ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ И НАУКИ
САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
АКАДЕМИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Ж.И. АЛФЕРОВА
РОССИЙСКОЙ АКАДЕМИИ НАУК

«_____» _____20____г.
Зав. каф. Биоинформатики и
математической биологи
_____ к.ф.-м.н. К.В. Вяткина

Методы и инструменты эффективного конструирования языков предметных областей

Магистерская диссертация

Направление 03.04.01 Прикладные математика и физика

Орищенко Александра Олеговна

Научный руководитель Ф.А.Новиков
д-р техн. наук

Рецензент Л.Н. Федорченко
канд. техн. наук

Рецензент И.В. Афанасьева
канд. техн. наук

Студент гр. 605 . А.О. Орищенко

Санкт-Петербург, 2020

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ И НАУКИ
САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
АКАДЕМИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Ж.И. АЛФЕРОВА
РОССИЙСКОЙ АКАДЕМИИ НАУК

НАПРАВЛЕНИЕ 03.04.01 Прикладные математика и физика

ПРОФИЛЬ Математические и информационные технологии

ЗАДАНИЕ на выполнение диссертации на соискание академической степени магистра

студенту

Орищенко Александре Олеговне

Тема МД Методы и инструменты эффективного конструирования языков предметных областей

Срок сдачи студентом МД: 15.06.2020

Содержание МД:

В рамках магистерской диссертации:

1) предложен новый язык программирования для моделирования взаимодействия автоматных объектов CIAOv2.
2) разработано программное решение инструментальной поддержки конструирования языков предметных областей

Объем МД __65__ страниц, __1___прил., ____0___таблиц, _ _25___ рис.

Дата выдачи задания «08» февраля 2020 г.

Руководитель: Новиков Фёдор Александрович

Задание принял к исполнению: Орищенко Александра Олеговна

Оглавление

Введение

Создание программного обеспечения необходимо во множестве отраслей промышленности и экономики. Разработка бизнес-приложений связана с решением задач в различных предметных областях, например таких как хранение данных или проектирование интерфейса пользователя. Для разработки приложений можно воспользоваться языками программирования общего назначения. Такой универсальный инструмент рассчитан для решения многих задач, однако он не является оптимальным для каждой конкретной задачи, особенно в случае, когда она нестандартна и имеет много индивидуальных особенностей. Также при программировании с помощью языков общего назначения возникает шаблонный код, который будет повторяться из раза в раз и который не несет в себе никакой специфической логики. Программисту приходится писать этот код каждый раз, как правило, занимая ненужное место, загромождая программу и отвлекая внимание от бизнес-логики.

В отличие от языков программирования общего назначения, языки предметных областей (Domain-Specific Language — DSL) — это языки программирования с более высоким уровнем абстракции, которые отражают специфику решаемых с их помощью задач [**Error! Reference source not found.**] и позволяют формулировать решение целевой задачи в терминах предметной области. Такие языки оперируют понятиями и правилами из определенной предметной области. Преимущества использования DSL заключаются в существенном снижении денежных и временных затрат на разработку. В некоторых случаях, благодаря снижению сложности, разрабатывать ПО могут люди, поверхностно знающие программирование. Подход, в основе которого лежит идея создания языка, специально разработанного под поставленную задачу, называется языково-ориентированное программирование (Language-oriented programming — LOP)[**Error! Reference source not found.**].

При всех достоинствах DSL у них есть один большой недостаток – сложность разработки. Если языки общего назначения позволяют создавать программы безотносительно предметной области, то в случае DSL для каждой предметной области, а иногда даже для каждой конкретной задачи приходится создавать свой предметно-ориентированный язык. Если предметная область достаточно проста и язык несложен, то создать транслятор будет нетрудно. Для более сложной предметной области создание языка потребуют больших усилий, несмотря на то, что в настоящее время существуют генераторы лексический и синтаксических анализаторов, и другие инструменты для создания компиляторов, облегчающие работу программиста.

Развитие в области языково-ориентированного программирования привело к созданию языковых инструментальных средств [3]. Под инструментарием понимается набор программ, который предоставляет разработчику возможности по созданию и использованию предметно-ориентированных языков. Одна из основных задач языкового инструментария – упрощение процесса обучения при создании и освоении предметно-ориентированных языков.

Целью работы является разработка программного решения инструментальной поддержки конструирования DSL для использования студентами, а также выработка методики конструирования DSL, позволяющая легко создавать и модифицировать языки.

Обзор

Описание DSL

Предметно-ориентированный язык – это язык программирования с ограниченными выразительными возможностями, ориентированный на решение задач в некоторой конкретной предметной области. Сильная связь предметно-ориентированных языков с предметной областью позволяет разработчику оперировать понятиями, близкими к этой предметной области, выражать решения прикладных задач в более естественном виде, тем самым заметно упрощая написание сложных программ.

DSL, как и любой другой язык программирования, является средством общения между программистом и компьютером.

Специалисты по языкам программирования обычно выделяют следующие составляющие языка[4]:

- синтаксис — совокупность правил некоторого языка, которые определяют, как будут сформированы его элементы. Синтаксис может быть задан с помощью правил, которые описывают понятия некоторого языка.
- семантику — правила и условия, определяющие соотношения между элементами языка и их смысловыми значениями, а также интерпретацию содержательного значения синтаксических конструкций языка. Объекты языка программирования не только размещаются в тексте в соответствии с некоторой иерархией, но и дополнительно связаны между собой посредством других понятий, образующих разнообразные ассоциации.
- прагматику — связь программы с ее конкретной реализацией.

Таким образом, для описания DSL потребуется задать синтаксис и семантику. Далее разберем основные подходы к формализации синтаксиса и семантики языка.

Описание синтаксиса

Использование грамматики в задании синтаксиса обеспечивает значительное преимущество разработчикам языков программирования и создателям компиляторов [**Error! Reference source not found.**]. Грамматика дает точную и при этом простую для понимания синтаксическую спецификацию языка программирования. Также хорошо построенная грамматика придает языку программирования структуру, которая способствует облегчению трансляции исходной программы в корректный код и выявлению ошибок.

Один из наиболее распространенных способов описания синтаксиса языка – это форма Бэкуса-Наура. Этот способ был разработан для описания Алгола-60, однако, в дальнейшем он использован для многих других языков. Существует также расширенная форма Бэкуса-Наура. Главное преимущество РБНФ перед БНФ — возможность описывать простые повторяющиеся конструкции неопределённой длины (списки, строки, последовательности и так далее) без рекурсивных правил, что делает запись в РБНФ одновременно и короче, и удобнее для восприятия человеком. Естественной платой за преимущества РБНФ перед БНФ является увеличение сложности автоматической интерпретации его описаний.

Стандартная запись расширенной формы Бэкуса-Наура состоит из следующих частей [6]:

- терминальные символы. Такие символы существуют в виде предопределенных идентификаторов или последовательностей символов в кавычках
- нетерминальные символы. Такие символы обычно состоят из других нетерминальных или терминальных символов.

Задание нетерминалов происходит через правила грамматики

- правила грамматики. Правило состоит из двух частей: нетерминал в левой части и выражение – в правой. Правила сопоставляют нетерминальным символам некоторые выражения. Запись:

  ```
  НЕТЕРМИНАЛ := ВЫРАЖЕНИЕ
  ```

- выражение конкатенации. Такое выражение обозначает, что нетерминал в левой части правила соответствует объединению двух символов, указанных в конкатенации через запятую. Запись:

  ```
  A = B,C
  ```

- выражение выбора. Такое выражение обозначает, что нетерминал в левой части правила соответствует или одному или другому символу, указанному в выборе. Запись:

  ```
  A = B|C
  ```

- выражение условия. Такое выражение обозначает, что нетерминал в левой части правила либо пустой, либо соответствует символу в правой части. Запись:

  ```
  A = [B]
  ```

- выражение повторения. Такое выражение обозначает, что нетерминал в левой части правила либо пустой, либо соответствует любому количеству подряд идущих символов из правой части. Запись:

  ```
  A = {B}
  ```

- выражение группировки. Такое выражение используется для формирования более сложных выражений. Запись:

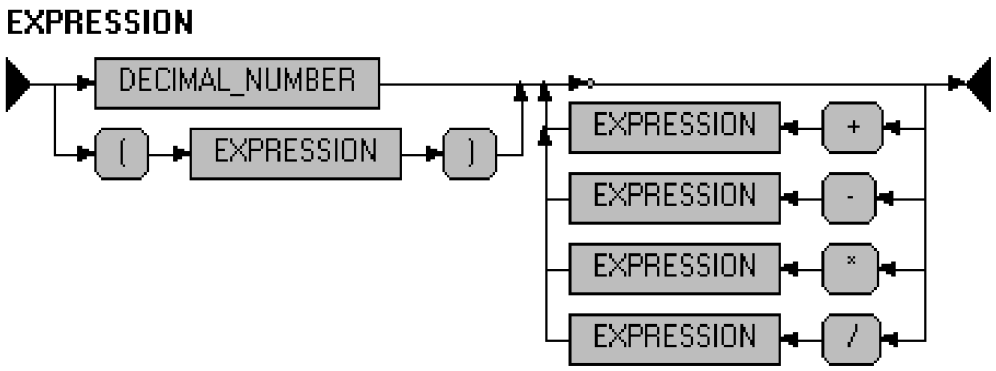
  ```
  A = (A B)
  ```

Итоговая грамматика – список правил. Каждое правило последовательно определяет все нетерминальные символы грамматики.

Таким образом, каждый нетерминальный символ сводится к комбинации терминальных символов путём ступенчатого применения правил. РБНФ никак не регламентирует последовательность записи правил. Однако, такие правила могут быть заданы программным инструментом, который генерирует синтаксический анализатор по заданной грамматике.

Задать синтаксис языка можно также через синтаксические диаграммы [7].

Синтаксическая диаграмма — это направленный граф с одним входным ребром и одним выходным ребром и помеченными вершинами. Представление грамматики состоит из набора синтаксических диаграмм. Каждая диаграмма определяет нетерминал. Существует главная диаграмма, которая определяет язык следующим образом: чтобы принадлежать языку, слово должно описывать путь в главной диаграмме. Каждая диаграмма имеет точку входа и конечную точку. Диаграмма описывает возможные пути между этими двумя точками, проходя через другие нетерминалы и терминалы. При этом терминалы представлены круглыми ячейками, а нетерминалы – прямоугольными. Таким образом, цепочка пометок при вершинах на любом пути от входного ребра к выходному — это цепочка языка, задаваемого синтаксической диаграммой. Пример синтаксический диаграммы можно рассмотреть на рисунке 1.



Описание семантики

В работе [**Error! Reference source not found.**] рассматриваются три основных подхода к формализации семантики:

1) Операционная семантика специфицирует поведение языка программирования, определяя для него абстрактную машину. Состояние машины представляет собой значение текущего

Рисунок 1. Пример синтаксической диаграммы для нетерминала EXPRESSION [8]

состояния, а поведение ее определяется функцией перехода, которая для каждого состояния либо указывает следующее состояние, произведя шаг вычисления (упрощения), либо объявляет машину остановившейся.

2) Денотационная семантика рассматривает значение с более абстрактной точки зрения: в качестве значения принимается не последовательность машинных состояний, а некоторый математический объект, например, число или функция. Построение денотационной семантики для языка состоит в нахождении некоторого набора семантических доменов, а также определении функции интерпретации, которая ставит элементы этих доменов в соответствие выражениям.
3) Аксиоматическая семантика предполагает более прямой подход к этим законам: вместо того, чтобы сначала определить поведение программ (с помощью операционной или денотационной семантики), а затем строить законы, соответствующие этому поведению, аксиоматические методы принимают сами законы в качестве определения языка. Значение выражения — это то, что о нем можно доказать.

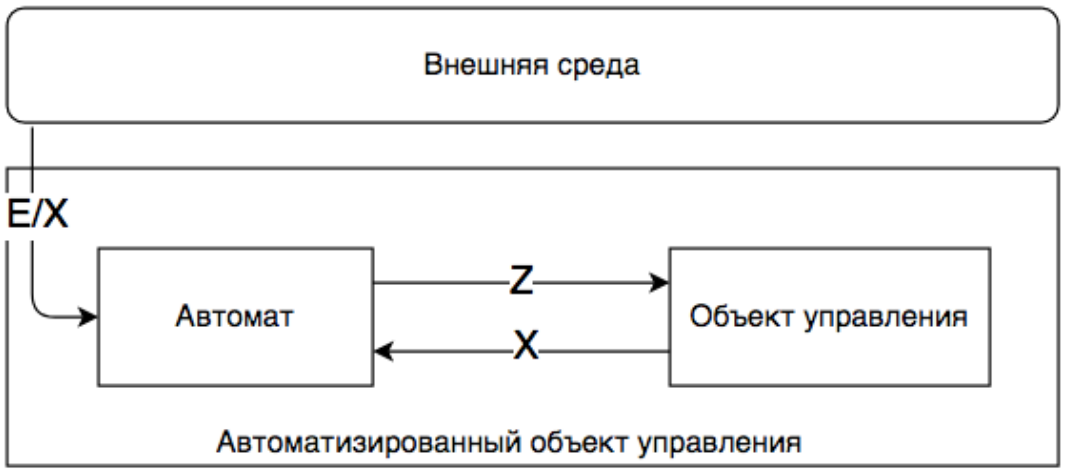
В качестве основного подхода описания семантики для разрабатываемого в работе инструмента задания DSL был выбран

операционных подход, манипулирующий автоматами.

SWITCH-технология

В работе [9] был предложен подход к разработке программных систем со сложным поведением, который основан на модели автоматизированного объекта управления (расширении конечного автомата). Данный подход известен как парадигма автоматного программирования или, иначе, SWITCH-технология.

Такой способ описания программ основывается на базовом понимании автоматизированного объекта. Данный объект состоит из: системы управления (СУ) и объекта управления (ОУ). Система управления описывается конечным автоматом или несколькими взаимодействующими друг с другом автоматами. Важно уточнить, что также в модели существуют источники событий. Система управления может получать события от источника, а также запрашивать некоторые данные от объекта управления и далее, если полученные данные удовлетворяют некоторым входным условиям, совершать переход в новое состояние. Такая система, реагирующая на события, называется «реагирующая» («reactive»).



Рассмотрим более подробно схему работы реагирующей системы. На вход автомату может поступить событие от источника событий. Далее автомат может перейти в следующее состояние по переходу, помеченным данным событием. Однако переход может произойти только при выполнении определенных условий, данные для которых запрашиваются у объектов управления. При переходе по событию, а также при входе в состояние, автомат может воздействовать на объекты управления,

Рисунок 2. Взаимодействие компонентов модели автоматизированного объекта

выполняя набор определенных действий.

Ниже приведен пример работы счетного триггера с включением/выключением лампочки. Такой триггер состоит из кнопки (источник событий), лампочки (объект управления) и управляющего автомата (системы управления). Управляющий автомат в данном случае имеет одну входную переменную x, описывающую состояние кнопки, если нажата, то единица, если отжата, то ноль, и одну выходную переменную z, отвечающей на вопрос: «что нужно сделать с лампочкой?» – единица, если лампочку требуется включить, и ноль – если выключить.

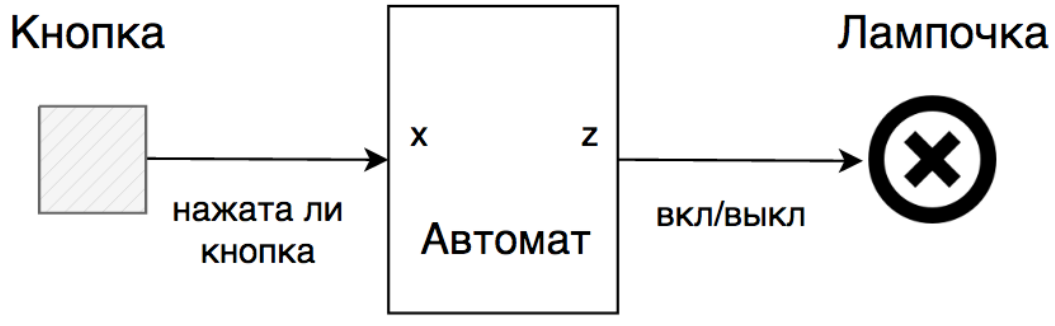


Рисунок 3. Счетный триггер

Алгоритм работы счетного триггера таков: при нажатой кнопке лампочка горит, а после того, как кнопку отжали, лампочка гореть

перестает. Далее в основной части работы будет рассмотрен данный пример для демонстрации работы разработанного инструмента.

Обзор инструментальных средств

В данном разделе рассмотрим программные продукты, которые позволяют проектировать и моделировать DSL.

MPS (Meta Programming System)

Языковой инструментарий — это программное обеспечение, предназначенное облегчить процесс создания новых компьютерных языков и средств написания программ на этих языках.

Такой известный языковой инструментарий — система с открытым исходным кодом Jetbrains Meta Programming System (MPS) [28]. Он предназначен для разработчиков программного обеспечения. В нем конечным пользователям DSL предлагается взаимодействовать с абстрактным синтаксическим деревом (АСД) через проекционные редакторы (projectional editor). На рисунке 4 представлен пример графической проекции матричного выражения, встроенной в исходный код программы на языке Java.

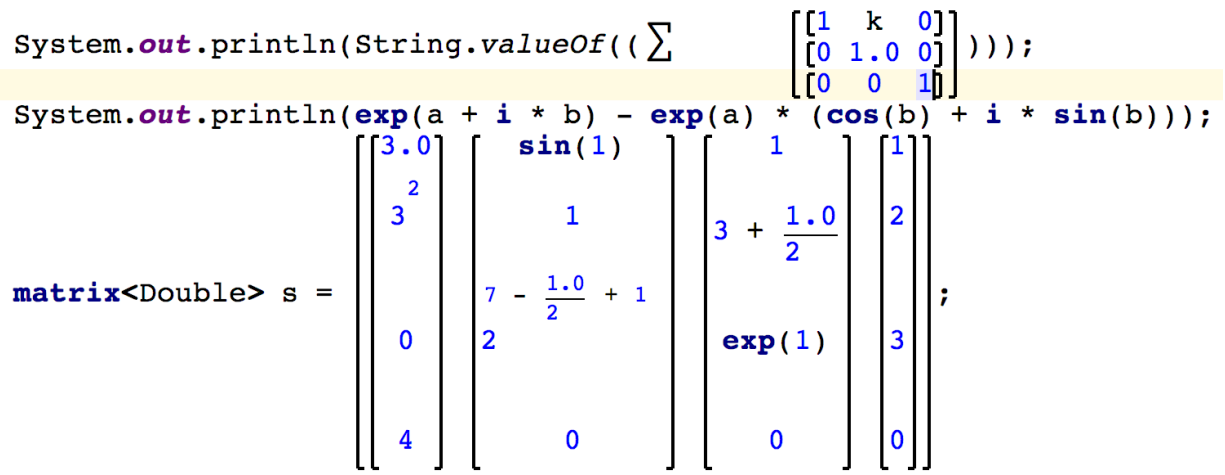


Рисунок 4. Пример кода в проекционном редакторе

С точки зрения системы MPS любой язык состоит из строительных блоков, «концептов». Так, во многих промышленных языках существуют

концепты условного оператора и арифметических выражений. Подобно тому как в ООП организована работа с классами, концепты могут наследовать (абстрактные) концепты и реализовывать интерфейсы.

Для описания характеристик концептов используются «аспекты», для каждого из которых существует собственный язык, заточенный под проблемы, которые возникают в аспекте:

- **структурный аспект** описывает концепты в DSL и по своей сути формирует типы узлов в АСД: их поля и отношения с другими узлами дерева. Например, в аспекте концепта «условный оператор» может присутствовать поле с условием и ссылки на узлы then-else.
- **ограничительный аспект** устанавливает ограничения на поля узлов, взаимное расположение узлов в дереве, методы доступа к полям.
- **аспект-редактор** определяет способ представления, с которым работает конечный пользователь, например, исходный код, должно конвертироваться в АСД концепта.
- **аспект-генератор** содержит правила для трансформации концепта исходного определяемого языка в низкоуровневые абстракции и/или код целевого языка (например, Java, байткод).
- прочие аспекты, выполняющие задачи проверки типов, детектирования недостижимого кода, тестирования, миграций между версиями языка, отладки, автодополнения в IDE.

Для создания языка или расширения уже существующего языка программист последовательно определяет весь набор концептов языка, которые описывают понятия, используемые в нём. Далее собирает проект MPS и получает плагин с поддержкой языка для платформы Intellij.

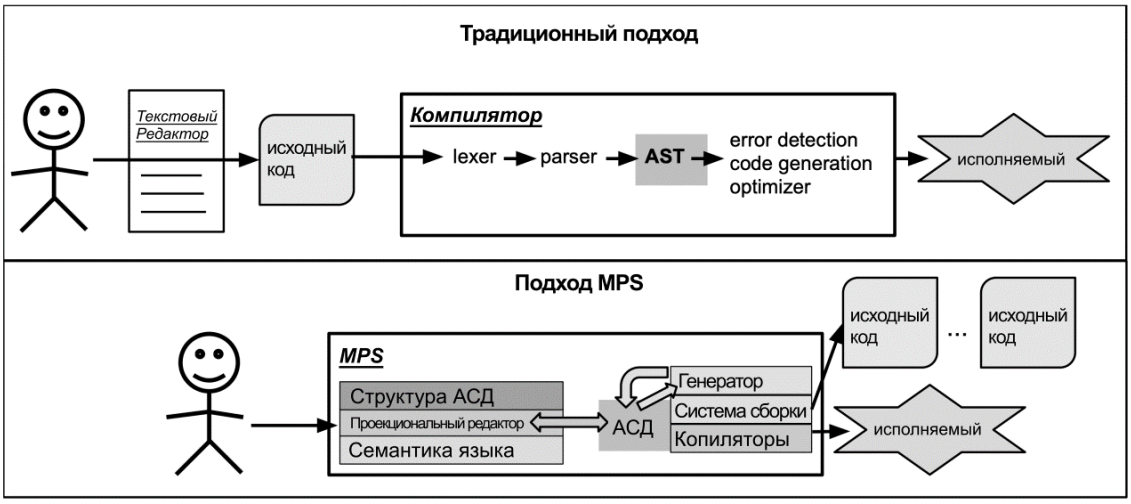


Рисунок 5. Традиционный и MPS подход к способу задания DSL

По сравнению с традиционным подходом, когда программа представляется в виде текста и далее с помощью лексического и синтаксического анализаторов трансформируется в АСД, подход MPS, работая с АСД напрямую, обладает следующими преимуществами:

1. Возможность комбинировать различные языки в одной программе без риска неоднозначности парсинга конкретного синтаксиса.
2. Представление элементов языка не только в виде текстовых примитивов, но и формул, таблиц, диаграмм, графиков, что упрощает восприятие и позволяет сосредоточиться на предметной области.
3. Интегрированная со средой разработки поддержка автодополнения, навигация, подсветка синтаксиса и ошибок, инспекции кода, инструменты рефакторинга. Для реализации не требуется знать теорию синтаксических анализаторов.
4. Упрощённая разработка расширений уже существующих языков. Расширение компонентов компилятора для сложных языков требует как наличие опыта и особых умений, из-за риска случайно ввести неоднозначности в грамматику языка, так и знание внутреннего устройства конкретного компилятора. Расширение же

языка с помощью MPS состоит из определения новых структур языка, проекционных редакторов для этих структур и семантики.

Из минусов можно выделить:

1. Поддерживается только платформа Intellij.
2. Крутая кривая обучения [30]: требуется усвоить большое количество абстракций, прежде получится написать язык.
3. Стандартный проекционный редактор, который автоматически генерируется для каждой структуры (концепта) языка при отсутствии аспекта редактора неудобен в использовании. [29]
4. Структурные правила описывают как выглядит абстрактное синтаксическое дерево, на какие узлы оно разбито. Но отсутствует информация о конкретных характеристиках написанного кода: о пробелах, разрывах строки и прочем форматировании — которая может быть важна для реализуемого DSL.

Rational Rose

IBM Rational Rose - популярное средство визуального моделирования, которое считается полулярным среди средств визуального проектирования приложений [20]. Этот продукт входит в состав пакета IBM Rational Suite и предназначен для моделирования программных систем с использованием широкого круга инструментальных средств и платформ.

Являясь простым и мощным решением для визуальной разработки информационных систем любого класса, Rational Rose позволяет создавать, изменять и проверять корректность модели. Rational Rose объединяет команду разработчиков на базе универсального языка моделирования UML, который определяет стандартную графическую символику для описания архитектуры ПО. Любые участники проекта - аналитики, специалисты по моделированию, разработчики и другие - могут использовать модели, построенные в Rational Rose, для большей

эффективности создания конечного продукта.

Rational Rose в отличие от подобных средств проектирования способна проектировать системы любой сложности, то есть инструментарий программы допускает как высокоуровневое (абстрактное) представление (например, схема автоматизации предприятия), так и низкоуровневое проектирование (интерфейс программы, схема базы данных, частичное описание классов). Вся мощь программы базируется всего на 7 диаграммах, которые в зависимости от ситуации способны описывать различные действия[21].

Однако стоит отметить, что Rose позволит упростить разработку сложных классов посредством выразительных возможностей по графическому представлению классов и их взаимоотношений. Иначе говоря, этот инструмент сможет сгенерировать только код классов и их связей на определенном языке программирования. Исполняемый код генерироваться не будет.

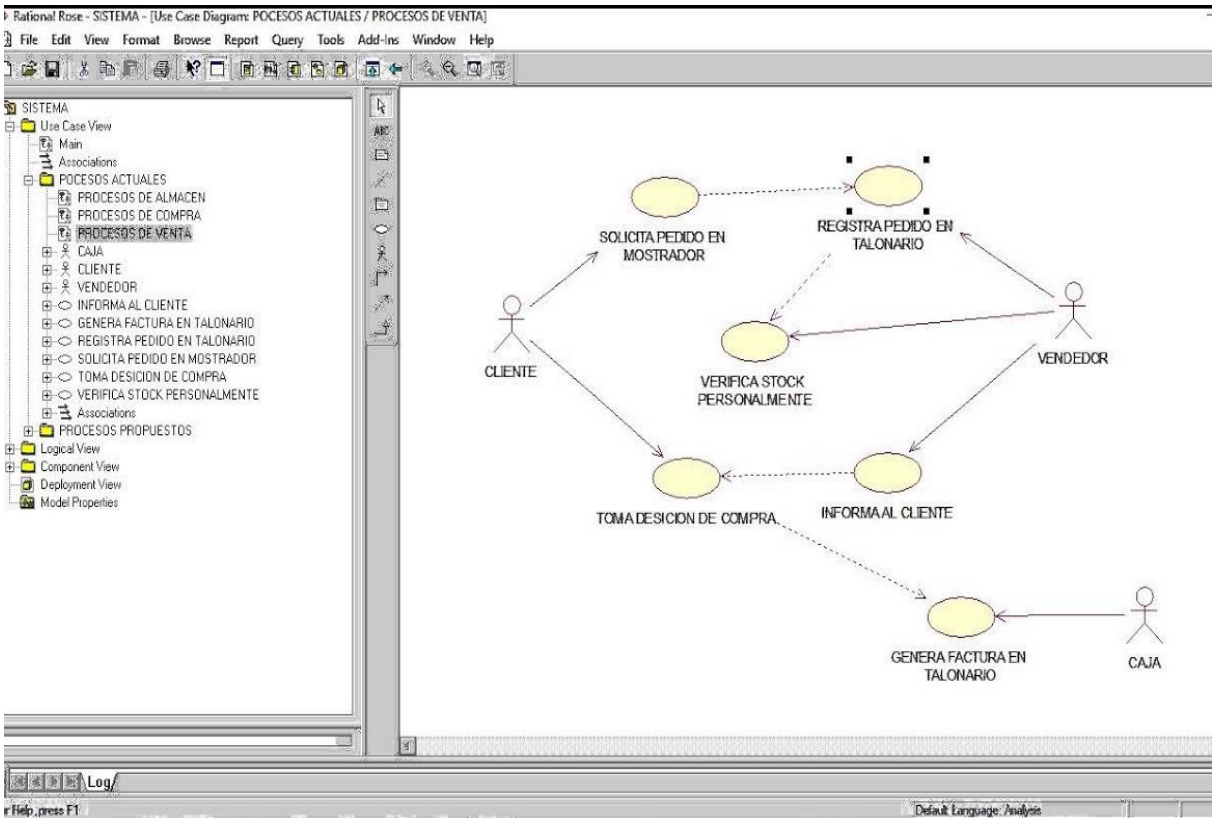


Рисунок 6. Пользовательский интерфейс Rational Rose

Windows Workflow Foundation

Windows Workflow Foundation создан для описания движения и взаимодействия объектов внутри программ, позволяет производить визуальное проектирование моделей и генерацию кода по построенным моделям.

При помощи WF могут быть описаны три типа процессов[21]:

- последовательный процесс (Sequential Workflow) — переход от одного шага в другой без возвратов обратно;
- конечный автомат (State-Machine Workflow) — переход из одного состояния в другое, возможны и произвольные возвраты в предыдущие состояния;
- процесс, управляемый правилами (Rules-driven Workflow) — частный случай последовательного процесса, в котором переход на следующий шаг определяется набором правил.

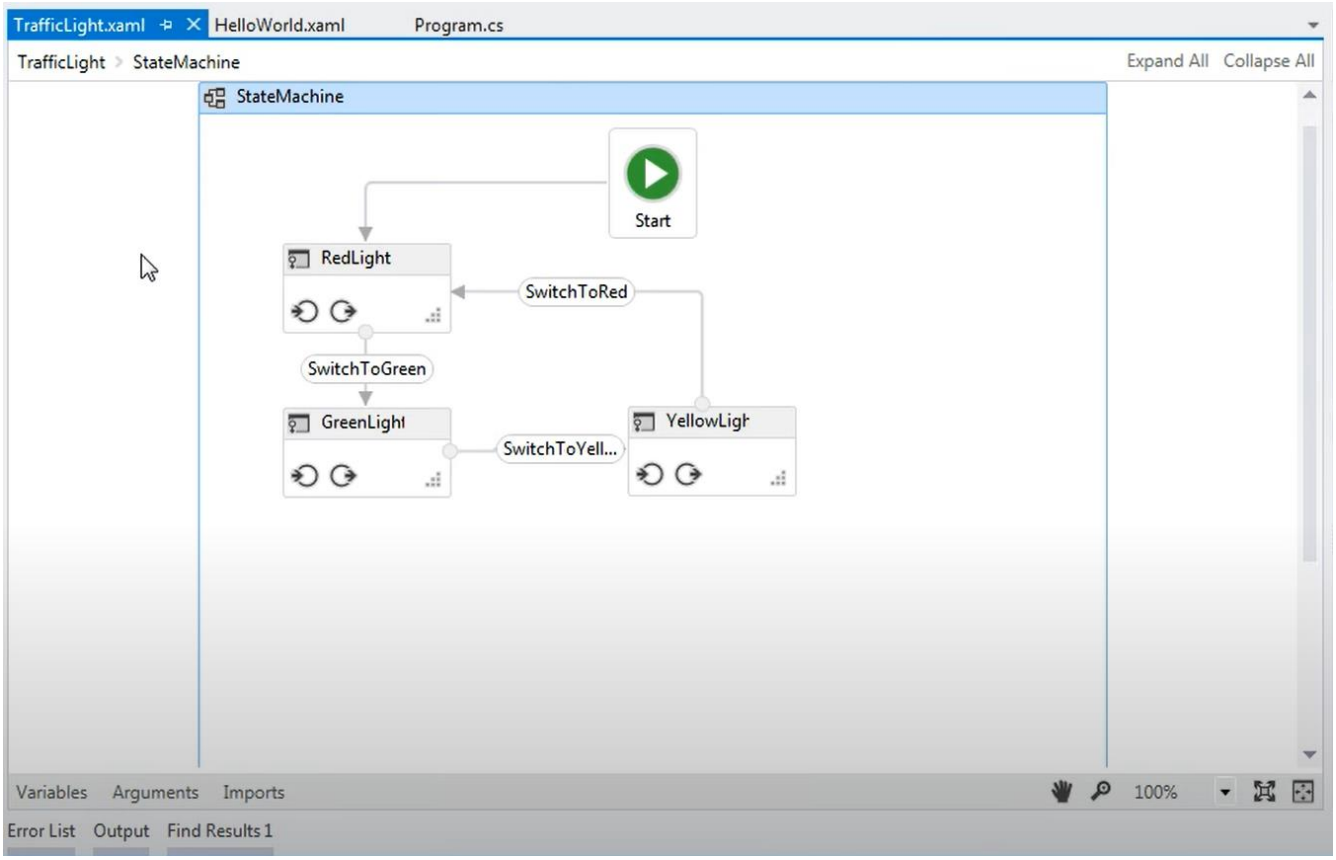


Рисунок 7. Пользовательский интерфейс Windows Workflow Foundation

State-Machine Workflow предоставляет пользователю возможность

визуального программирования на основе диаграммы состояний. Также возможны переходы при наступлении некоторого события. После того, как диаграмма переходов между состояниями готова, полученный автомат можно сгенерировать в код в C#.

Выводы к главе

1) Показано, что для создания языка программирования требуется задать его синтаксис и семантику. Рассмотрены основные походы к описанию синтаксиса и семантики.
2) Выполнен обзор существующих технологий и инструментальных средств задания языков предметных областей.
3) Установлено, что существующие технологии обладают рядом недостатков и ограничений, затрудняющих их использование при простом и быстром процессе обучения создания и освоения предметно-ориентированных языков.

Реализация инструментального средства editor.cf

Глава посвящена изложению подхода к описанию синтаксиса DSL с помощью формальной грамматики, семантики DSL — с помощью модели взаимодействия автоматных объектов. Также рассмотрена архитектура и дизайн программного решения инструментальной поддержки DSL.

Описание синтаксиса

Описать синтаксис входного языка пользователю предлагается с помощью инструмента ANTLR4 [15]. ANTLR4 — генератор парсеров, позволяющий автоматически создавать программы лексического и синтаксического анализа на одном из целевых языков программирования (C++, Java, C#, Python). Помимо того, что он поддерживает РБНФ входных грамматик, инструмент обладает всеми преимуществами LL-техники генерации парсеров: простое отслеживание ошибок вывода, простое осуществление “нагрузки” анализа различными семантическими конструкциями и функциями [16].

ANTLR4 представляет собой набор возможностей, состоящий из двух частей.

- генератор анализаторов — приложение, которое получает на вход описание грамматики и генерирует код для лексического и синтаксического анализатора.
- библиотека времени выполнения, которая используется для создания конечной программы. Эта библиотека содержит базовые классы для анализаторов, а также классы, управляющие потоками символов, обрабатывающие ошибки разбора, генерирующие выходной код на основе шаблонов и многое другое.

В данной работе [**Error! Reference source not found.**] автор

утверждает, что ANTLR4 достаточно прост для понимания новичка в области конструирования языков программирования, так как этот инструмент тщательно документирован, для него существует много успешных примеров реализации DSL, а также доступны инструменты разработчика, такие как интегрированные плагины среды разработки и собственная среда разработки AnlrWorks.

Так как представленный в работе инструмент разрабатывается для студентов, то аргументы выше стали основополагающими в выборе средства для описания синтаксиса. Кроме того DSL, создающиеся студентами в рамках курсовой работы, в основном простые, то есть их синтаксис в большинстве случаев регулярный и не имеет неоднозначностей в своем строении.

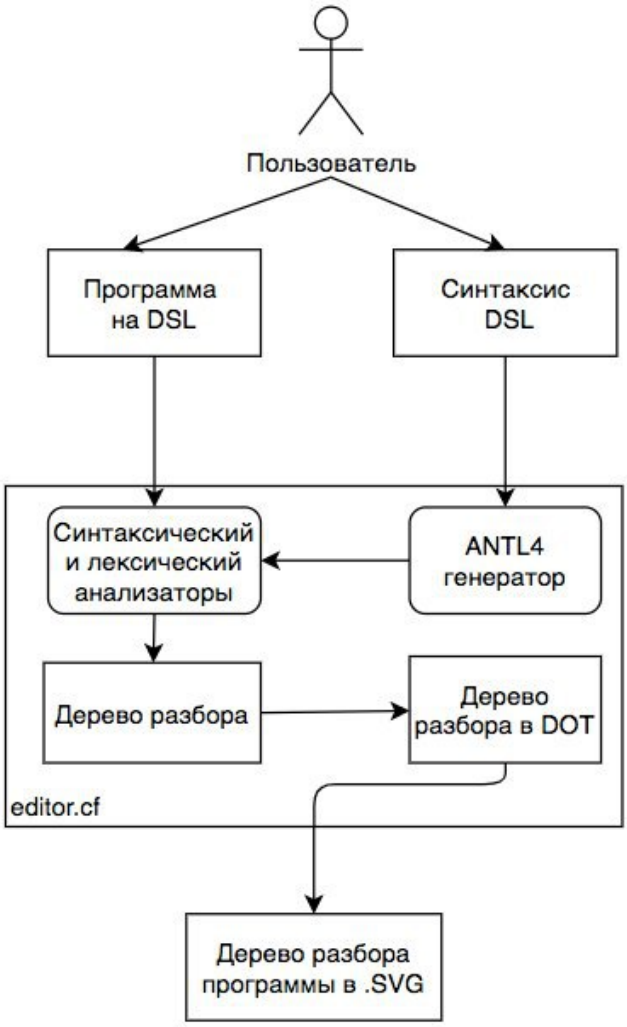


Рисунок 8. Процесс генерации дерева разбора

Создатель языка описывает синтаксис в расширенной форме Бэкуса-Наура с помощью инструмента ANTRL4. Далее принимая на вход текст входной программы, разработанный инструмент (editor.cf – подробнее об инструменте указано в главе Архитектура и дизайн решения) может отобразить пользователю дерево разбора программы. Дерево разбора программы является удобным и наглядным способом отладки синтаксиса программы.

Процесс создания дерева разбора начинается с генерации лексического и синтаксического анализаторов. После этого система строит дерево разбора программы. Дерево разбора конвертируется из внутреннего представления ANTLR4 в нотацию языка описания графов и диаграмм DOT [23].

Далее с помощью пакета утилит по автоматической визуализации графов Graphviz (Graph Visualization Software [24]) формируется векторное изображение в SVG формате. Данный файл может быть просмотрен и скачан пользователем. SVG-формат был выбран из-за таких очевидных преимуществ, как:

- возможность редактирования. То есть создатель языка при желании, может изменить, например, цвет или расположение узлов дерева в полученном файле.
- возможность масштабируемости. Так как в общем случае деревья разбора получаются громоздкими для больших программ, важным фактором будет выступать увеличение любой части SVG-изображения без потери качества.

Если при формировании дерева разбора возникли исключительные ситуации, то они будут транслированы пользователю с указанием этапа обработки программы, на котором возникла ошибка. Также по возможности будет указан номер проблемной строки или символа, если ошибка возникла при определении грамматики или при анализе кода

программы. Отображение ошибок и неточностей при разработке DSL в обучающих инструментах создает комфортные условия для быстрого и эффективного обучения.

Описание семантики

Описать семантику пользователю предлагается автоматным подходом.

Мотивация

Автоматное программирование — парадигма программирования, в рамках которой программные системы представляются в виде конечных автоматов [**Error! Reference source not found.**]. Суть автоматного программирования состоит в выявлении набора состояний описываемого процесса, условий перехода между этими состояниями и выявлении действий, которые должны выполняться в том или ином состоянии. Иначе говоря, автоматный объект реагирует на входное воздействие не только сменой своего состояния, но и формированием определенных значений или действий на выходе.

Развитием парадигмы автоматного программирования стала модель автоматного объекта [10]. Концепция автоматного объекта расширяет возможности обычного автомата и позволяет ему общаться с внешним миром через интерфейсы взаимодействия. Автоматный объект обеспечивает входное и выходное взаимодействие. При этом в соответствии с принципом Б. Мейера [**Error! Reference source not found.**], операции интерфейса любого объекта разделяются на запросы, доставляющие значения и не меняющие состояния объекта, и команды, меняющие состояние объекта, но не доставляющие значений. Таким образом, на основе этих двух факторов автоматный объект может взаимодействовать с окружающим миром четырьмя различными способами:

- событие (входное взаимодействие меняющее состояние автомата)
- сторожевые условия (входное взаимодействие доставляющее значение в автомат)
- эффекты (выходное взаимодействие меняющее состояние внешнего мира)
- текущее состояние (выходное взаимодействие доставляющее значение во внешний мир)

Предоставляя такие возможности автоматным объектам их можно связывать их друг с другом, тем самым получая модель взаимодействующих автоматных объектов.

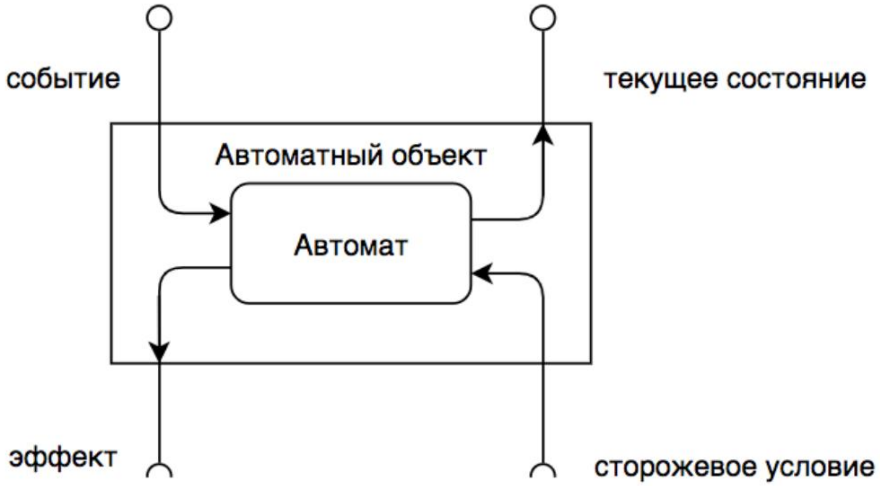


Рисунок 9. Автоматный объект

Описанная выше технология легла в основу создания языка описания взаимодействия автоматных объектов CIAO (Cooperative Interaction of Automata Objects) [13]. Идеи данной работы легли в основу конструирования схожего графического языка, но с некоторыми изменениями:

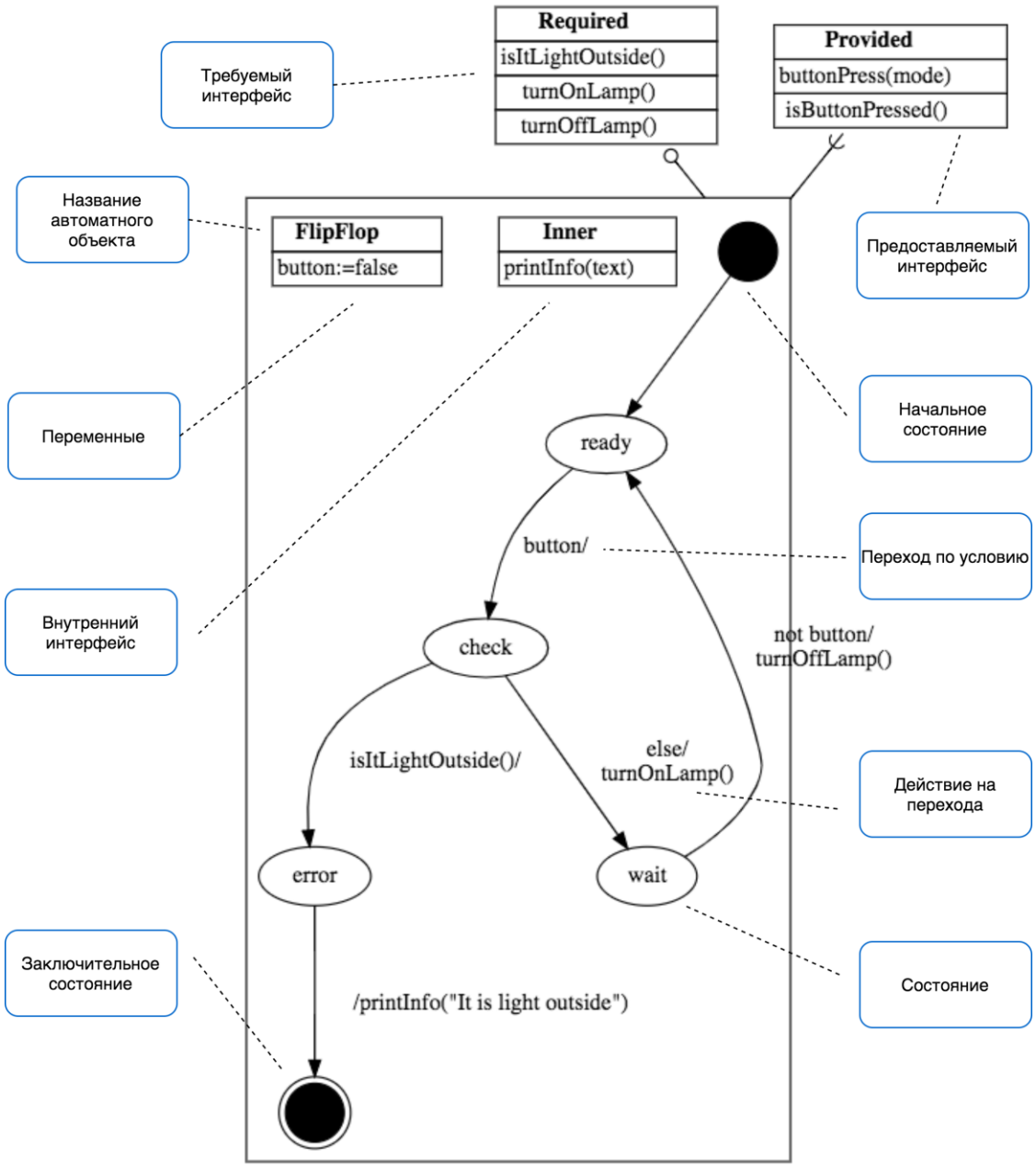
- стирается грань между событием и сторожевым условием, данные операции заменяются на предоставляемый интерфейс. В

графической нотации предоставляемый интерфейс выглядит как «шарик» или «леденец»

- стирается грань между эффектом и текущим состоянием, данные операции заменяются на требуемый интерфейс. В графической нотации требуемый интерфейс выглядит как «сокет»
- появляется раздел внутренних операций, которые используются только внутри автомата и не имеют завязки на внешний мир

Созданный язык в честь предшественника был назван CIAOv2.

Графическое представление языка CIAOv2



Пример автоматного объекта в виде изображения на языке CIAOv2 можно увидеть на рисунке 10. Каждый автоматный объект содержит:

- Название. Обязательны параметр. Является идентификатором автоматного объекта
- Раздел переменных. Необязательный раздел. Содержит переменные, используемые и изменяемые в течении жизни автоматного объекта
- Раздел требуемого интерфейса. Необязательный раздел. В данном примере isItLightOutside() сторожевое условие, а turnOnLamp() и turnOffLamp() — эффекты.
- Раздел предоставляемого интерфейса. Необязательный раздел. В данном примере isButtonPressed() — текущее состояние, а buttonPress(mode) — событие. Данные методы не представлены на диаграмма переходов, так как предназначены для вызова из внешнего контекста
- Раздел внутреннего интерфейса. Необязательный раздел. В данном примере printInfo(text) внутренняя операция, которая может вызываться несколько раз в течении жизни автоматного объекта

Рисунок 10. Нотация диаграммы переходов на примере счетного триггера

- Начальное состояние. Обязательный параметр. Всегда единственно
- Конечное состояние. Необязательный параметр. Необязательно единственно
- Переходы, ведущие из состояний в состояния. Обязательно имеют начальное и конечное состояние
- Состояния. Необязательные параметры. Но обязательно присутствие хотя бы одного состояния — начального

- Условие перехода. Необязательный параметр. Условие должно возвращать истину или ложь. Также возможет переход по условию «else», если переход по всем другим условиям не сработал
- Действие при переходе. Необязательный параметр. Операция, которая выполняется после проверки условия в момент перехода

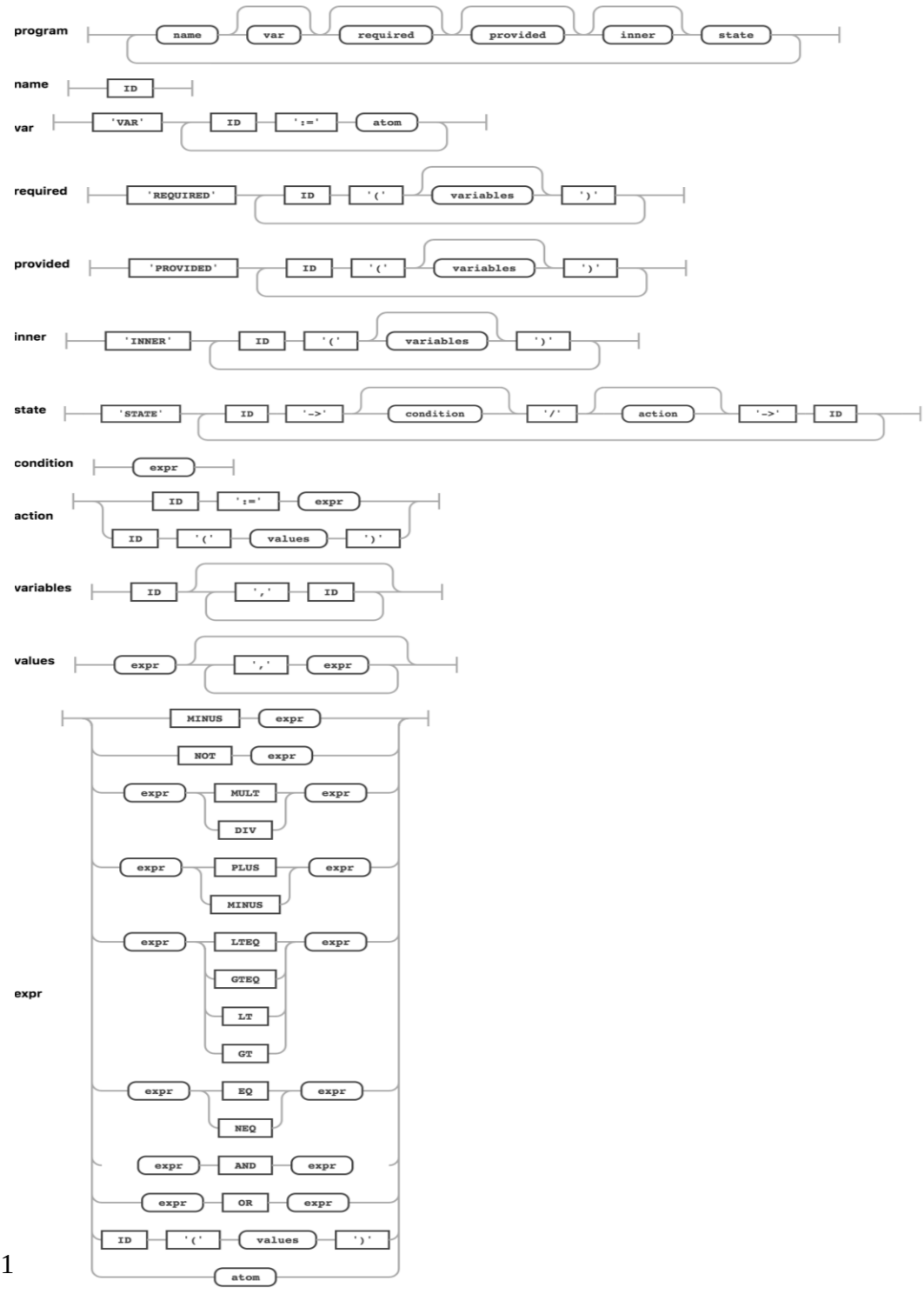


Рисунок 11)

в следующее состояние. Может возвращать значение, однако оно будет игнорироваться.

Формальная грамматика CIAOv2

CIAOv2 — текстовый язык описания автоматных объектов. На рисунке 11 приводится описание нетерминалов грамматики CIAOv2 в виде синтаксических диаграмм.

Рабочее описание грамматики со всеми терминалами и нетерминалами и заданное с помощью инструмента ANTLR4 приведено в приложении 1 к работе.

Пример текста программы на языке CIAOv2, диаграмма которой приведена на рисунке 10 выглядит следующим образом:

```
FlipFlop
VAR
button := false
REQUIRED
isItLightOutside()
turnOnLamp()
turnOffLamp()
PROVIDED
buttonPress(mode)
isButtonPressed()
INNER
printInfo(text)
STATE
entry -> / -> ready
ready -> button / -> check
check -> isItLightOutside() / -> error
check -> else / turnOnLamp() -> wait
wait -> !button / turnOffLamp() -> ready
error -> / printInfo("It is light outside") -> exit
```

Листинг 1. Программа на CIAOv2 для реализации счетного триггера

В первой строке находится имя (идентификатор) автоматного объекта. Далее идут разделы определения переменных, требуемого, предоставляемого и внутреннего интересов. В секции STATE находится перечисление состояний и переходов в следующей форме:

“текущее состояние -> условие / действие -> следующее состояние”

Однако, если входная программа успешно прошла этап синтаксического анализа, это не говорит о ее корректности, так как существует ряд ограничений (например, единственность входного состояния), которые не заложены в описание грамматики. Таким образом, существует следующий ряд контекстных условий программы на языке CIAOv2, которые запускаются после прохода синтаксического анализатора:

- наличие входного состояния
- единственность входного состояния
- отсутствие одинаковых имен переменных и интерфейсов
- количество переменных в описании интерфейса совпадает с количеством переменных в вызове
- начальное состояние не может быть концом перехода
- заключительное состояние не может быть началом перехода
- на множестве переходов из состояния A в состояние B есть не более одного условия «else»
- «else» зарезервированное слово (переменные с таким именем запрещены)

Генерация кода по CIAOv2

После того, как исходный текст программы прошел все вышеописанные проверки, он может быть транслирован:

- в диаграмму графа переходов состояний на языке DOT
- в заготовку кода программы на языке Python3

Пример сформированного векторного изображения по коду в DOT представлено на рисунке 10.

Ниже представлен листинг заготовки кода на Python3, построенный на основе примера, обсуждаемом в этой главе.

```
class FlipFlop:

    #VARS
    def __init__(self):
        self.button = false
        self.__state__ = "entry"

    #REQUIRED
    #uses
    # isItLightOutside()
    # turnOnLamp()
    # turnOffLamp()

    #INNER
    def __printInfo(self, text):
        pass

    #PROVIDED
    def buttonPress(self, mode):
        pass

    def isButtonPressed(self):
        pass

    def run(self):
        while (True):
            if self.__state__ == "exit":
                break
            if self.__state__ == "entry" :
                if True:
                    self.__state__ = "ready"
                    continue
            if self.__state__ == "ready" :
                if button:
                    self.__state__ = "check"
                    continue
            if self.__state__ == "check" :
                if isItLightOutside():
                    self.__state__ = "error"
                    continue
                turnOnLamp()
                self.__state__ = "wait"
                continue
            if self.__state__ == "wait" :
                if not button:
```

```
                turnOffLamp()
                self.__state__ = "ready"
                continue
        if self.__state__ == "error" :
            if True:
                self.__printInfo("It is light outside")
                self.__state__ = "exit"
              continue
```

Листинг 2. Шаблон кода на Python3 для реализации счетного триггера

Фрагмент кода представляет собой класс, где есть:

- Конструктор, который инициализирует переменные и присваивает служебной переменной __state__ начальное состояние
- Заготовки для описания приватных методов (раздел INNER) и публичных методов (раздел PROVIDED). Раздела REQUIRED в коде представлен в виде комментария (#uses), так как это требуемый интерфейс от внешней системы
- Основной ход программы в виде бесконечного цикла, условий выбора следующего состояния и действий при переходе.

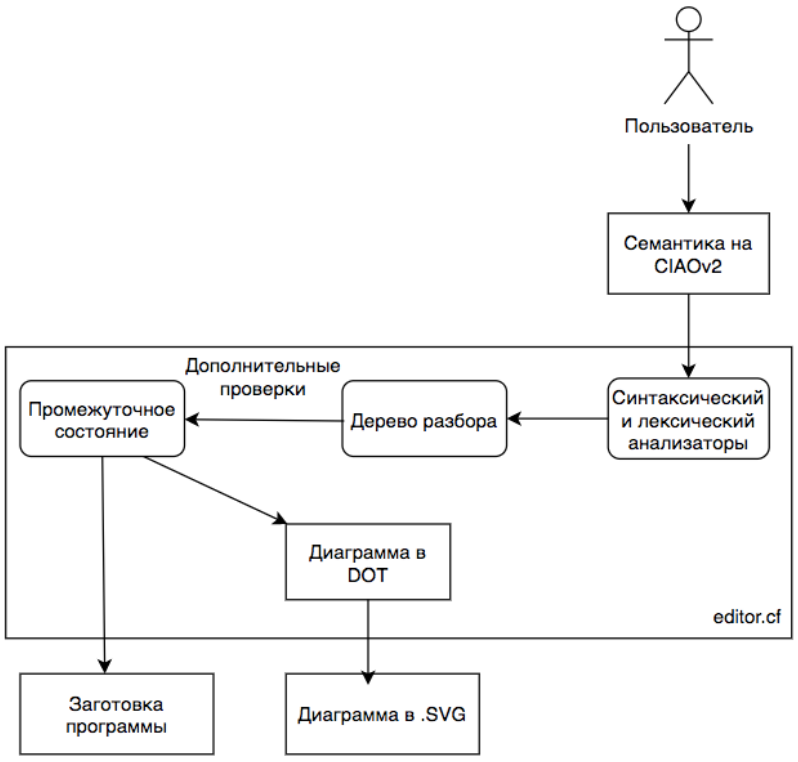


Рисунок 12. Процесс генерации диаграммы и заготовки программы

В общем случае процесс работы программного решения для описания семантики на языке CIAOv2 и трансляции кода с языка CIAOv2 в код программы на Python3 или диаграмму переходов представлен на рисунке 12.

Архитектура и дизайн решения

В рамках решения поставленной задачи был разработан программный модуль, который представляет собой веб-приложение со следующими возможностями для пользователя:

- описания синтаксиса языка на ANTLR4
- описания семантики языка на CIAOv2
- скачивание SVG-файла с деревом разбора программы на основе введенного синтаксиса
- скачивание SVG-файла с семантикой языка в виде диаграммы переходов
- скачивание .py-файла с заготовкой программы на языке Python3 для описания семантики

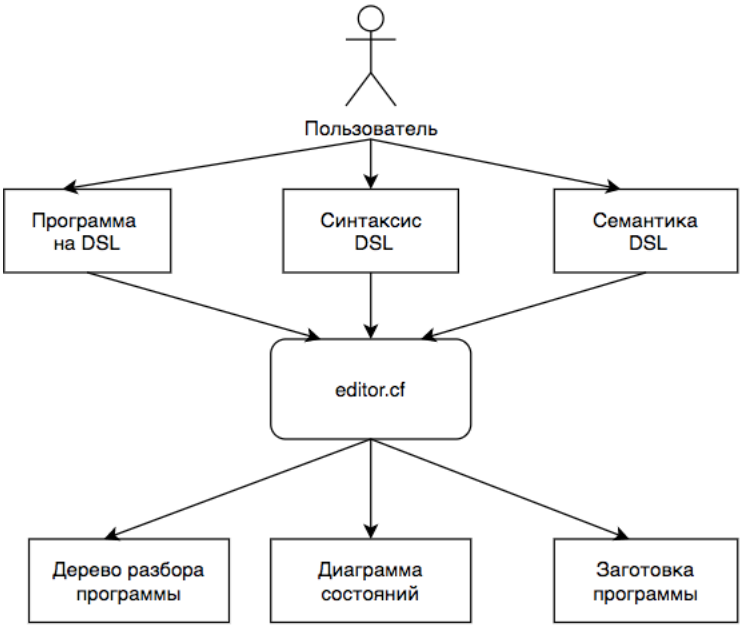


Рисунок 13. Диаграмма потоков данных для сервиса editor.cf

На момент написания работы инструмент доступен по данному адресу: http://editor.cf/

Интерфейс веб-приложение editor.cf представлен на рисунке 14. В левом верхнем окне пользователь указывает исходный текст программы, в левом нижнем окне — синтаксис программы, в правом верхнем — семантику. Также у пользователя есть три кнопки внизу экрана, с помощью которых он может скачать дерево разбора программы, диаграмму состояний и заготовку программы.

Фронтальная часть приложения написана с помощью технологии ReactJS [25]. ReactJS — разработанный компаниями Facebook и Instagram фреймворк с открытым исходным кодом для создания пользовательских интерфейсов. ReactJS позволяет разработчикам создавать крупные веб-приложения, при этом отрисовка элементов будет происходить только в том случае, если они были затронуты изменением данных.

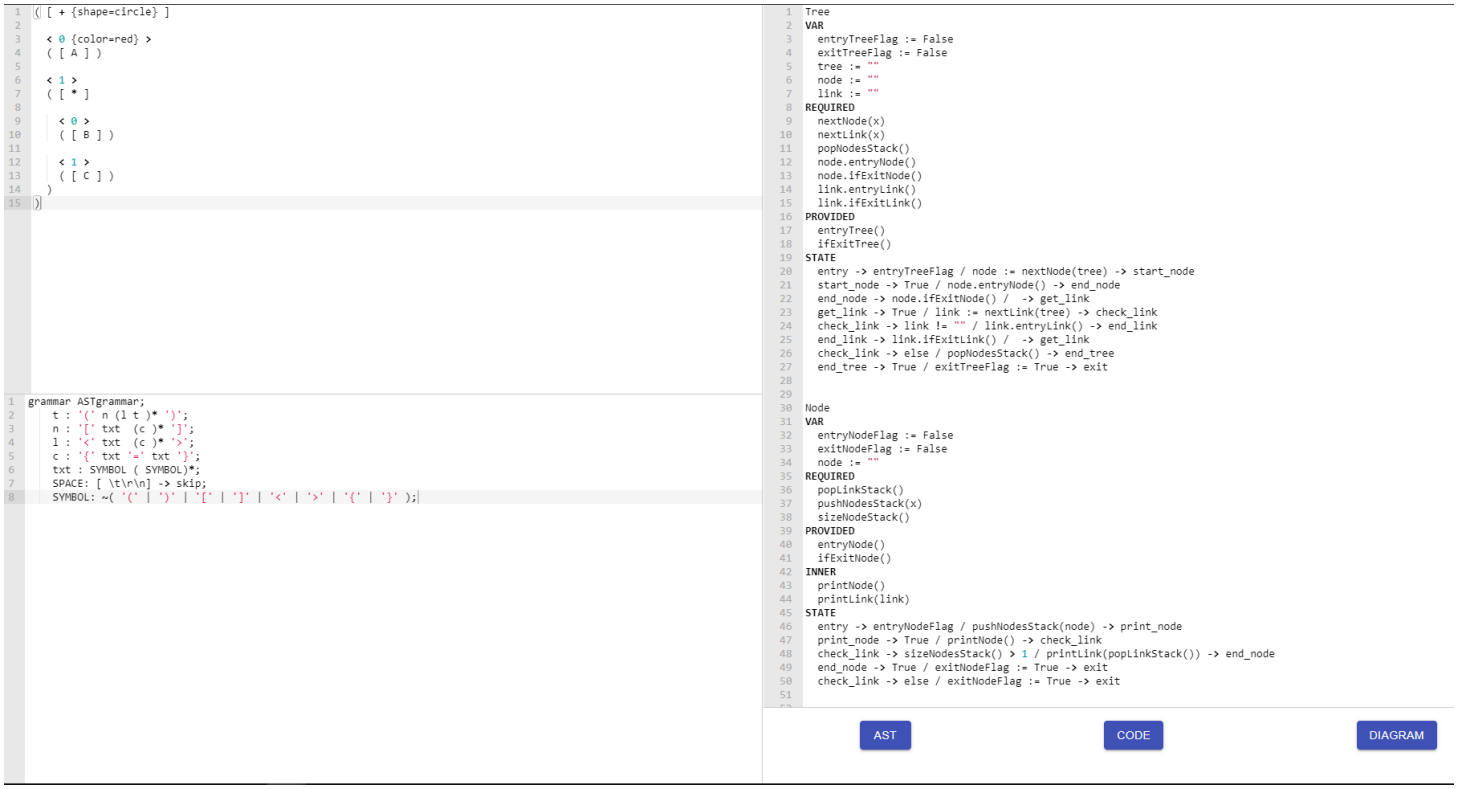


Рисунок 14. Интерфейс пользователя

Основная концепция создания React-приложений — разбивка на компоненты. Компоненты позволяют разделить интерфейс на независимые, повторно используемые части и работать с каждой из них отдельно. Компоненты, которые были созданы во время работы над тем или иным проектом, не имеют дополнительных зависимостей. Таким образом, компонентно-ориентированный подход дает возможность переиспользовать код и превращают React-разработку в непрерывный процесс улучшения.

В качестве компонента для отображения и редактирования кода используется AceEditor [26]. AceEditor — онлайн-редактор исходного кода с подсветкой синтаксиса и темами, написанный на Javascript. Основное преимущество этого редактора, что он может быть встроен в любую веб-страницу и приложение JavaScript.

AceEditor поддерживает изменение подсветки синтаксиса для некоторых предопределённых режимов. В рамках данной работы для языка CIAOv2 был написан собственный режим подсветки синтаксиса и выделения таких ключевых слов как VAR, REQUIRED, PROVIDED, INNER и STATE.

Серверная часть приложения была создана с помощью веб-фреймворка Flask[27]. Flask представляет собой легковесный инструмент для создания веб-приложений. Например, в нем отсутствуют встроенный интерфейс администратора, собственный ORM, система кеширования и другие возможности, которые предоставляют высокоуровневые веб-фреймворка. Таким образом Flask является минималистичным каркасом, дающий самые базовые возможности разработчику. Данный факт и послужил причиной выбора его в качестве веб-сервера, так как в данном проекте от серверной части требуется поддержка минимальной функциональности.

В результате разработки серверной части было создан API – программный интерфейс приложения – со следующими возможностями:

- запрос дерева разбора
    - данные запроса:
        - текст исходной программы
        - текст синтаксиса
    - данные ответа:
        - код ошибки
        - страница для перенаправления или информация об ошибке
- запрос диаграммы состояний
    - данные запроса:
        - текст семантики
    - данные ответа:
        - код ошибки
        - страница для перенаправления или информация об ошибке
- запрос заготовки кода
    - данные запроса:
        - текст семантики
    - данные ответа:
        - код ошибки
        - страница для перенаправления или информация об ошибке

Фронтальная часть программного комплекса в зависимости от действий пользователя отправляет необходимые запросы серверу. На основе полученных данных (поля «error» в ответе сервера) фронтальная часть отображает пользователю или сообщение об ошибке или перенаправляет его на страницу для просмотра и скачивания запрашиваемой информации.

Выводы к главе

1) Предложен новый язык автоматного программирования CIAOv2, который используется для описания семантики языков предметных областей.
2) Рассмотрены основные возможности инструмента editor.cf: описание синтаксиса и семантики языка, отображение дерева разбора программы и семантики в виде диаграмм, преобразование семантики в заготовку программы на языке Python3.
3) Описан общий архитектурный подход, применяемый в реализации инструмента editor.cf. Инструмент представляет собой веб-приложение, где фронтальная часть использует библиотеку ReactJS, а серверная часть – веб-фреймворк Flask.

Применение инструмента editor.cf

Для демонстрации основных возможностей программного модуля предлагается рассмотреть его работу на нескольких примерах.

Пример 1. Задача об “умном муравье”

В статье [**Error! Reference source not found.**] приведено описание задачи об «Умном муравье» («Artificial Ant»). Задача об «Умном муравье» является одним из классических примеров применения автоматного программирования и сравнения автоматных подходов. В данном примере рассмотрим, каким образом можно описать решение такой задачи с помощью инструмента editor.cf.

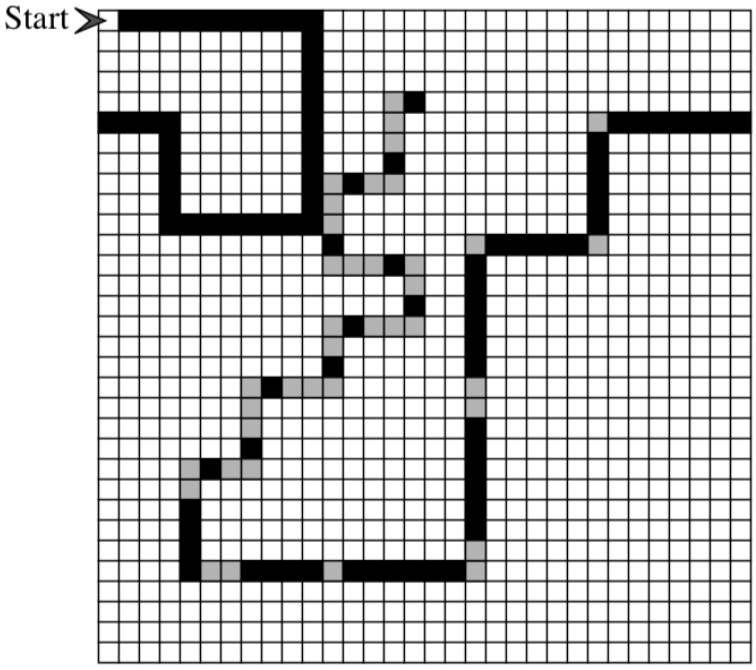


Рисунок 15. Игровое поле

Дано поле — двумерная тороидальная сетка размером 32 на 32 клетки, иначе говоря поле зациклено по границам. В отдельных клетках поля находится еда для муравья. Такие клетки отображаются на поле черным цветом. Всего клеток с едой на поле 89. Черные клетки в задаче

строго заданы, то есть их расположение константно от игры к игре. Остальные клетки пустые. Они отображены белым и серым цветом. Серым цветом отображены клетки пути - ломанной линии - по которой расположена еда для муравья.

Муравей всегда стоит в одной клетке, обращенной в одну из сторон света (север, восток, юг, запад). У муравья есть сенсор, с помощью которого он может определить, если ли на обозреваемой клетке еда или нет. Муравей может обозревать клетку только впереди себя и проверять наличие на ней еды. На каждом шаге муравей должен совершить одно из четырех действий:

а) продвинуться вперед на один шаг;

б) повернуть направо (не двигаясь);

в) повернуть налево (не двигаясь);

г) ничего не делать.

Изначально муравей находится в левом верхнем углу поля и смотрит на восток. Если считать горизонтальную ось поля осью X, а вертикальную — осью Y, то изначально муравей находится в координате (0, 0) и обозревает клетку (1, 0).

Всего муравью дается 200 шагов. За это количество ходов муравей может двигаться по полю, совершая одно из действий выше. Как только муравей входит в клетку на сетке с едой, еда исчезает и муравью присуждается очко. Цель игры — набрать как можно больше очков за 200 шагов.

Ниже на рисунке 16 приведен пример автомата, описывающего движение муравья, который за 200 ходов обходит 81 черную клетку, а 89 черных клеток — за 314 шагов.

На изображении 16 применяются следующие обозначения [**Error! Reference source not found.**]:

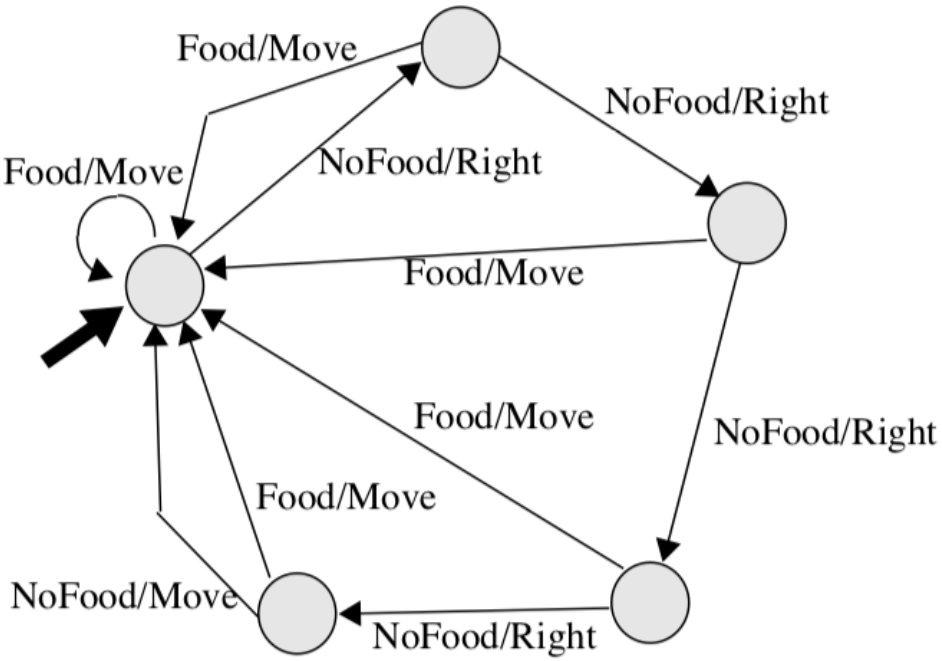


Рисунок 16. Решение задачи об умном муравье

- `MOVE` - продвинуться вперед на один шаг
- `RIGHT` - повернуть направо
- `LEFT` - повернуть налево
- `FOOD` - индикатор наличия еду в обозреваемой муравьем клетке
- `NOFOOD` - индикатор отсутствия еды в обозреваемой муравьем клетке

Выделенная жирным стрелка обозначает начальное состояние муравья.

Если проследить движение муравья по данному алгоритму, то можно увидеть, что он действует по принципу: вижу еду — иду вперед, не вижу еду — поворачиваю.

Однако это не единственное решение задачи об умном муравье с помощью автоматов [**Error! Reference source not found.**]. Данная задача и ее решение используется для иллюстрации работы различных алгоритмов автоматного программирования.

Далее зададим язык, который описывает множество автоматов,

описывающих движение муравья с помощью автоматов. Частными случаями таким программ будут являться программы – решения задачи об «Умном муравье».

Ниже приводится пример того, каким образом разработанный в работе инструмент может отобразить синтаксис и семантику языка описания таких автоматов.

На изображении выше можно выделить следующие элементы:

- узлы -- состояния, в которых находится муравей
- ребра -- переходы между состояниями
- метки к ребру
    - до / - условия перехода из состояния в начале ребра в состояние на конце ребра
    - после / - действия, совершаемые при переходе из состояния в начале ребра в состояние на конце ребра

Таким образом, сохраняя всю информацию об автомате его можно представить в следующем виде:

```
0 -> 0    FOOD      MOVE ;
0 -> 1    NOFOOD    RIGHT ;
1 -> 0    FOOD      MOVE ;
1 -> 2    NOFOOD    RIGHT ;
2 -> 0    FOOD      MOVE ;
2 -> 3    NOFOOD    RIGHT ;
3 -> 0    FOOD      MOVE ;
3 -> 4    NOFOOD    RIGHT ;
4 -> 0    FOOD      MOVE ;
4 -> 1    NOFOOD    MOVE ;
```

Листинг 3. Программа задающая решение задачи об «Умном муравье» на языке описаний движения муравья с помощью автоматов

В такой записи каждое состояние имеет свой номер. Входное состояние номер 0, далее номера растут, обходя вершины по часовой

стрелке. Каждая строка представляет собой запись: «текущее состояние муравья» -> «следующее состояние муравья» «наличие/отсутствие еды в обозреваемой клетке» «действие при переходе в следующее состояние».

Синтаксис языка может быть описан следующей грамматикой с помощью технологии ANTLR4.

```
grammar Game;
rgame        :      (rrule rcolumn )*;
rrule        :      rstate rarrow rstate rcondition raction ;
rstate       :      INT;
rcondition   :      'FOOD' | 'NOFOOD';
raction      :      'MOVE' | 'RIGHT' | 'LEFT';
rcolumn      :      ';';
rarrow       :      '->';
INT          :      [0-9]+ ;
WS           :      [ \t\n\r]+ -> skip;
```

Листинг 4. Грамматика языка описания движения муравья в виде автоматов

После описания грамматики создатель языка может приступить к тестированию корректности задания синтаксиса на примерах. В предлагаемом в работе инструментом можно построить дерево разбора программы.

По грамматике выше и варьируя исходные коды программы создатель языка может получить следующие изображения деревьев разбора:

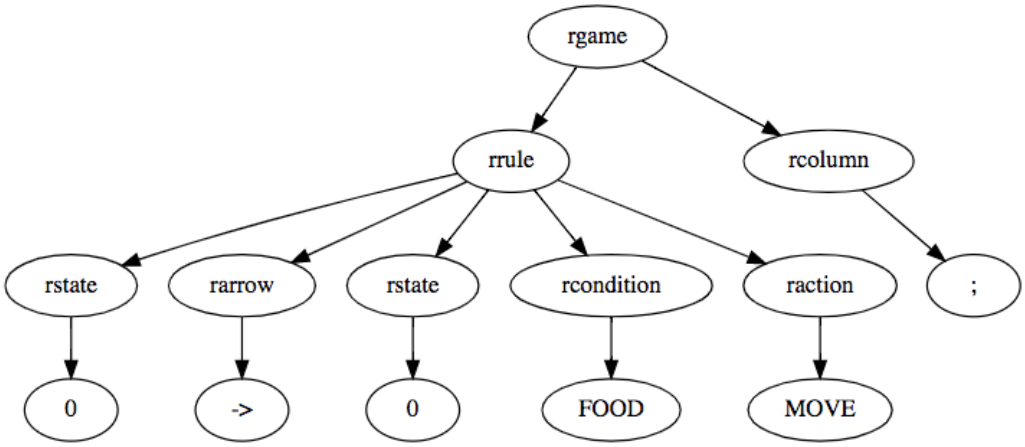


Рисунок 18. Дерево разбора для программы

```
0 -> 0 FOOD MOVE;
```

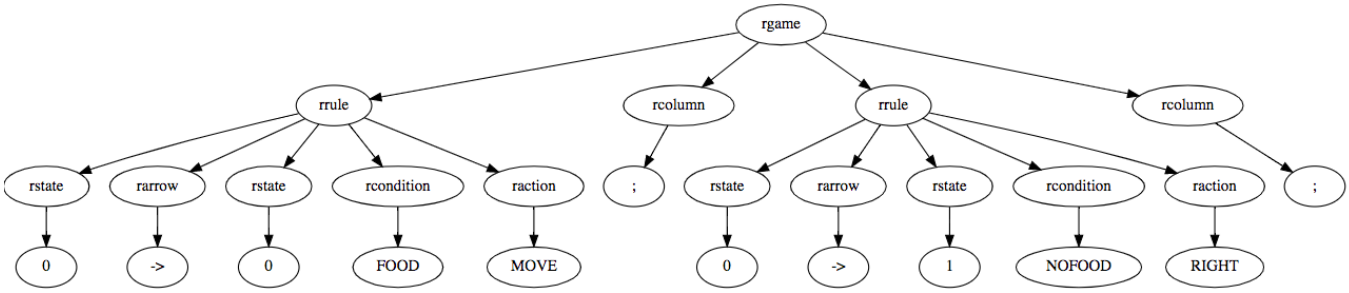


Рисунок 17. Дерево разбора для программы

```
0 -> 0   FOOD     MOVE
0 -> 1   NOFOOD   RIGHT
```

Далее создатель языка может описать семантику с помощью языка CIAOv2.

Семантика описывает процесс выполнения программы с помощью конечного автомата, предоставляемый интерфейс, а также требуемый интерфейс от внешнего окружения. Также в описании семантики могут быть использованы некоторые внутренние (приватные) методы, которые незначительно изменяют внутреннее состояние автомата и используются для сокрытия второстепенной логики программы.

Из иллюстрации задачи можно определить, что муравей находится обычно в четырех состояниях:

- готов к движению
- определил наличие еды в обозреваемой клетке
- выбрал и выполнил действие
- выбрал и перешел в новое состояние

Муравей приходит в завершающее состояние в том случае, если он съел всю еду (то есть количество очков больше или равно 89) или если муравей выполнил более 200 шагов.

Муравью для корректной работы требуется запрашивать от внешнего мира:

- наличие еды на обозреваемой клетке
- действие, которое он может совершить из текущего состояния и наличия еды на обозреваемой клетке
- состояние, в которое он может перейти, из текущего состояния и наличия еды на обозреваемой клетке

Муравей должен уметь выдавать информацию:

- о своей текущей позиции
- о количестве заработанных очков

Также муравей может иметь некоторую второстепенную логику, изменяющую его локальное состояние. Данную логику можно вынести, как и в основные переходы, так и можно скрыть во внутренних (приватных) методах. Например, в качестве второстепенной логики можно будет выделить следующие этапы:

- Шаг муравья вперед. Подразумевается, что в данном случае потребуется:
    - в зависимости от направления муравья сделать шаг вперед
    - увеличить количество очков, если муравей съел еду. Проверить это будет несложно, так как в предыдущем шаге мы уже знаем, видел ли муравей еду на обозреваемой клетке
    - увеличить количество шагов

- Поворот направо
    - изменить обозреваемую муравьем клетку
    - увеличить количество шагов
- Поворот налево
    - изменить обозреваемую муравьем клетку
    - увеличить количество шагов
- Ничего не делать
    - увеличить количество шагов

Также у муравья есть локальные переменные, которые хранят его текущее состояние:

- координаты муравья
- направление движения муравья
- состояние муравья
- наличие еды перед муравьем
- действие муравья
- количество очков
- количество шагов

Итоговая программа на языке CIAOv2 будет выглядеть следующим образом:

```
Game
VAR
  x := 0
  y := 0
  lookAt := "E"
  curState := 0
  food := ""
  action := ""
  score := 0
  steps := 0
REQUIRED
  checkFood(x, y)
```

```
  getState(x, y)
  getAction(x, y)
PROVIDED
  getX()
  getY()
  getScore()
INNER
  move()
  turnRight()
  turnLeft()
  noop()
STATE
  entry ->  / -> ready
  ready -> score >= 89 || steps > 200 / -> exit
  ready -> lookAt == "N" / food := checkFood(x, y - 1) -> scan
  ready -> lookAt == "E" / food := checkFood(x + 1, y) -> scan
  ready -> lookAt == "S" / food := checkFood(x, y + 1) -> scan
  ready -> lookAt == "W" / food := checkFood(x - 1, y) -> scan
  scan -> / action := getAction(curState, food) -> start
  start -> action == "MOVE" / move() -> move
  start -> action == "RIGHT" / turnRight() -> move
  start -> action == "LEFT" / turnLeft() -> move
  start -> action == "NOOP" / noop()  -> move
  move -> / curState := getState(curState, food) -> ready
```

Листинг 5. Решение задачи об «Умном муравье» на языке CIAOv2

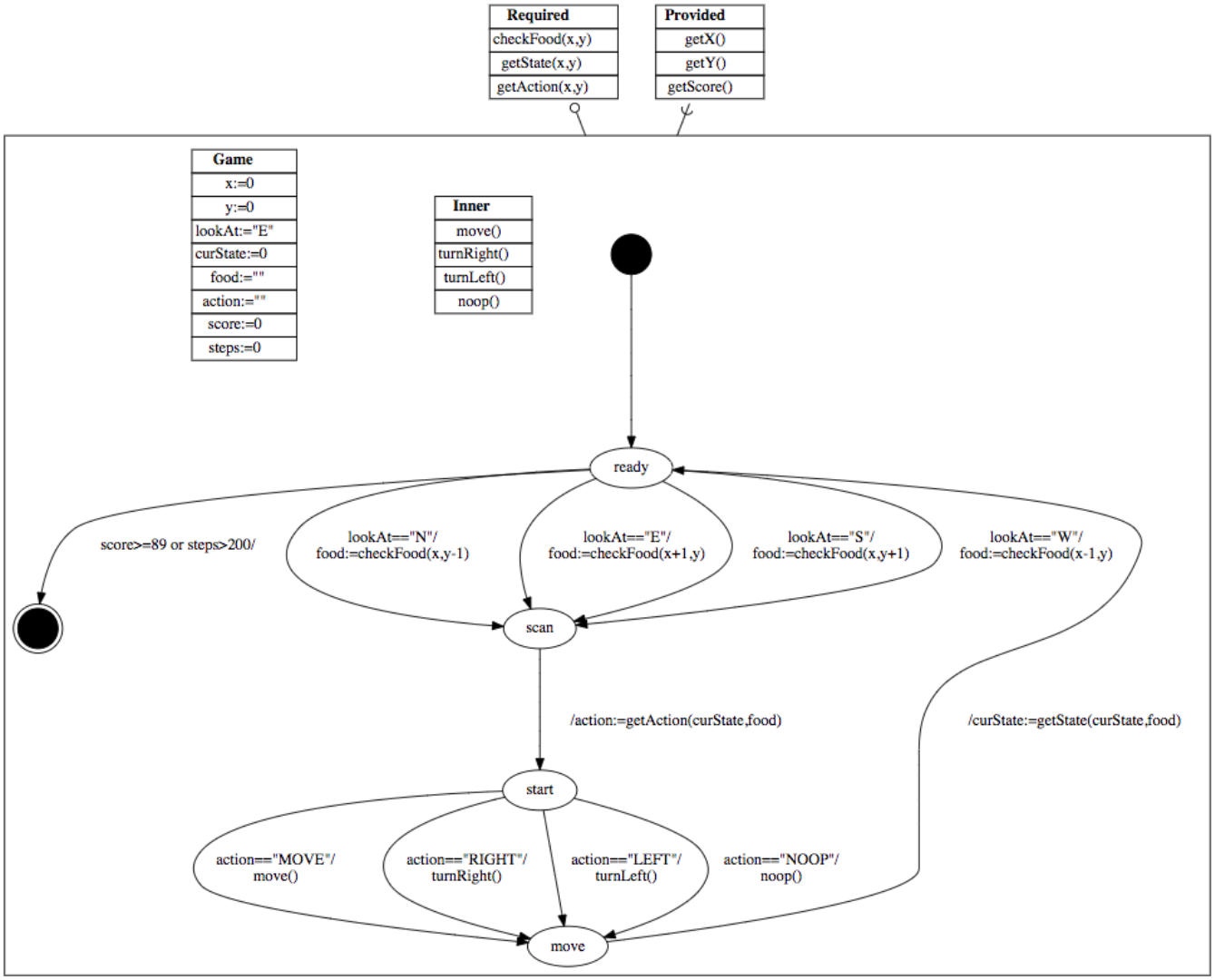


Рисунок 19. Решение задачи об «Умном муравье» на языке CIAO2v2 в графической нотации

Пример 2. Генерация деревьев на языке DOT

Инструмент editor.cf использовался в рамках курса «Грамматики и автоматы», где под руководством Фёдора Александровича Новикова в Санкт-Петербургском политехническом университете Петра Великого (СПбПУ) студентам предлагается получить расширенные знания в теории конечных автоматов и грамматик, а также применить полученные навыки в создании своего языка программирования в предметно-ориентированном подходе.

Язык, который решила описать группа студентов, звучал следующим образом: по описанию дерева на языке, порождаемом заданной грамматикой, сгенерировать код на языке представления графов DOT.

Входной язык описания деревьев порождается следующей грамматикой:

```
grammar ASTgrammar;
t : '(' n (l  t )* ')';
n : '[' txt  (c )* ']';
l : '<' txt  (c )* '>';
c : '{'  txt '=' txt  '}';
txt : SYMBOL ( SYMBOL)*;
SPACE: [ \t\r\n] -> skip;
SYMBOL: ~( '(' | ')' | '[' | ']' | '<' | '>' | '{' | '}' );
```

Листинг 6. Грамматика языка описания деревьев

В такой грамматике дерево представляет собой структуру, заключенную в круглые скобки (), состоящую из узла в квадратных скобках [], и возможно поддеревьев, соединённых с корнем связями в угловых скобках <>. У каждого узла или связи должно быть непустое название. За названием может один или несколько комментариев, заключенных в фигурные скобки «{}».

Комментарии используются для того, чтобы указывать атрибуты отрисовки такие, как цвет связи или форма узла дерева. Атрибуты имеют синтаксис name=value, где name и value произвольные строки, не содержащие =. Любые комментарии, удовлетворяющие данному синтаксису, будут интерпретированы как атрибуты.

В качестве имён узлов и связей могут использоваться любые наборы символов, за исключением “()[]<>{}”.

После того, как грамматика создана, пользователь может попробовать “запустить” ее на примерах реальных программ. Для этого в

инструменте он может ввести синтаксис и пример входного языка и получить дерево разбора. Если программа не будет удовлетворять грамматике, то пользователь получит сообщение об ошибки. Например, для программы вида:

```
(
  [+{shape}]
    <0{color=blue}> ([A])
    <1{color=red}> ([A])
)
```

Листинг 7. Пример некорректной программы на языке описания деревьев

Пользователь увидит на экране:

```
Exception: line 1:9:mismatched input '}' expecting '='
```

Сообщение об ошибке будет указывать пользователю на номер строки и столбца, где возникла ошибка. В данном случае проблема возникла из-за того, что запись “{shape}” не удовлетворяет правилу грамматики “c : '{' txt '=' txt '}';”.

Примеры корректных программ:

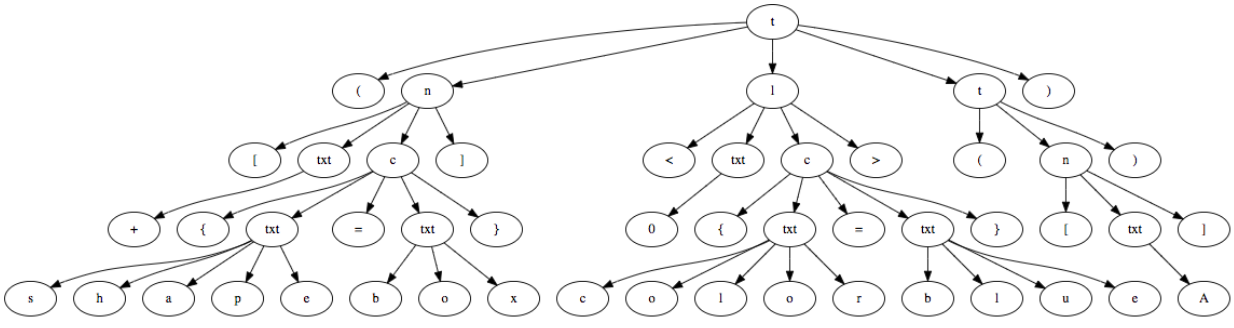


Рисунок 20. Дерево разбора программы

```
([+{shape=box}]<0{color=blue}>([A]))
```

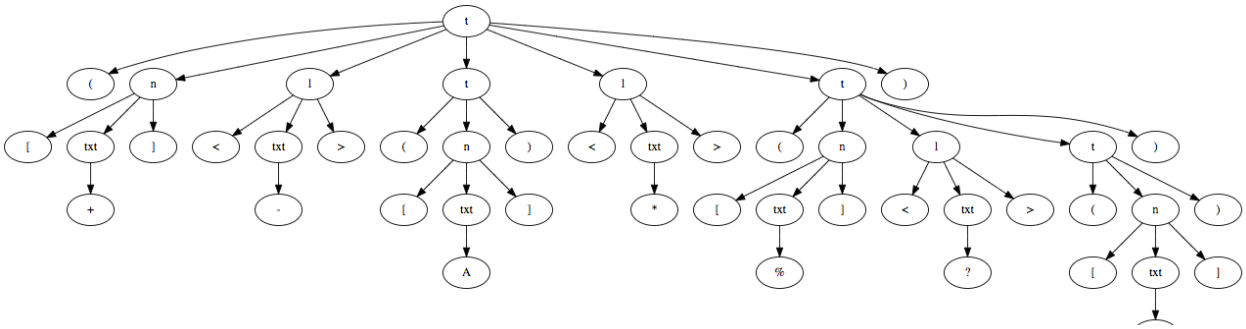


Рисунок 21. Дерево разбора программы

```
([+]<->([A])<*>([%]<?>([%])))
```

Далее переходим к описанию семантики в автоматном стиле. В данном примере семантика должна реализовать обход по дереву разбора. Обход по дереву в автоматном стиле можно задать разными способами.

Первый вариант задания семантики будет монолитным. Иначе говоря, будет создан один автоматный объект, который будет описывать поведение всей программы. Данный автоматный объект будет самостоятельно регулировать обход по дереву, т.е. он знает, на какой узел смотрим сейчас, на какой узел надо смотреть следующим, нужно ли заканчивать осмотр текущего узла и тд. Такому автоматному объекту потребуется «умный» контекст, который потребуется задать пользователю.

Состояние автоматного объекта будут иметь следующий смысл:

- get_tree — получение дерева
- start_tree — старт обхода дерева
- start_node — обход узла дерева
- print_node — вывод информации об узле
- print_link — вывод информации о дуге
- get_link — получение дуги дерева
- check_link — проверка наличия дуги
- end_tree — окончания обхода дерева

Предназначение внутреннего интерфейса автомата:

- printNode(node) — выводит информацию об узле в формате DOT
- printLink(link) — выводит информацию о дуге в формате DOT

Предназначение требуемого интерфейса:

- popNodesStack() — удаление и возврат элемента со стека узлов
- pushNodesStack(x) — добавление элемента на стек узлов
- pushLinkStack(x) — добавление элемента на стек дуг
- popLinkStack() — удаление и возврат элемента со стека дуг
- nextTree(link) — возвращает дерево под дугой link (для пустой дуги - головное дерево)
- prevTree(tree) — возвращает предыдущее дерево от tree (можно было реализовать с помощью стека)
- nextNode(tree) — возвращает узел для дерева tree
- nextLink(tree) — возвращает следующую дугу для дерева t

Можно заметить, что в такой реализации (рисунок 22) у автомата нет предоставляемого интерфейса. В данном случае сгенерированный код программы сразу печатается на экран. Однако, если пользователь решит сохранять код генерируемого языка DOT внутри автомата (например, для последующей обработки), то в качестве предоставляемого интерфейса можно выделить метод, позволяющий внешнему окружению, получать текущее состояния прогресса генерации.

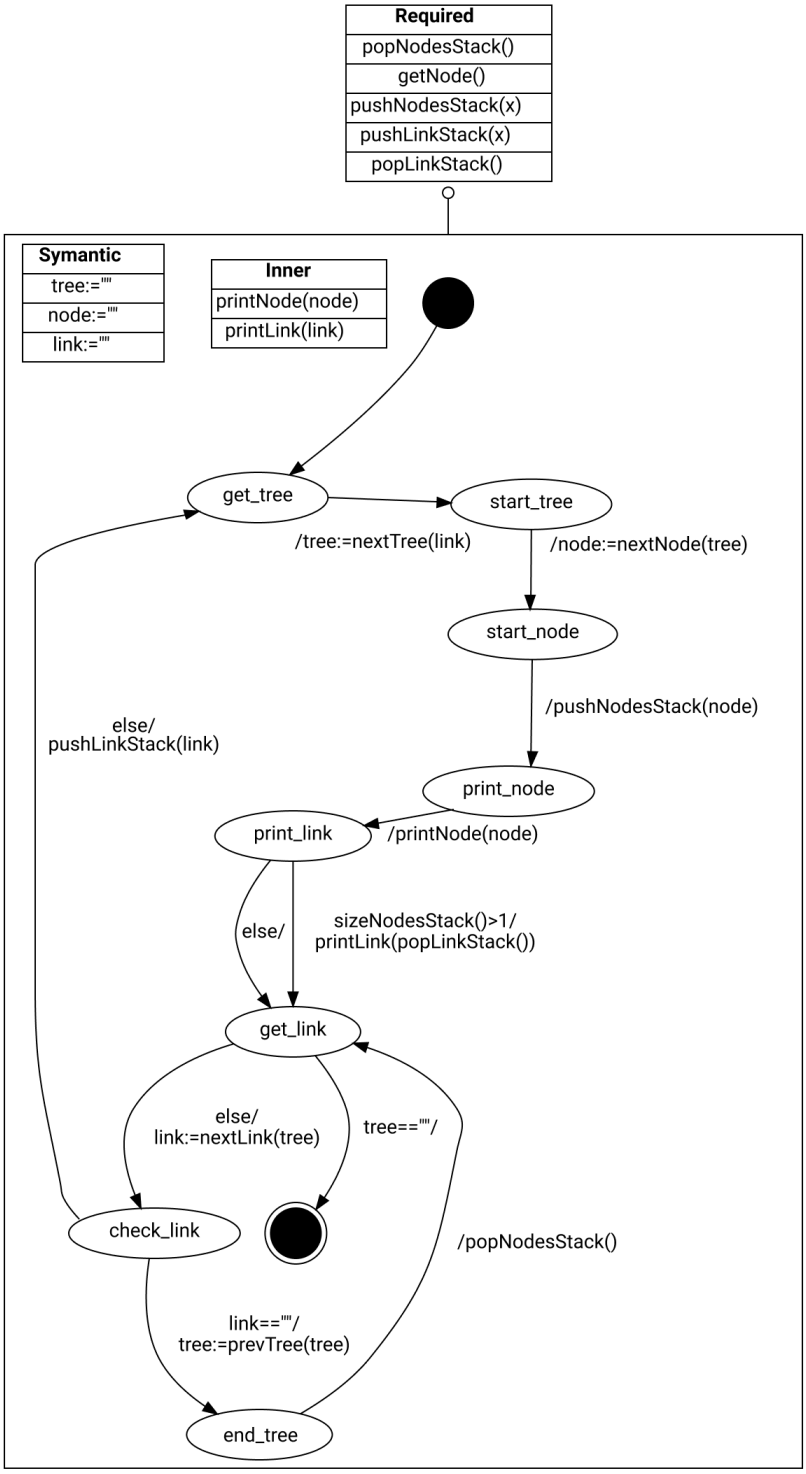


Рисунок 22. Обход по дереву на языке CIAO2v2
в графической нотации

Второй вариант описания семантики для этой же самой задачи будет выглядеть иначе. Если в предыдущем случае семантика описывалась одним автоматным объектом, взаимодействующим с внешним миром, то в данном примере будет представлено несколько автоматных объектов, взаимодействующих друг с другом.

Создадим для каждого дерева, узла и дуги при обходе дерева разбора свой автоматный объект. Внутри каждого объекта опишем стратегию обхода конкретно этого элемента. Иначе говоря, семантика для описания дерева “не знает” как нужно обходить узел или дугу. Аналогично, узел “не знает”, каким образом совершается обход в дереве и дуге, а дуга “не знает”, как происходит обход в узле и дереве. Однако, каждый автоматный объект зависит от обхода других элементов и может принимать сигналы о том, что нужно начать свой обход, или давать сигналы другим объектам о том, что им нужно начать обход. Например, автоматный объект, описывающий семантику дерева, будет находиться в ожидании, пока не удостоверится, что связанный с ним узел и все связанные дуги закончили свой обход. Аналогично, автоматный объект, описывающий поведение дуги, будет ожидать, пока не будет закончен обход по поддереву, связанным с ней. Иначе говоря, автоматные объекты должны взаимодействовать друг с другом, чтобы передавать команды дочерним узлам (эффект) и ожидать информации от дочерних узлов (сторожевое условие).

Диаграмма переходов для такого описания семантики представлена на рисунке 23.

Рассмотрим автоматный объект Tree. Из диаграммы видно, что данный объект предоставляет два метода интерфейса: entryTree() и ifExitTree(). При этом метод entryTree() будет использован другими объектами для изменения локальной переменной entryTreeFlag, ifExitTree() будет использован внешними объектами для запроса значения exitTreeFlag.

Заметим, что переход из начального состояния в состояние start_node происходит только в том случае, если переменная entryTreeFlag выставлена в True. Так как изначально данная переменная выставлена в False (в разделе VAR), то переход в состояние start_node будет выполнен

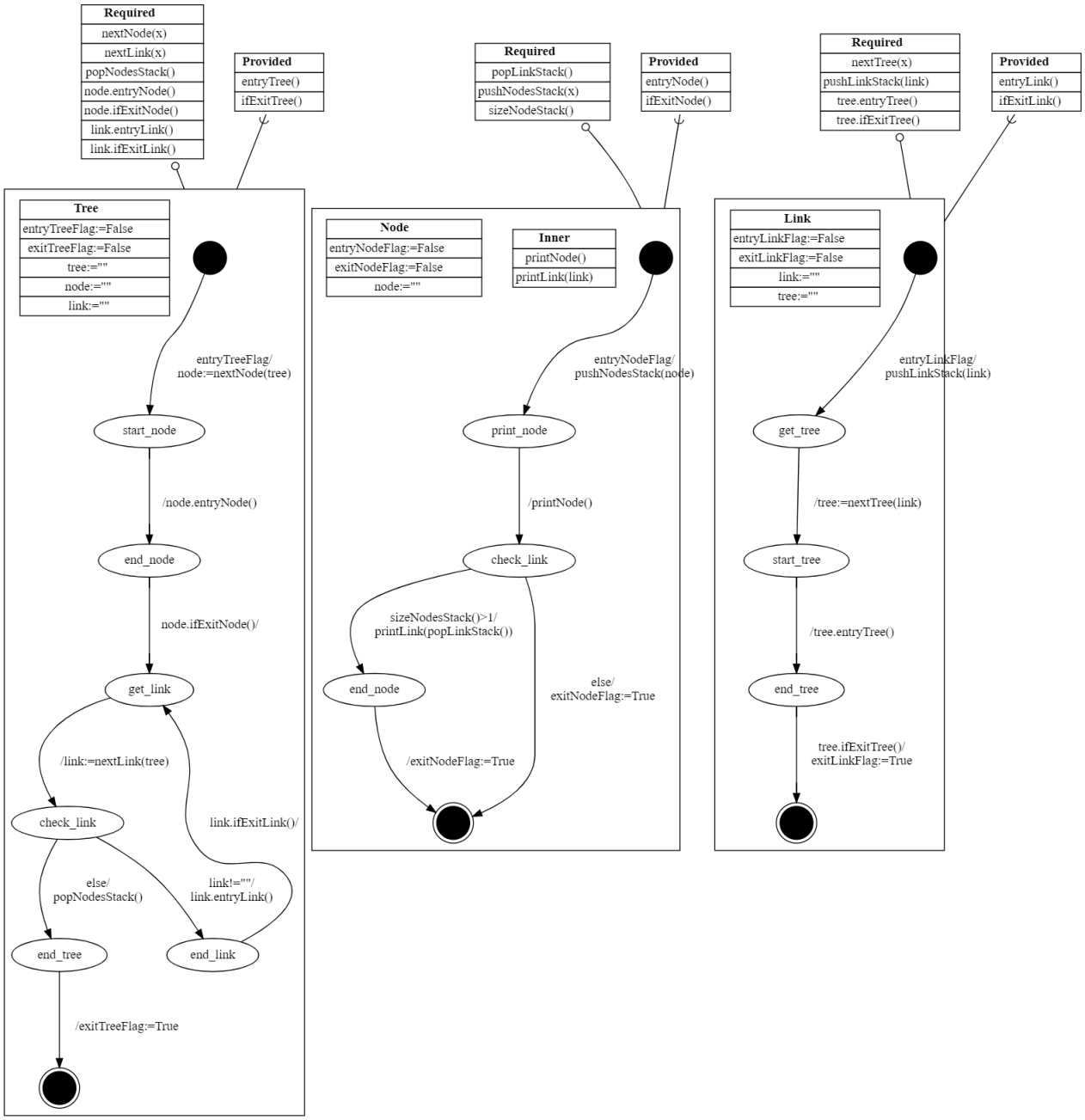


Рисунок 23. Обход по дереву на языке CIAO2v2 в графической нотации

только в том случае, когда внешняя среда изменит значение переменной entryTreeFlag с False на True. Иначе говоря, когда дереву будет дана команда начать обход.

Далее при переходе из состояния start_node в end_node автоматный объект дерева взаимодействует с узлом через предоставляемый интерфейс, а именно через метод entryNode(). Тем самым дерево передает команду узлу, что можно начинать обход. Таким образом дерево

производит «эффект», а узел принимает «событие» в терминах модели взаимодействующих автоматных объектов. Далее дерево переходит в ожидание, то есть сможет перейти в следующее состояние только в том случае, когда «сторожевое условие» ifExitNode() будет верным, то есть, когда узел закончит свой обход. Обход каждой дуги при этом будет аналогичен обходу узла: передаем дуге команду, что можно начинать, а затем ждем, что дуга закончит выполнение. При этом важно заметить, что исходное дерево запускает обход только в связанной с ним дуге и ничего “не знает” про то, что под каждой дугой есть дочернее поддерево.

Перед переходом в заключительное состояние автоматный объект выставляет значение exitTreeFlag в True. Таким образом автоматный объект сообщает о том, что он закончил обход.

Пример 3. Язык CIAOv2

Для демонстрации работоспособности и применимости инструмента традиционно применяют метод раскрутки: если на вход инструмента подать модель инструмента, спроектированную в этом инструменте, то на выходе мы получим тот же инструмент [31]. В качестве первого приближения к возможной раскрутке языка CIAOv2 можно сформировать диаграмму переходов, представленную на рисунке 24.

Эта диаграмма верхнеуровнево описывает семантику языка CIAOv2: перебираем условия, которые допустимы в текущем состояния (переменная currState); далее проверяем, верно ли это условие (используется внешний метод exec(string)); если нет, то возвращаемся назад, если да, то выполняем действие на переходе, обновляем текущее состояние и снова ищем следующий переход, перебирая условия. При этом текущее состояние, условие и действие имеют строковое представление. В данном примере предполагается, то внешний контекст

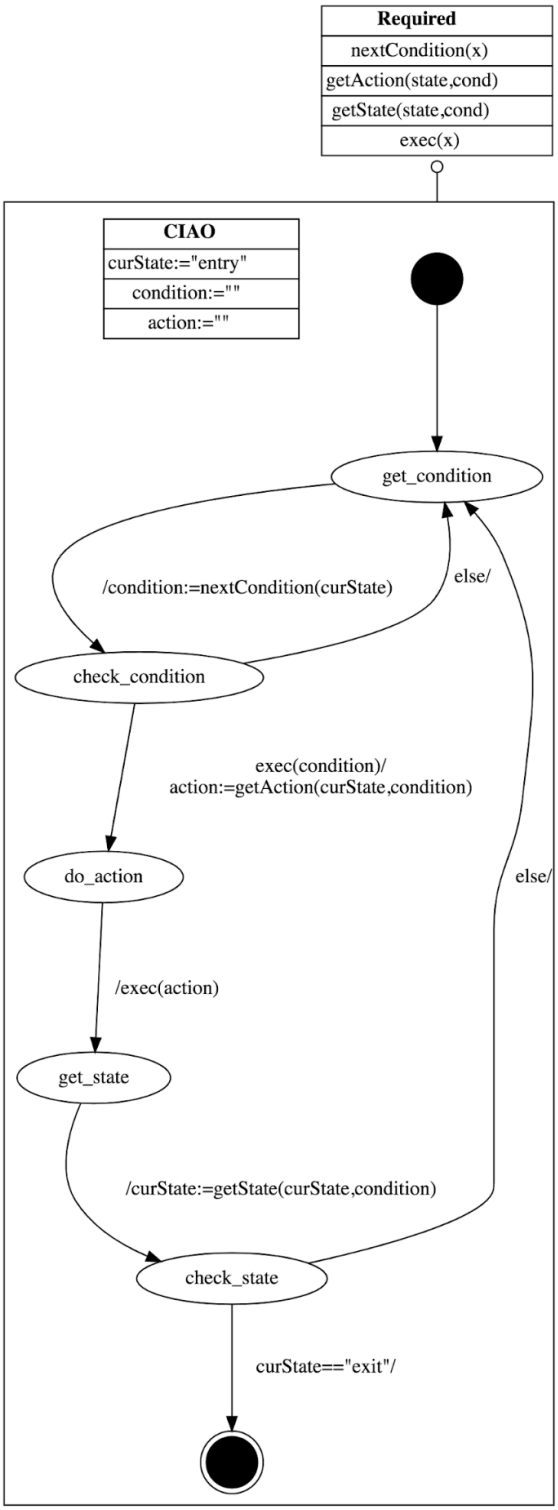


Рисунок 24. Графическое описание семантики CIAOv2 на языке CIAOv2

«умный», он умеет искать следующее условие по состоянию (nextCondition(state)), возвращать состояние (getState(state, condition)) и действие, которое находится на переходе в новое состояние (getAction(state, condition)).

Также есть возможность построить дереве разбора программы на языке CIAOv2 на основе грамматики. Дерево разбора для программы будет выглядеть следующим образом:

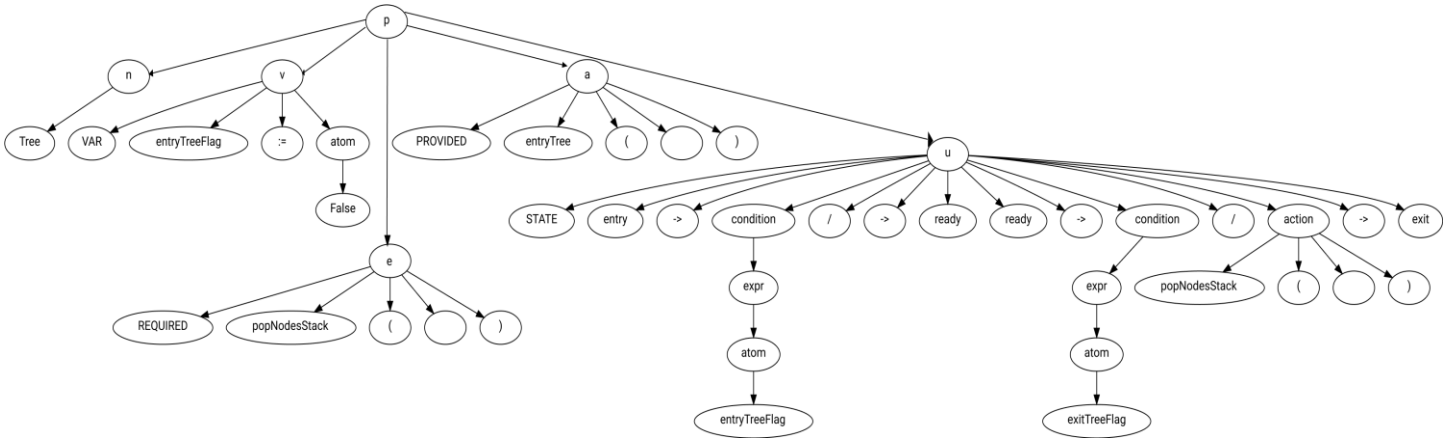


Рисунок 25. Дерево разбора программы из листинга 8

```
Tree
VAR
  entryTreeFlag := False
REQUIRED
  popNodesStack()
PROVIDED
  entryTree()
STATE
  entry -> entryTreeFlag / -> ready
  ready -> exitTreeFlag / popNodesStack() -> exit
```

Листинг 8. Пример программы на языке CIAOv2

Выводы к главе

1) Рассмотрен пример использования инструмента editor.cf в учебном процессе. Показано, что инструмент editor.cf предоставляет возможность описывать семантику одной задачи

различными способами.

2) Показана возможность самоприменимости инструмента editor.cf.

Заключение

В ходе выполнения данной работы были получены следующие результаты:

1. Изучена предметная область – методы и инструменты построения языков предметных областей
2. Предложен новый язык программирования для моделирования взаимодействия автоматных объектов CIAOv2.
3. Было разработано программное решение инструментальной поддержки конструирования DSL, где синтаксис описывается с помощью языка ANTLR, а семантика -- с помощью автоматных объектов. Данной решение предоставляет возможность:
   - Отображать дерево разбора программы на основе введенного синтаксиса
   - Отображать семантику языка в виде диаграммы переходов
   - Отображать семантику языка в виде шаблона программы на языке Python3
4. Проведена апробация и тестирования инструмента в рамках курса «Грамматики и автоматы» Санкт-Петербургского политехнического университета Петра Великого

Список использованных источников

1. Фаулер М. Предметно-ориентированные языки программирования. // М.: Вильямс, 2011.
2. Ward M. P., Language Oriented Programming // October 1994
3. Fauler M., Language Workbenches: The Killer-App for Domain Specific Languages? [Электронный ресурс] – URL: https://www.martinfowler.com/articles/languageWorkbench.html
4. Lämmel R., Software Languages: Syntax, Semantics, and Metaprogramming // 2006
5. Ахо А. В., Сети Р., Лам М. С, Компиляторы: принципы, технологии и инструменты // 1986
6. Feynman R., EBNF: A Notation to Describe Syntax // 2016
7. Карпов Ю. Г. Теория автоматов. // СПб.: Питер, 2002.
8. Л. Н. Федорченко, Извлечение крайней рекурсии из КСР-грамматики в системе SYNGT // / Труды СПИИРАН. Вып. 1, т. 1. — СПб.: СПИИРАН, 2002.
9. Лавров С.С., Программирование. Математические основы, средства, теория // БВХ-Петербург, 2001
10. Поликарпова Н. И., Шалыто А. А., Автоматное программирование // Санкт-Петербург, 2008
11. Atiskov A. Y., Novikov F. A., Fedorchenko L. N. et al., Ontology-Based Analysis of Cryptography Standards and Possibilities of Their Harmonization / // Theory and Practice of Cryptography Solutions for Secure Information Systems, 2013.
12. Meyer, B. Object-Oriented Software Construction // Prentice-Hall, 2000
13. Афанасьева И.В. Метод проектирования и реализации параллельных реагирующих систем // Диссертация на соискание ученой степени кандидата технических наук, 2018

14. Jefferson D., Evolution as a theme in Artificial Life // Los Angeles, California, 1990
15. Pollack J, Evolutionary Module Acquisition // The Second Annual Conference on Evolutionary Programming, La Jolla, California, 1993
16. ANother Tool for Language Recognition [Электронный ресурс] – URL: https://www.antlr.org/
17. Parr T., The Definitive ANTLR4 Reference // The Pragmatic Bookshelf, Dallas, Texas, 2012
18. Фаулер М. Предметно-ориентированные языки программирования. // М.: Вильямс, 2011.
19. Мотвани Раджив, Хопкрофт Джон Э., Ульман Джеффри Д, Введение в теорию автоматов, языков и вычислений // М.: Вильямс, 2008
20. Обзор IBM Rational [Электронный ресурс] – URL: https://www.ibm.com/developerworks/ru/rational/
21. Новичков А.Н, Rational Rose для разработчиков и ради разработчиков // 2000
22. Новые возможности Windows Workflow Foundation в .NET 4.5 [Электронный ресурс] – URL: https://docs.microsoft.com/ru-ru/dotnet/framework/windows-workflow-foundation/whats-new-in-wf-in-dotnet
23. Emden R., Drawing graphs with dot // 2015
24. Welcome to Graphviz [Электронный ресурс] – URL: https://www.graphviz.org/
25. JavaScript-библиотека для создания пользовательских интерфейсов [Электронный ресурс] – URL: https://ru.reactjs.org/
26. AceEditor [Электронный ресурс] – URL: https://ace.c9.io/
27. Flask’s documentation [Электронный ресурс] – URL: https://flask.palletsprojects.com/en/1.1.x/
28. Fowler M., A Language Workbench in Action – MPS // 12 June 2005

29. Vysoky P., Grammar to JetBrains MPS Convertor // Prague 2016
30. Anzanello M. Learning curve models and applications: literature review and research directions // International Journal of Industrial Ergonomics, 2011
31. Новиков Ф.А., Визуальное конструирование программ // Информационно-управляющие системы № 6, 2005

Приложение 1

Грамматика языка CIAOv2, записанная в ANTLR4

```
grammar ciao;

p : (name var? required? provided? inner? state)+ ;

name : ID ;

var : 'VAR' ( ID  ':=' atom )+ ;

required : 'REQUIRED' ( ID '(' variables ')' )+ ;

provided : 'PROVIDED' ( ID '(' variables ')' )+ ;

inner : 'INNER' ( ID '(' variables ')' )+ ;

state : 'STATE' (ID '->' ( condition )? '/' ( action )? '->'
ID)+ ;

condition :
   expr              #exprCondition
  ;

action :
  ID ':=' expr         #exprAction
  | ID '(' values ')' #funcAction
  ;

variables : ID? (',' ID)* ;

values : expr? (',' expr)* ;

expr :
   MINUS expr                          #unaryMinusExpr
```

```
 | NOT expr                                  #notExpr
 | expr op=(MULT | DIV) expr                 #multiplicationExpr
 | expr op=(PLUS | MINUS) expr               #additiveExpr
 | expr op=(LTEQ | GTEQ | LT | GT) expr      #relationalExpr
 | expr op=(EQ | NEQ) expr                   #equalityExpr
 | expr AND expr                             #andExpr
 | expr OR expr                              #orExpr
 | ID '(' values ')'                         #func
 | atom                                      #atomExpr
 ;

 atom
  : '(' expr ')'    #parExpr
  | (INT | FLOAT)   #numberAtom
  | (TRUE | FALSE)  #booleanAtom
  | ID              #idAtom
  | STRING          #stringAtom
  ;

OR : '||';
AND : '&&';
EQ : '==';
NEQ : '!=';
GT : '>';
LT : '<';
GTEQ : '>=';
LTEQ : '<=';
NOT : '!';
MINUS : '-';
PLUS : '+';
MULT : '*';
DIV : '/';

TRUE : 'True';
FALSE : 'False';
```

```
ID
 : [a-zA-Z_] [.a-zA-Z_0-9]*
 ;

INT
 : [0-9]+
 ;

FLOAT
 : [0-9]+ '.' [0-9]*
 | '.' [0-9]+
 ;

STRING
 : '"' (~["\r\n] | '""')* '"'
 ;

SPACE
: [ \t\r\n] -> skip
;
```

Приложение 2

В результате разработки серверной части было создан API - программный интерфейс приложения - со следующими методами:

- запрос дерева разбора

```
POST /ast
Content-Type: application/json
Body:
{
  "source": "<текст исходной программы>",
  "syntax": "<текст синтаксиса>"
}
Response:
{
  "error": "<код ошибки>",
  "info": "<страница для перенаправления или информация об ошибке>"
}
```

- запрос диаграммы состояний

```
POST /diagram
Content-Type: application/json
Body:
{
  "semantics": "<текст семантики>"
}
Response:
{
  "error": "<код ошибки>",
  "info": "<страница для перенаправления или информация об ошибке>"
}
```

- запрос заготовки кода

```
POST /code
Content-Type: application/json
Body:
{
  "semantics": "<текст семантики>"
}
Response:
{
  "error": "<код ошибки>",
  "info": "<страница для перенаправления или информация об ошибке>"
}
```

Приложение 3

Заготовка программы генерации деревьев на языке DOT (по рисунку 23).

```
class Tree:

    #VARS
    def __init__(self):
        self.entryTreeFlag = False
        self.exitTreeFlag = False
        self.tree = ""
        self.node = ""
        self.link = ""
        self.__state__ = "entry"

    #REQUIRED
    #uses
    # nextNode(x)
    # nextLink(x)
    # popNodesStack()
    # node.entryNode()
    # node.ifExitNode()
    # link.entryLink()
    # link.ifExitLink()

    #INNER
    #PROVIDED
    def entryTree(self):
        pass

    def ifExitTree(self):
        pass

    def run(self):
        while (True):
```

```
if self.__state__ == "exit":
    break
if self.__state__ == "entry" :
    if entryTreeFlag:
        node = nextNode(tree)
        self.__state__ = "start_node"
        continue
if self.__state__ == "start_node" :
    if True:
        node.entryNode()
        self.__state__ = "end_node"
        continue
if self.__state__ == "end_node" :
    if node.ifExitNode():
        self.__state__ = "get_link"
        continue
if self.__state__ == "get_link" :
    if True:
        link = nextLink(tree)
        self.__state__ = "check_link"
        continue
if self.__state__ == "check_link" :
    if link!="":
        link.entryLink()
        self.__state__ = "end_link"
        continue
    popNodesStack()
    self.__state__ = "end_tree"
    continue
if self.__state__ == "end_link" :
    if link.ifExitLink():
        self.__state__ = "get_link"
        continue
if self.__state__ == "end_tree" :
    if True:
```

```
                    exitTreeFlag = True
                    self.__state__ = "exit"
                    continue


class Node:

    #VARS
    def __init__(self):
        self.entryNodeFlag = False
        self.exitNodeFlag = False
        self.node = ""
        self.__state__ = "entry"

    #REQUIRED
    #uses
    # popLinkStack()
    # pushNodesStack(x)
    # sizeNodeStack()

    #INNER
    def __printNode(self):
        pass

    def __printLink(self, link):
        pass

    #PROVIDED
    def entryNode(self):
        pass

    def ifExitNode(self):
        pass

    def run(self):
```

```
        while (True):
            if self.__state__ == "exit":
                break
            if self.__state__ == "entry" :
                if entryNodeFlag:
                    pushNodesStack(node)
                    self.__state__ = "print_node"
                    continue
            if self.__state__ == "print_node" :
                if True:
                    self.__printNode()
                    self.__state__ = "check_link"
                    continue
            if self.__state__ == "check_link" :
                if sizeNodesStack()>1:
                    self.__printLink(popLinkStack())
                    self.__state__ = "end_node"
                    continue
                exitNodeFlag = True
                self.__state__ = "exit"
                continue
            if self.__state__ == "end_node" :
                if True:
                    exitNodeFlag = True
                    self.__state__ = "exit"
                    continue


class Link:

    #VARS
    def __init__(self):
        self.entryLinkFlag = False
        self.exitLinkFlag = False
        self.link = ""
```

```
    self.tree = ""
    self.__state__ = "entry"

#REQUIRED
#uses
# nextTree(x)
# pushLinkStack(link)
# tree.entryTree()
# tree.ifExitTree()

#INNER
#PROVIDED
def entryLink(self):
    pass

def ifExitLink(self):
    pass

def run(self):
    while (True):
        if self.__state__ == "exit":
            break
        if self.__state__ == "entry" :
            if entryLinkFlag:
                pushLinkStack(link)
                self.__state__ = "get_tree"
                continue
        if self.__state__ == "get_tree" :
            if True:
                tree = nextTree(link)
                self.__state__ = "start_tree"
                continue
        if self.__state__ == "start_tree" :
            if True:
                tree.entryTree()
```

```
            self.__state__ = "end_tree"
            continue
    if self.__state__ == "end_tree" :
        if tree.ifExitTree():
            exitLinkFlag = True
            self.__state__ = "exit"
            continue
```