задача достаточно объёмная, поэтому делится на две части:

- Стадия анализа, строящая т.н. «льезонное представление» (см. определение в спецификации) с повышением местности.
- Стадия синтеза, которая преобразует льезонное представление в код на Java.

Предлагается первую часть сделать в рамках курсовой, вторую — в рамках ВКР.

Язык реализации — любой.

13. Компилятор функционального языка программирования

Цель работы: написать компилятор простого функционального ЯП с алгебраическими типами данных, сопоставлением с образцом и параметризованными типами.

Синтаксис языка программирования описан на примере в приложении А.

Реализация функционального языка программирования должна представлять собой пару из компилятора входного языка в байткод и интерпретатора этого байткода.

Компилятор и интерпретатор могут быть разными программами и даже могут быть написаны на разных языках. Языки реализации — любые на выбор студента.

14. Компилятор диалекта Бейсика в ассемблер

Цель работы: написать компилятор Бейсика в ассемблер реальной машины (x86-32, x86-64, ARM) или виртуальной (LLVM, CIL[1]) по выбору студента.

Синтаксис диалекта реконструируется из примера кода, приведённого в приложении Б.

Язык реализации — любой на выбор студента.

15. Компилятор подмножества Фортрана в ассемблер

Цель работы: написать компилятор подмножества Фортрана в том объёме, в каком язык описан в книге «Программирование для всех»[2]. Целевым языком должен быть ассемблер реальной машины (x86-32, x86-64, ARM) или виртуальной (LLVM, модельный ассемблер из курса РАЯПа[3]) по выбору студента.

Компилятор должен принимать все примеры кода, приведённые в этой книге.

Расширения языка, описанные в §3.6 — по желанию, полный перечень функций из приложения II делать **не нужно**, достаточно описанных в §2.1.12.

Язык реализации — любой на выбор студента.

16. Компилятор подмножества Паскаля в ассемблер MIX Кнута

Цель работы: написать компилятор Паскаля в ассемблер компьютера MIX[4] [5].

Возможно продолжение на ВКР — оптимизирующий компилятор.

17. Перфектный суперкомпилятор лиспоподобного языка

Цель работы: написать перфектный суперкомпилятор простого функционального языка программирования.

Синтаксис диалекта реконструируется из примера кода, приведённого в приложении В.

Перфектный суперкомпилятор — суперкомпилятор, учитывающий отрицательную информацию, т.е. хранящий в конфигурациях рестрикции-неравенства для параметров.

На функции `is_substring_BARBARA` из приложения суперкомпилятор должен проходить т.н. «КМП-тест», т.е. строить остаточную программу, реализующую эффективный конечный автомат.

---

[1] CIL (common intermediate language) — ассемблер платформы .NET.

[2] Салтыков А.И., Семашко Г.Л. Программирование для всех. — М.: Наука. Главная редакция физико-математической литературы, 1986. — 176 с.

[3] Потребуется дополнить его вещественной арифметикой.

[4] Кнут, Дональд, Эрвин. Искусство программирования, том 1. Основные алгоритмы, 3-е изд. : Пер. с англ. — М. : Издательский дом «Вильямс», 2002. — 720 с. : ил. — Парал. тит. англ. ISBN 5-8459-0080-8 (рус.)

[5] https://gitlab.com/x653/mix-fpga

18. Самоприменимый генератор компиляторов для Java рекурсивным спуском

Цель работы: написать генератор синтаксических анализаторов для Java, строящий по данной спецификации парсер методом рекурсивного спуска.

Синтаксис языка спецификаций реконструируется из примера кода, приведённого в приложении Г.

По спецификации должен строиться абстрактный класс с именем, указанным в секции `%class` и абстрактными методами, указанными в секции `%methods`.

19. Компилятор языка L_1–L_4 в ассемблер Microsoft.NET или LLVM

Цель работы: написать компилятор языка L_i в текст на ассемблере CIL или на языке промежуточного представления[6] LLVM.

При реализации для платформы LLVM должно поддерживаться автоматическое управление памятью при помощи подсчёта ссылок.

20. Компилятор Алгола-60 в ассемблер Microsoft.NET или LLVM

Цель работы: написать компилятор подмножества Алгола-60.

Подмножество должно включать, как минимум, одну из следующих возможностей:

- Вложенные функции.
- Передача параметров по наименованию.
- … (список уточняется)

Возможно продолжение работы на ВКР: объектно-ориентированные расширения из Симулы-67.

21. Перенос компилятора BTPC(64) на байткод

Цель работы: обеспечить переносимость и удобство использования компилятора BTPC, используемого в лабораторных работах, путём компиляции в байткод.

В рамках курсовой нужно разработать двоичный интерпретируемый код, в который будут компилироваться программы на диалекте Паскаля BTPC, а также написать интерпретатор этого байткода на Си89.

Из соображений максимальной переносимости и удобства проведения лабораторных работ, интерпретатор должен быть написан на чистом Си и реализован в виде единого файла `btpcgo.c`.

[6] Фактически, это тоже ассемблер.

22. Интерпретатор П-кода P5 на Си89

Цель работы: обеспечить переносимость и производительность компилятора P5, используемого в лабораторных работах.

В рамках курсовой работы нужно реализовать интерпретатор П-кода компилятора P5 на Си89.

Из соображений максимальной переносимости и удобства проведения лабораторных работ, интерпретатор должен быть написан на чистом Си и реализован в виде единого файла `pint.c`.

23. Перенос BTPC64 на Apple ARM

Цель работы: перенести компилятор BTPC64 на платформу Apple ARM.

Разумеется, взять эту тему может тот, у кого соответствующий компьютер есть.

24. Перенос BTPC(64) на JVM

Цель работы: обеспечить переносимость и удобство использования компилятора BTPC, используемого в лабораторных работах, путём компиляции в байткод JVM.

Компилятор должен порождать двоичный .class-файл, пригодный для выполнения виртуальной машиной Java.

25. Модель универсального транслятора, использующего СУО

Цель работы: написать модельный универсальный транслятор, работающий по спецификации, описывающей произвольное синтаксически управляемое определение.

КС-грамматика на входе может быть произвольной — для построения её дерева вывода следует использовать алгоритм Эрли.

Для правил грамматики можно задавать произвольное количество семантических правил, вычисляющих произвольные атрибуты символов.

Синтаксис спецификации предстоит разработать. Приветствуется написание спецификации в форме EDSL для какого-либо языка программирования.

26. Самоприменимый компилятор подмножества Haskell-98 в Java

Цель работы: написание самоприменимого транспилятора Хаскеля в Java.

При выполнении работы достаточно реализовать только то подмножество, которое необходимо для самоприменения.

Код на целевом языке не предназначен для чтения человеком, за исключением чтения в отладочных целях.

27. Самоприменимый транспилятор подмножества Haskell-98 в Scala

Цель работы: написание самоприменимого транспилятора Хаскеля в Скалу.

При выполнении работы достаточно реализовать только то подмножество, которое необходимо для самоприменения.

Транспилятор должен рассматриваться как средство, используемое для переноса программ с Хаскеля на Скалу — целевой текст должен быть человекочитаем, отображать библиотечные средства Хаскеля в аналогичные библиотечные средства Скалы, в идеале даже сохранять комментарии.

28. JIT-компилятор модельного ассемблера РАЯПа для реального процессора

Цель работы: Написать интерпретатор модельного ассемблера из курса РАЯП, использующий JIT-компиляцию.

Возможно продолжение на ВКР: оптимизации.

29. Курсовая для Ивана Токарева

Цель работы: разработать Go-подобный синтаксис языка программирования и стадию анализа с использованием библиотеки `parser_edsl`, формирующую текст на НУИЯПе в синтаксисе Рефала-5.

НУИЯП в курсе рассматривается как язык промежуточного представления, полученный на выходе стадии анализа. Поэтому при выполнении лабораторных корректность программ проверять не нужно — предполагается, что типы уже проверены на стадии анализа.

Эту самую стадию анализа (включая сам входной язык) и предстоит разработать.

Язык должен быть достаточно «тонким», т.е. без синтаксического сахара, чтобы по тексту было очевидно, какой целевой код от него ожидать. Но при этом довольно безопасным, чтобы не пропускать очевидно некорректные программы: проверять типы и т.д.

За основу предлагается взять синтаксис языка Go, т.к., в отличие от синтаксиса Си, он прозрачнее и проще для анализа.

Язык должен поддерживать возможности не только базового НУИЯПа, но и НУИЯП++, прототипное ООП и динамические переменные. Опционально: расширения из вариантов лабораторных работ.

30. Расширение возможностей библиотеки `parser_edsl`

Цель работы: расширить библиотеку следующими возможностями:

- Поддержка лексических ошибок — функции вычисления атрибутов токенов могут возбуждать библиотечное исключение, которое затем транслируется в синтаксическую ошибку.
- Поддержка режимов разбора «предсказывающий анализ» (только для LL(1)-грамматик) и «алгоритм Эрли».
- Возможность указания приоритета и ассоциативности в стиле Bison'а.

31. Самоприменимый компилятор подмножества Оберона в ассемблер MS-DOS

Допустимо некоторые языковые возможности не реализовывать, например, вложенные процедуры, расширение записей или сборку мусора. Также допустимо ограниченно расширять язык, например, добавить оператор для вызова прерывания 21h.

Допустимо формировать .com-программы и работать только в одном сегменте. 64 Кбайт для такого компилятора должно хватить.

32. Реализация эффективного алгоритма раскраски синтаксиса

Цель работы: реализовать эффективный алгоритм подсветки синтаксиса для сервера тестирования.

На данный момент для сервера тестирования реализована подсветка синтаксиса вводимых листингов (но на сервер на 15.05.2024 не выкачена), однако она не эффективная и кривая — по умолчанию перекрашивает только текущую строку и каждый раз заново её пересканирует.

Возможна реализация работающая за время $O(\log|text| + |upd|)$ — за логарифм от длины всего текста плюс линейное время на перекрашиваемую часть, с минимальными правками DOM браузера. Её надо реализовать.

Реализация основана на представлении текста в виде дерева — структура данных, известная как rope + построении конечного автомата (вернее, автомата Мили), управляющего раскраской синтаксиса. В узлах дерева кэшируются действия строчек (отображения состояний в состояния) и состояния на входе и выходе. При модификации дерева модификация дерева и выделение фрагмента для перераскраски выполняются за логарифмическое время + линейное время на перераскраску найденного фрагмента.

33. Самоприменимый компилятор Scheme в C89

Цель работы: реализовать самоприменимый компилятор подмножества Scheme R^5RS в C89, использующий сборку мусора по поколениям и стек вызовов в качестве нулевого поколения (примерно как в Chicken Scheme).

Подмножество Scheme, по очевидным причинам, не будет включать процедуру `eval`. Также можно не реализовывать те части языка, которые не используются в самом компиляторе (т.е. подмножество, достаточное только для самоприменения).

34. Языковой сервер для Рефала-5

Цель работы: изучить технологию языковых серверов и разработать языковой сервер для Рефала-5.

В современных средах разработки поддержка языка программирования (подсветка синтаксиса, автодополнение, выделение синтаксических ошибок) осуществляется не при помощи плагинов, а при помощи языковых серверов. Следует разобраться в этой технологии и разработать языковой сервер для Рефала-5.

Приложения

Приложение А. Пример кода на языке из заданий 6 и 13

Образец программы на входном языке:

```
<< Объявления типов >>
type [List x]: Cons x [List x] | Nil.
type [Pair x y]: Pair x y.
type [Letter]: A | B | C | D.

<< Объединение двух списков >>
fun (zip [List x] [List y]) -> [List [Pair x y]]:

(zip [Cons x xs] [Cons y ys]) -> [Cons [Pair x y] (zip xs ys)] |
(zip xs ys) -> [Nil].

<< Декартово произведение >>
fun (cart_prod [List x] [List y]) -> [List [Pair x y]]:

(cart_prod [Cons x xs] ys) -> (append (bind x ys) (cart_prod xs ys)) |
(cart_prod [Nil]) -> [Nil].

fun (bind x [List y]) -> [List [Pair x y]]:

(bind x [Cons y ys]) -> [Cons [Pair x y] (bind x ys)] |
(bind x [Nil]) -> [Nil].

<< Конкатенация списков >>

fun (append [List x] [List x]) -> [List x]:

(append [Cons x xs] ys) -> [Cons x (append xs ys)] |
(append [Nil] ys) -> ys.
```

```
<< Расплющивание вложенного списка >>
fun (flat [List [List x]]) -> [List x]:

(flat [Cons [Cons x xs] xss]) -> [Cons x (flat [Cons xs xss])] |
(flat [Cons [Nil] xss]) -> (flat xss) |
(flat [Nil]) -> [Nil].

<< Сумма элементов списка >>
fun (sum [List Int]) -> Int:

(sum [Cons x xs]) -> (add x (sum xs)) |
(sum [Nil]) -> 0.

<< Вычисление полинома по схеме Горнера >>
fun (polynom Int [List Int]) -> Int:

(polynom x [Nil]) -> 0 |
(polynom x [Cons coef coefs]) -> (add (mul (polynom x coefs) x) coef).

<< Вычисление полинома x³+x²+x+1 >>
fun (polynom1111 Int) -> Int:

(polynom1111 x) -> (polynom x [Cons 1 [Cons 1 [Cons 1 [Cons 1 [Nil]]]]]).

<< Функция, заменяющая A на B >>
fun (fab [List Letter]) -> [List Letter]:

(fab [Cons [A] xs]) -> [Cons [B] (fab xs)] |
(fab [Cons x xs]) -> [Cons x (fab xs)] |
(fab [Nil]) -> [Nil].

<< Функция, заменяющая B на C >>
fun (fbc [List Letter]) -> [List Letter]:

(fbc [Cons [B] xs]) -> [Cons [C] (fbc xs)] |
(fbc [Cons x xs]) -> [Cons x (fbc xs)] |
(fbc [Nil]) -> [Nil].

<< Функция, заменяющая A и B на C >>
fun (fabc [List Letter]) -> [List Letter]:

(fabc xs) -> (fbc (fab xs)).
```

Опеделения языка программирования начинаются с ключевого слова (`type` и `fun`, соответственно), за которыми идёт заголовок (имя типа и сигнатура функции), двоеточие и набор вариантов (альтернатив для типов и предложений для функций), разделённых знаком вертиальной черты. Определения завершаются точкой.

Вызовы функций записываются в круглых скобках (после открывающей скобки записывается имя функции), конструкторы данных — в квадратных скобках (после открывающей, соответственно, имя конструктора).

Сигнатура функции записывается как вызов функции (т.е. при помощи круглых скобок), в котором вместо аргументов записываются их типы, и тип возвращаемого значения, между которыми записывается знак →.

Предложение записывается как образец и выражение, разделённые знаком →. Образец записывается как вызов функции (т.е. при помощи круглых скобок), на месте аргументов находятся их образцы.

Приложение Б. Образец диалекта Бейсика для задания 14

Диалект Бейсика

```
' Суммирование элементов массива
Function SumArray#(Values#(ValCount%))
  SumArray# = 0
  For i% = 1 To ValCount%
    SumArray# = SumArray# + Values#(i%)
  Next i%
End Function

' Вычисление многочлена по схеме Горнера
Function Polynom!(x!, coefs!(n%))
  Polynom! = 0
  For i% = 1 to n%
    Polynom! = Polynom! * x! + coefs!(i%)
  Next i%
End Function

' Вычисление многочлена x³ + x² + x + 1
Function Polynom1111!(x!)
  Dim coefs!(4)

  For i% = 1 To 4
    coefs!(i%) = 1
  Next i%

  Polynom1111! = Polynom!(x!, coefs!)
```

```
End Function

' Инициализация массива числами Фибоначчи
Sub Fibonacci(res&(n%))
  If n% ⩾ 1 Then
    res&(1) = 1
  End If

  If n% ⩾ 2 Then
    res&(2) = 1
  End If

  i% = 3
  Do While i% ⩽ n%
    res&(i%) = res&(i% - 1) + res&(i% - 2)
    i% = i% + 1
  Loop
End Sub

' Склеивание элементов массива через разделитель: Join$(", ", words)
Function Join$(sep$, items$(count%))
  If count%% ⩾ 1 Then
    Join$ = items$(1)
  Else
    Join$ = ""
  End If

  For i% = 2 To count%
    Join$ = Join$ + sep$ + items$(i%)
  Next i%
End Function

' Главная процедура
Sub Main(argv$(argc%))
    Print "Аргументы программы: ", Join$(", ", argv$)

    ' Объявление локального массива
    Dim fibs&(100)
    Fibonacci(fibs&)
    Print "50-е число Фибоначчи — ", fibs&(50)

    Dim ys#(101)
    Print "Таблица значений функции y = x³ + x² + x + 1:"
```

```
    For x% = -50 To 50
        y! = Polynom1111!(x%)
        Print "x = ", x%, ", y = ", y!
        ys#(x% + 51) = y!
    Next x%

    Print "Сумма перечисленных значений y: ", SumArray#(ys#)
End Sub
```

Комментарии начинаются с апострофа `'` и продолжаются до конца строки.

Идентификаторы и ключевые слова не чувствительны к регистру.

В имени каждой переменной и каждой функции указывается её тип (`%` — целое, `&` — длинное целое, `!` — вещественное одинарной точности, `#` — вещественное двойной точности, `$` — строка).

Внутри функции неявно объявляется переменная с тем же именем, что и имя самой функции — её значение является возвращаемым значением функции.

И индексация массивов, и вызов функции записываются при помощи круглых скобок — отличить одно от другого на этапе синтаксического анализа невозможно, поэтому в дереве они различаться не должны.

Цикл с условием может быть записан пятью способами:

```
Do While …      Do Until …      Do              Do                  Do
  …               …               …               …                   …
Loop            Loop            Loop While …    Loop Until …        Loop
```

Первые две формы — циклы с предусловием (положительным и отрицательным), две другие — с постусловием и, наконец, пятая — бесконечный цикл.

Цикл `For` можно прерывать операторами

```
Exit For
Exit For i%
```

Первая форма прерывает текущий цикл, вторая (с переменной) позволяет прервать сразу несколько вложенных циклов.

Цикл `Do`/`Loop` можно прервать любым из двух операторов (они синонимы):

```
Exit Do
Exit Loop
```

Процедуры и функции прерываются, соответственно, операторами

```
Exit Sub
Exit Function
```

Можно определять глобальные переменные:

```
Dim some_global_var%
Dim some_global_array%(100)
```

Можно определять многомерные массивы:

```
Dim matrix!(100, 100)
```

Приложение B. Образец лиспоподобного языка для задания 17

Статически типизированный функциональный язык программирования:

```
-- Объединение двух списков
zip(xs : [sym], ys : [sym]) : [(sym, sym)] =
  if null(xs) or null(ys) then []
  else cons((car(xs), car(ys)), zip(cdr(xs), cdr(ys)));

-- Декартово произведение
cart_prod(xs : [sym], ys : [sym]) : [(sym, sym)] =
 if null(xs) then [] else append(bind(car(xs), ys), cart_prod(cdr(xs), ys));

bind(x : sym, ys : [sym]) : [(sym, sym)] =
  if null(ys) then [] else cons((x, car(ys)), bind(x, cdr(xs)));

-- Конкатенация списков пар
append(xs : [(sym, sym)], ys : [(sym, sym)]) : [(sym, sym)] =
  if null(xs) then ys else cons(car(xs), append(cdr(xs), ys);

-- Расплющивание вложенного списка
flat(xss : [[sym]]) : [sym] =
  if null(xss) then [] else append(car(xss), flat(cdr(xss)));

-- Проверка префикса
is_prefix_of(xs : [sym], ys : [sym]) : bool =
  null(xs)
  or (not null(ys)
      and car(xs) == car(ys)
      and is_prefix_of(cdr(xs), cdr(ys)));

-- Поиск подстроки в строке
is_substring(needle : [sym], haystack : [sym]) : bool =
  is_prefix_of(needle, haystack) or is_substring(needle, cdr(haystack));

-- Поиск подстроки BARBARA
is_substring_BARBARA(text : [sym]) : bool =
```

```
  is_substring(['B, 'A, 'R, 'B, 'A, 'R, 'A], text);
```

Язык поддерживает символы, логические значения, списки и кортежи (длиной не менее чем 2). Тип списка записывается в квадратных скобках, тип кортежа — в круглых скобках через запятую. Символы записываются как идентификаторы, предварённые апострофом (`'symbol`).

Тело функции состоит из единственного выражения, которое может быть условным выражением, проверкой на равенство, логической операцией (двуместные `or`, `and`, одноместная `not` с обычным приоритетом, приоритет ниже, чем у арифметических операций) вызовы функций, литералы кортежей (несколько выражений в круглых скобках через запятую) и литералы списков (несколько, в том числе и ноль, выражений в квадратных скобках через запятую).

Тело функции завершается точкой с запятой.

Комментарии начинаются с двух дефисов и продолжаются до конца строки.

Приложение Г. Пример спецификации грамматики для задания 18

Язык спецификации для генератора синтаксических анализаторов:

```
%class
  SimpleImperativeLang

%tokens
  NUMBER PLUS MINUS STAR FRAC LBRAC RBRAC
  TRUE FALSE ADD OR NOT LT GT LE GE NE EQ
  IF THEN ELSE END WHILE DO SEMICOLON
  VAR ASSIGN INPUT PRINT COMMA

%types
  Expr, Term, Factor, NUMBER: ArithmExpr;
  PLUS, MINUS, STAR, FRAC: ArithmOp;
  BoolExpr, BoolTerm, BoolFactor, TRUE, FALSE: BoolExpr;
  LT, GT, LE, GE, NE, EQ: RelaOp;
  Program, Statement, StatementList, Program: Statement;
  VAR, STRING: String;
  PrintItem: PrintItem;

%methods
  ArithmExpr neg_op(ArithmOp, ArithmExpr);
  ArithmExprChunk chunk(ArithmOp, ArithmExpr);
  ArithmExpr bin_op(ArithmExpr, ArithmExprChunk[]);
  ArithmExpr deref(String);
```

```
  BoolExpr rela_op(ArithmExpr, RelaOp, ArithmExpr);
  BoolExpr disj_op(BoolExpr, BoolExpr[]);
  BoolExpr conj_op(BoolExpr, BoolExpr[]);
  BoolExpr not_op(BoolExpr);

  Statement assign_stmt(String, ArithmExpr);
  Statement append(Statement, Statement);
  $ для упрощения описания языка считаем последовательность операторов
  $ оператором
  Statement compound(Statement, Statement[]);
  Statement if_else_stmt(BoolExpr, Statement, Statement);
  Statement empty_stmt();
  Statement while_stmt(BoolExpr, Statement);
  Statement input_stmt(String, String[]);

  PrintItem print_value(ArithmExpr);
  PrintItem print_string(String);
  Statement print_stmt(PrintItem, PrintItem[]);

%grammar
  Program = StatementList;

  StatementList = Statement %rep (SEMICOLON Statement) / compound;

  Statement =
      VAR ASSIGN Expr / assign_stmt
      $ Ветка else может отсутствовать
    | IF BoolExpr THEN StatementList (/ empty_stmt | ELSE StatementList) END
      / if_else_stmt
    | WHILE BoolExpr DO StatementList END / while_stmt
    | INPUT VAR %rep (COMMA VAR) / input_stmt
    | PRINT PrintItem (COMMA PrintItem) / print_stmt
    ;

  PrintItem = Expr / print_value | STRING / print_string;

  BoolExpr = BoolTerm %rep (OR BoolTerm) / disj_op;
  BoolTerm = BoolFactor %rep (AND BoolFactor) / conj_op;
  BoolFactor =
      TRUE | FALSE
    | Expr RelaOp Expr / rela_op
    | NOT BoolFactor / not_op
    | LBRAC BoolExpr RBRAC
```

```
    ;

  $ Первому терму в выражении может предшествовать знак минус
  Expr = (Term | MINUS Term / neg_op) %rep ((PLUS | MINUS) Term / chunk)
      / bin_op;
  Term = Factor %rep ((STAR | FRAC) Factor / chunk) / bin_op;
  Factor = NUMBER | VAR / deref | LBRAC Expr RBRAC;

%axiom
  Program

%end
```

Комментарии начинаются на знак доллара и продолжаются до конца строки.

Имена терминалов, нетерминалов, типов и методов — идентификаторы, начинающиеся с буквы и состоящие из букв, цифр и знаков прочерка.

Секция методов содержит сигнатуры методов в стиле языка Java без указания имён параметров. Типы могут быть простыми или массивами.

Каждое правило грамматики состоит из нескольких альтернатив. Альтернатива может завершаться необязательным именем метода, предварённого знаком дробной черты, который вызывается после её разбора.

- Для символов грамматики, определённых в секции `%tokens`, не должно быть правил в секции `%grammar`, т.к. они терминальные символы.
- Для остальных (т.е. нетерминальных) символов должно быть ровно одно правило в секции `%grammar`.
- Символ в секции `%types` может встречаться не более одного раза, если символ встречается, то будем называть его *типизированным*, иначе — *нетипизированным*.
- Если альтернатива заканчивается на дробь и имя метода, то типы типизированных элементов альтернативы должны совпадать с типами аргументов метода, тип альтернативы будет совпадать с типом возвращаемого значения метода (если `void` — нетипизированная).
- Если альтернатива не заканчивается на дробь и имя метода, то в ней должно быть не более одного типизированного элемента:
  - если таковой один, то его тип будет типом альтернативы,
  - если таковых нет, то альтернатива будет нетипизированной.
- Все альтернативы в правиле должны иметь одинаковый тип, либо все быть нетипизированными.
- Тип элемента альтернативы, предварённый префиксом `%rep`, является массивом.
- В секции `%methods` имена методов не могут повторяться.
- Все методы, вызываемые из правил, должны быть определены в секции `%methods`, все определённые методы должны хотя бы один раз вызываться в правилах.