**1 ИССЛЕДОВАТЕЛЬСКИЙ РАЗДЕЛ**

**1.1 Функциональное программирование**

*Функциональное программирование (ФП)* является одной из парадигм программирования, используемых в современной разработке программного обеспечения. Ее суть заключается в том, что программа представляется в виде последовательности вычисления математических функций.

Математико-логической базой для функциональной парадигмы программирования является теория *лямбда-исчисления (*англ*. λ-calculus, lambda-calculus)*, разработанная выдающимся американским математиком Алонзо Черчем в 1930х годах. Эта формальная система позволяет описывать вычисления, используя всего два оператора:

* *Абстракция* - используется для построения функций по выражениям.
* *Аппликация* - используется для подстановки выражений вместо переменных и вычисления результата.

Первоначальная версия лямбда-исчисления не накладывала ограничения на типы аргументов. В 1940 году Алонзо Черчу пришлось доработать свою теорию и добавить в нее механизм *простой типизации*, поскольку описанный пятью годами ранее парадокс Клини-Россера доказывал внутреннюю противоречивость нетипизированного лямбда-исчисления.

Несмотря на кажущуюся простоту, лямбда-исчисление является мощнейшим инструментом для построения систем и описания алгоритмов, обладающим *полнотой по Тьюрингу*. Это означает, что на его основе можно создать алгоритм для вычисления любого выражения, которое формально вычислимо. Следовательно, ФП как таковое также является полным по Тьюрингу.

Первым языком программирования, напрямую основанный на λ-исчислении и воплощающих идеи ФП в реальности, стал язык *LISP* (аббревиатура от англ. List Processing, обработка списков). Он был разработан Джоном Маккарти в 1958 году и является одним из старейших языков программирования высокого уровня, уступая только языку Fortran. Основной структурой данных, как можно догадаться по названию, является связанный список: в виде связанных списков представлен даже исходный код программы, что позволяет оперировать им наравне с данными и создавать самомодифицирующиеся программы. Эта особенность способствовала использованию LISP в качестве основного языка для исследований в области искусственного интеллекта и самообучающихся систем. Многие из концепций, впервые появившиеся в LISP (например, сборка мусора и широчайшие возможности макропрограммирования), были революционными для своего времени, поэтому LISP имеет множество современных независимых реализаций и до сих пор используется в практической разработке.

Еще одним крупным семейством функциональных ЯП является язык *ML* (аббревиатура от *Meta Language*, также Milner Language по фамилии создателя, Робина Милнера) и его многочисленные потомки. Первоначально ML разрабатывался в качестве языка для автоматического доказательства теорем, однако впоследствии оказалось, что его можно эффективно применять для написания вычислительных программ. ML - язык со строгой статической типизацией, однако использование алгоритма Хиндли-Милнера позволяет избавить код от излишних аннотаций типов, если компилятор может вывести их автоматически. Он поддерживает многие удобные для программистов возможности, такие как автоматическая сборка мусора, параметрический полиморфизм, алгебраические типы и сопоставление с образцом. На основе ML впоследствии были разработаны такие известные языки, как F#, Haskell и OCaml.

Долгое время ФП использовалось в основном для академического обучения программированию, в научных трудах и теориях, однако не было активно использовано в практической разработке программного обеспечения. Основной причиной является тот факт, что по историческим причинам техническая реализация современных вычислительных систем ближе к машине Тьюринга: данные изменяемы и хранятся в ячейках памяти, процессор последовательно исполняет элементарные инструкции. ФП же является более высокоуровневой концепцией, поэтому его реализация поверх традиционной архитектуры неизбежно вызывает накладные расходы, влияющие на производительность. Однако, по мере того, как вычислительные мощности растут, а стоимость аппаратного обеспечения становится несоизмеримо меньше стоимости поддержания и развития программного кода, функциональная парадигма становится все более и более актуальной.

Для того, чтобы понять, почему функциональный код проще поддерживать, чем императивный, нужно рассмотреть понятие "математическая функция". В отличие от традиционной записи алгоритма в виде последовательности каких-либо действий, математическая функция оперирует только своими параметрами и возвращает какое-либо значение. Математические функции также называются чистыми функциями потому, что они не воздействуют на окружающую их среду, не имеют побочных эффектов.

Наиболее распространенным видом побочных эффектов является хранение и изменение состояния по мере вычисления значения функции. Наличие состояния и изменяемых переменных предполагает, что вызывая одну и ту же функцию дважды с одинаковыми параметрами, мы можем получить разные результаты, поскольку состояние может измениться при первом вызове и быть использовано при вычислении результата второго вызова.

При использовании чисто функционального подхода обычно предполагается, что результат функции зависит только от ее параметров - сколько бы раз функция ни вызывалась с одними и теми же параметрами, она всегда будет возвращать одно и то же значение. Следствием такого подхода является детерминированность и идемпотентность процесса вычисления, которая несет в себе несколько полезных свойств, таких как:

* *Упрощение тестирования*. Для того, чтобы протестировать корректность работы функции, достаточно выбрать несколько характерных наборов параметров и сравнить результат с соответствующими эталонными значениями.
* *Облегчение понимания программы и ее отладки*. При изучении нового кода или при отладке программисту, работающему с кодом в функциональном стиле, не придется держать в уме все возможные изменения хода работы программы, зависящие от внешних факторов.
* *Отделение высокоуровневой логики от реализации*. Функциональный подход позволяет сфокусировать внимание программиста на логике работы программы, не вдаваясь в технические подробности и детали низкоуровневой реализации используемых возможностей. Программист формулирует задачу в виде того, какой ответ он хочет получить, позволяя компилятору и среде исполнения выбрать оптимальный конкретный способ его вычисления. Одним из наиболее важных следствий этого является возможность легко переносить код на новые платформы и распараллеливать его на несколько вычислительных ядер: при отсутствии состояния программы последовательность вычисления промежуточных значений не играет никакой роли.
* *Возможность кеширования промежуточных результатов*. Отсутствие побочных эффектов позволяет производить над кодом различные оптимизации, напрямую влияющие на скорость его работы. Например, функции без параметров, по сути, являются константами и их вызовы могут быть заменены на соответствующие значения еще на этапе компиляции программы. Кроме того, чистота функций дает возможность кешировать результаты их работы: после вызова функции значения аргументов и результат ее работы может сохраняться на будущее, тогда при повторном вызове с теми же параметрами значение уже будет заранее известно и повторное вычисление не потребуется. Эта техника также называется *мемоизацией* функции.

В языках, поддерживающих парадигму ФП, функции обычно представлены как *объекты первого класса*: как и любые другие объекты, они могут быть созданы во время исполнения программы, сохранены в структуру, возвращены из одной функции и переданы в качестве параметра в другую. Функции, принимающие другие функции в качестве параметров, называются *функциями высшего порядка* (*higher order functions*).

Использование функций высшего порядка позволяет языку программирования быть очень гибким и лаконичным. Например, многие языки, поддерживающие частичное применение, рассматривают функцию с двумя параметрами как функцию, принимающую один параметр и возвращающую функцию, принимающую второй параметр и возвращающую значение. Аналогичным образом можно рассматривать функцию любой арности. Такой подход позволяет создавать новые функции как частные случаи уже существующих, фиксируя значения некоторых из их параметров, и называется *частичным применением* (или *каррированием,* англ*. currying* в честь американского математика Хаскелла Карри).

Возможность передавать функции в качестве параметров существенным образом сокращает количество рутинного, многократно повторяемого кода. Например, типичная операция фильтрации списка объектов по какому-либо критерию обычно реализуется следующим образом:

1. Создается список, в который будут попадать нужные объекты.
2. Существующий список обходится в цикле.
3. Для каждого из объектов вычисляется условие.
4. Если условие истинно, объект добавляется в новый список.
5. По окончанию цикла созданный список возвращается.

Такой код может встречаться во множестве мест в программе. Он подвержен трудноуловимым логическим ошибкам, вызванным опечатками. Функциональный подход позволяет описать один общий метод фильтрации любых коллекций, который принимает исходный список и условие фильтрации в качестве функции. Такой подход оказался настолько удобным и популярным, что его поддержка добавляется в языки, изначально не считающиеся функциональными, например LINQв C#, портированные версии для PHP, Python, Ruby и других.

Для того, чтобы эффективно использовать функции высшего порядка, язык должен поддерживать механизм *лексических замыканий*. Если одна функция была создана внутри другой, он позволяет вложенной функции оперировать конкретными значениями параметров родительской функции. При повторном вызове родительской функции будет создан новый экземпляр дочерней и она замкнет новые значения параметров, в то время как ранее созданная дочерняя функция по прежнему будет оперировать теми значениями, которые ей были доступны на момент ее создания. Более наглядно этот механизм можно представить в виде кода на языке C#:

Func<int, int> AdderFactory(int value)

{

return x => x + value;

}  
var addTwo = AdderFactory(2);

var addFive = AdderFactory(5);

var ten = addTwo(addFive(3));

В данном примере метод AdderFactory - фабричный метод, возвращающий функции, прибавляющие к параметру заранее заданное число. При первом его вызове параметр value был равен двум, поэтому функция addTwo будет всегда прибавлять число 2 к своему аргументу, сколько бы раз AdderFactory ни вызывалась после этого. Замыкание обеспечивает защиту от неумышленного изменения значений параметров, поскольку поменять однажды захваченное значение невозможно.

Еще одной важной концепцией в ФП является понятие *рекурсии*. Функция может быть описана рекурсивно при соблюдении двух условий:

1. Наличие *базовых значений аргументов*, для которых результат функции заранее известен.
2. Наличие *правил*, позволяющих выразить значение функции для других значений аргументов через операции над базовыми значениями.

Классическим примером рекурсивной функции является алгоритм вычисления n-ного *числа Фибоначчи*. Два первых числа в последовательности известны, остальные же могут быть получены как сумма двух предыдущих. Такое описание позволяет вычислить значение любого члена последовательности, поскольку он рано или поздно будет выражен через значения первых, многократно сложенных между собой. Рекурсия используется для описания множества известных алгоритмов: нахождение наибольшего общего делителя, двоичный поиск, алгоритм Ньютона, быстрая сортировка и сортировка слиянием могут быть описаны рекурсивно.

Рекурсия может применяться не только в описании алгоритмов, но и в описании структур данных. Например, описание дерева требует рекурсивной структуры «узел», которая содержит какое-либо полезное значение и указатели на такие же узлы, расположенные ниже данного. Обработка рекурсивных структур данных требует рекурсивных алгоритмов.

В мире ФП рекурсия является заменой для циклических конструкций из императивных языков программирования. Императивная организация цикла подразумевает либо наличие изменяемой переменной-счетчика, либо функции-условия, которая рано или поздно меняет свое значение в результате побочного эффекта, вызванного кодом в теле цикла (в противном случае циклы всегда были бы бесконечными).

В общем случае рекурсивный алгоритм начинается со спуска в глубину, который продолжается до тех пор, пока для всех параметров не найдутся нерекурсивно вычисляемые значения. Далее эти значения возвращаются вызывающей стороне, проходя по списку вызовов в обратном порядке, результатом чего становится вычисление значения изначально запрошенной функции. Для хранения списка вызовов используется стековая структура данных, поэтому глубина рекурсии ограничивается памятью, выделенной под стек.

Если бы рекурсия всегда требовала объем памяти, зависящий от количества итераций, это бы очень сильно ограничило круг практических задач, которые возможно решать методами ФП. Однако в некоторых случаях запоминать адрес возврата не требуется: если за рекурсивным вызовом сразу следует возврат полученного из него значения и никакие действия над этим значением не производятся, его можно просто оставить на стеке. Эта техника носит название *хвостовой рекурсии* (tail recursion) и используется многими оптимизирующими компиляторами функциональных ЯП, позволяя программистам писать высокоэффективный код, обходясь без циклов и переменных состояния.

В зависимости от контекста использования, операция рекурсивного вызова может быть использована для описания как обычной математической рекурсии, так и дуальной ей операции *корекурсии*: первая является сверткой и преобразует список значений в одно значение, вторая же, наоборот, создает список из значения. Функциональные языки поддерживают создание бесконечных корекурсивных последовательностей с помощью механизма *ленивого вычисления*. Он означает, что компьютер вычислит только тот минимум значений, который необходим для получения конечного результата. Таким образом, можно задать *правило* для построения бесконечной последовательности значений (например, все целые числа) и применять к нему оператор сложения столько раз, сколько значений нужно сложить. Императивный подход вне зависимости от желаемого количества складываемых элементов потребовал бы сначала создать массив, содержащий все значения бесконечной последовательности, что неизбежно привело бы к заполнению всей имеющейся памяти.

Разумеется, функциональная парадигма программирования не лишена недостатков. Основной из них является следствием основного достоинства функционального подхода: чистота функций и отсутствие побочных эффектов, по сути, запрещают программе любое взаимодействие с окружающей средой, будь то вывод информации на экран, запись в файл, пересылка по сети или интерактивное взаимодействие с пользователем. Для того, чтобы обойти это ограничение, не лишая программиста возможности писать лаконичный и эффективный код, создатели языка Haskell ввели понятие *монад*, позаимствовав его из теории категорий. *Монада* - особая среда-контейнер, управляющая вычислением выражений внутри себя. Например, все операции с вводом-выводом производятся в монаде *IO*, которая последовательно вычисляет все выражения, что позволяет избежать непредсказуемых результатов при использовании ввода-вывода в среде ленивого исполнения. Монада *Maybe* позволяет обработать возможные ошибки в блоке кода так, что при возникновении первой весь оставшийся набор выражений пропускается и возвращается флаг, указывающий на отсутствие значения. При успешном выполнении всех выражений будет возвращено значение последнего. Существует также монада *State*, позволяющая эмулировать изменяемое состояние функции. В языке F# аналогичная конструкция называется *вычислительным выражением* (англ. *computation expression*, или *workflow*) и используется для создания последовательностей, распараллеливания кода и т.д.

Другим недостатком функциональных языков можно назвать *сложность реализации* компиляторов и интерпретаторов. ФП подразумевает наличие автоматического сборщика мусора, требует довольно сложной грамматики, поддержки замыканий, механизма вывода типов, в некоторых случаях также оптимизации хвостовой рекурсии. Кроме того, написанные в функциональном стиле программы могут быть сложны для понимания начинающими программистами, не имеющими достаточной математической культуры или соответствующего образа мышления.

В свете вышесказанного становится понятно, почему большая часть функциональных языков программирования реализует только часть возможностей ФП, приходя к компромиссу между удобством написания пользовательских программ, теоретической "правильностью" самой системы и сложностью построения компилятора. Оптимальным решением является сохранение возможностей императивного программирования с использованием ленивых вычислений, функций высшего порядка и неизменяемых переменных.

**1.2. Встраиваемые языки сценариев**

*Языки сценариев*, также называемые *скриптовыми языками* (от англ. script), широко используются в современном программном обеспечении. Их основное назначение - давать пользователю возможность автоматизировать рутинные действия при работе с программой, описав их последовательность или алгоритм однократно и позволяя интерпретатору выполнять его автоматически. Скриптовый язык предоставляет доступ к функциям системы, в которую он встроен. Формального различия между скриптовым языком и языком программирования не существует, однако в большинстве своем скриптовые языки являются языками высокого уровня с легковесным лаконичным синтаксисом, работающими посредством интерпретатора или jit-компилятора. В некотором смысле их можно считать расширенной версией макросов.

Скриптовые языки могут применяться практически в любой программе. Например, практически любая командная строка (будь то cmd, bash, zsh или любые другие современные терминалы) позволяет сохранять список команд в файл (например, .bat или .sh) и впоследствии выполнять его автоматически. Пакет Microsoft Office предоставляет возможности для написания сценариев на языке *Visual Basic for Applications* (VBA). В продуктах компании Adobe для автоматизации используется язык, основанный на стандарте *EcmaScript*.

Существует несколько популярных языков, которые можно использовать в качестве встраиваемых при разработке собственного программного продукта.

**1.2.1 JavaScript**

Язык *JavaScript* (JS) был разработан Бренданом Айком в 1995 году для использования в браузере Netscape. Первоначально его основным предназначением было добавление интерактивности на веб-страницы: анимация, валидация формы и прочие действия с элементами DOM-дерева. Долгое время являясь единственным скриптовым языком, чье подмножество поддерживается всеми существующими браузерами, он исторически стал стандартом де-факто для разработки веб-приложений и неотъемлимой частью Web 2.0 и связки HTML5. Для JS написано несколько высокоэффективных реализаций, таких как V8 от компании Google или TraceMonkey от компании Mozilla.

Спонтанное развитие языка не могло не сказаться на его архитектуре. Спецификация была принята спустя два года после его создания и описывала поведение существующей реализации, далеко не всегда логичное. Использование слабой динамической типизации во многих случаях дает абсолютно непредсказуемые результаты, дополняемые различиями в реализации языка в разных браузерах и общей неортогональностью системы. По этим причинам JS подвергается постоянной критике, однако поскольку альтернатив ему по историческим причинам нет, он до сих пор остается одним из самых распространенных и востребованных языков в мире.

**1.2.2 TCL**

Язык *TCL* (аббревиатура от *Tool Command Language*, "Язык инструментальных команд", читается "*тикль*") является одним из старейших скриптовых языков в мире. Первая версия была выпущена Джоном Оустерхаутом в 1988 году во время его работы в университете Беркли. TCL набрал пик популярности к 1993 году, будучи используемым в качестве встраиваемого языка и языка для консольных команд во множестве программных продуктов. Его популярность резко упала с появлением интереса к Java-технологиям, однако он до сих пор применяется в таких программных пакетах и комплексах, как PostgreSQL, Verilog, VHDL. К недостаткам языка можно отнести отсутствие стандартной библиотеки, которое вкупе с гибким синтаксисом вызвало появление множества разрозненных сторонних инструментариев, а также отсутствие объектно-ориентированных возможностей и непривычный синтаксис.

**1.2.3 Lua**

Язык Lua ведет свою историю с 1993 года, когда его первая версия была написана Робертом Иерусалимски, работавшим в то время в Технологической Группе Компьютерной графики (TecGraf) при Католическом Университете Рио-де-Жанейро. В эти времена в Бразилии был запрет на импорт компьютеров и программного обеспечения, поэтому в TecGraf было принято решение о разработке собственного языка программирования. Основным принципом был простой, лаконичный и гибкий синтаксис. Со временем Lua стал использоваться и за стенами Католического Университета благодаря высокой скорости работы и свободно распространяемому переносимому исходному коду на языке C. Недостатками языка можно считать отсутствие объектно-ориентированных возможностей, а также нестрогую динамическую типизацию.

**1.2.4 Python**

Будучи в большей степени языком программирования, нежели языком сценариев, Python все же широко используется в качестве встраиваемого языка. В основе дизайна языка лежит философия "*есть только один (хороший) способ сделать что-либо*" ("there's only one (good) way to do it") - средства языка способствуют написанию лаконичного, надежного и понятного кода. Python поддерживает многие удобные конструкции и возможности, такие как генераторы списков и последовательностей, ленивые вычисления, функции как объекты первого класса, декораторы. Python часто применяется в качестве языка для написания программ для математических расчетов, поскольку интерпретатор работает очень быстро и позволяет писать расширения на языке С, обеспечивающие минимально возможное количество побочно исполняемого кода.

Использование динамической типизации делает код на языке Python весьма лаконичным, однако не позволяет проводить статический анализ кода на момент компиляции. Кроме того, по той же причине автодополнение кода ограничивается лишь несколькими базовыми возможностями. Другой проблемой языка Python можно назвать недавний выход версии 3.0, в которой была нарушена обратная совместимость, в результате чего практически весь старый код необходимо переписывать.

1.3 Обзор средств анализа и разбора исходного текста, генераторов байткода .NET

Платформа .NET является современной промежуточной средой для запуска приложений, разрабатываемой компанией Microsoft. .NET реализована в виде *виртуальной машины*, которая исполняет команды, написанные на специальном языке CIL (Common Intermediate Language, Общий Промежуточный Язык). Поскольку многие сложные аспекты реализации среды исполнения программ берутся на себя виртуальной машиной (например, управление памятью), код, написанный на CIL, называется *управляемым*[8,10].

Как показано на рисунке 1.6, структура фреймворка .NET состоит из множества уровней. Языки программирования для платформы .NET занимают верхний уровень, и, следовательно, могут использовать большое количество функций, чья сложная реализация скрыта на низших уровнях.

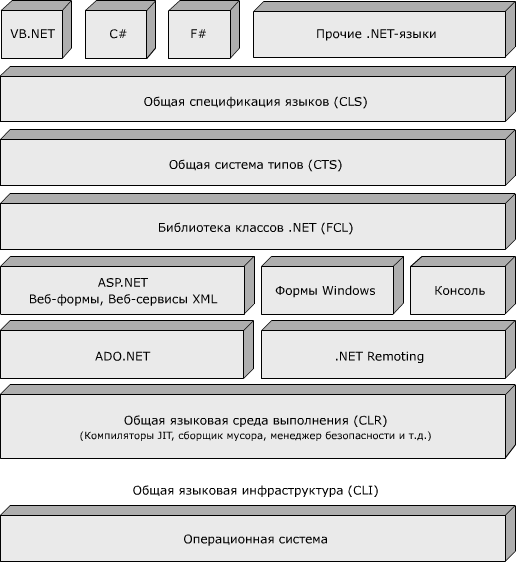


Рисунок 1.1 – Общая структура фреймворка .NET.

Данная виртуальная машина предоставляет программисту широкий спектр средств и возможностей, облегчающих написание программ:

* Виртуальная машина .NET основана на стеке[8], что упрощает написание программ под нее.
* Объектно-ориентированные возможности поддерживаются самой виртуальной машиной (например, создание новых объектов, виртуальные методы, наследование и интерфейсы), что позволяет беспрепятственно использовать парадигму ООП в новых разрабатываемых языках.
* *Сборка мусора* осуществляется автоматически, программы надежно защищены от утечек памяти по вине программиста.
* В .NET встроен *Just-In-Time-компилятор*, преобразующий байт-код в машинные команды конкретного процессора, на котором запускается программа[8,10]. Это обеспечивает программам, написанным под .NET, такое же быстродействие, как у программ, написанных не на управляемом коде, а в некоторых случаях даже превосходящее их за счет использования эффективных оптимизаций. За счет использования байт-кода программы для .NET становятся переносимыми на любую аппаратную платформу, где реализована среда исполнения .NET.
* В комплекте поставляется поистине огромная *стандартная библиотека*, позволяющая работать с различными форматами файлов, подключаться к базам данных, использовать сеть, отображать информацию на экране в различных форматах и использовать другие функции. В интернете существует множество качественных компонентов, коммерческих и бесплатных, реализующих дополнительный функционал.
* Программы под .NET могут быть написаны на разных языках, но они все равно смогут беспрепятственно взаимодействовать между собой, поскольку все .NET-совместимые языки основаны на общеязыковой инфраструктуре (Common Language Infrastructure, CLI).
* Для .NET также существуют *мощные средства для разработки* и отладки, такие как Visual Studio, Resharper и .NET Reflector.
* Существует *свободная кроссплатформенная реализация* среды исполнения .NET, которая называющаяся Mono. Она позволяет запускать программы под среду .NET в операционных системах, отличных от Microsoft Windows.

Языковой базой для написания дипломного проекта была выбрана связка C# и F#. Первый является основным языком разработки под платформу .NET, второй же имеет широкие возможности для написания синтаксических анализаторов и глубоко интегрируется с первым. Были рассмотрены следующие средства анализа и разбора исходного кода:

* ANTLR - один из старейших и наиболее продвинутых комплексов для написания парсеров под C#. Имеет встроенную среду разработки грамматик с подсказками и отладчиком. К сожалению, сгенерированные парсеры имеют низкую скорость работы и проблемы с разбором грамматик типа LL(n), где n > 1.
* Coco/R - генератор синтаксических анализаторов под несколько языков, в том числе C#. Обладает очень сложным и трудночитаемым синтаксисом описания грамматик.
* MinousseCC - портированная версия JavaCC для языка C#. Проект так и не был завершен, последняя версия датируется 2005 годом.
* FParsec - портированная на язык F# версия мощнейшей библиотеки parsec, изначально написанной на для написания парсеров на языке Haskell. Благодаря гибкости синтаксиса языка F# позволяет описывать грамматики в форме, очень похожей на РБНФ. Также имеется поддержка грамматик, в которых блоки кода выделяются отступами.

Таким образом, было решено использовать библиотеку FParsec для написания синтаксического анализатора. Ее малый вес позволяет включить ее в получаемую сборку с помощью утилиты ILMerge.

В качестве backend'а для генерации IL-кода была выбрана библиотека System.Reflection.Emit, поскольку она входит в состав .NET и не требует дополнительных зависимостей. В ее пользу также говорит ее популярность: на ее основе написан компилятор для языка F#, входящего в стандартную поставку средств разработки под .NET от Microsoft.

**2 СПЕЦИАЛЬНЫЙ РАЗДЕЛ**

**2.1 Общая структура синтаксического анализатора**

Основной задачей любого транслятора является отображение передаваемой исходной программы в семантически эквивалентную ей программу на целевом языке. Построение этого отображения можно логически поделить на два процесса: *анализ* и *синтез*.

В ходе фазы *анализа* программа разбивается на составные части, на которые накладывается грамматическая структура. Впоследствии эта структура используется для создания промежуточного представления исходной программы. Если анализатор обнаруживает, что программа неверно составлена синтаксически либо имеет семантический дефект, он должен выдать информативные сообщения об ошибках, чтобы пользователь мог их исправить. Анализ также собирает информацию об исходной программе в виде структуры данных, называемой таблицей символов, которая вместе с промежуточным представлением используется в фазе синтеза.

*Синтез* строит требуемую целевую программу на основе промежуточного представления и информации из таблицы символов. В англоязычной литературе фазу анализа также называют *начальной* *(front-end)*, а фазу синтеза – *заключительной (back-end)*.

Поскольку построение синтаксически управляемого транслятора является одной из наиболее сложных задач в информатике, для упрощения традиционно используется метод декомпозиции трансляции на *слои*, также называемые *проходами*. Каждый проход преобразует одно представление исходной программы в другое, более удобное для следующего прохода, пока заключительный синтезирующий слой не даст на выходе целевую программу.

Типичное разложение компилятора на проходы приведено на рисунке 2.1. На практике некоторые проходы могут объединяться, а межфазное промежуточное представление может не строиться явно. Таблицы символов используются всеми проходами.

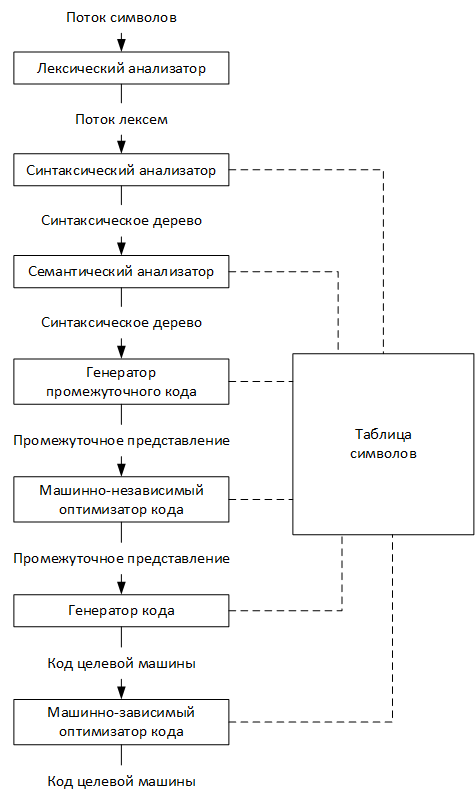


Рисунок 2.1: общая схема устройства компилятора

Некоторые компиляторы содержат фазу машинно-независимой оптимизации между анализом и синтезом. Назначение данной фазы – преобразовать промежуточное представление в вид, из которого синтез мог бы сгенерировать более эффективную программу. Критериями эффективности могут служить быстродействие, минимальный размер кода, минимальные требования к памяти и другим ресурсам. Поскольку оптимизация не обязательна для получения целевой программы, некоторые представленные на рисунке 2.1 фазы компилятора могут отсутствовать в различных реализациях.

**2.1.1 Традиционная схема четырехпроходного транслятора**

Первым проходом компилятора обычно является фаза *лексического анализа*, или *сканирования*. Программа на исходном языке представляется в виде потока символов, однако один отдельный символ – слишком маленькая единица информации для того, чтобы ей удобно было оперировать при анализе программы. Лексический анализатор принимает поступающий на вход транслятора поток символов и группирует эти символы в более крупные значащие последовательности, называемые *лексемами*. Каждая лексема имеет следующие свойства:

* Тип лексемы
* Расположение лексемы в исходном тексте программы (требуется для указания места при отображении сообщений об ошибках)
* Значение атрибута лексемы (опционально)

При прохождении программы через лексический анализатор комментарии вырезаются, незначащие символы (например, пробел или символ табуляции в зависимости от грамматики) игнорируются.

Для построения лексического анализатора необходимо знать полный список всех возможных лексем, используемых в грамматике языка. Каждой лексеме сопоставляется *терминал* – символ или последовательность символов из алфавита языка, однозначно определяющая лексему.

Второй проход компилятора – *синтаксический анализ*, также называемый *разбором* (англ. *parsing*). Синтаксический анализатор использует поток лексем и в соответствии с грамматикой языка формирует из них второе промежуточное представление – *синтаксическое дерево*. Каждый внутренний узел дерева представляет собой некоторую операцию, а его дочерние узлы – аргументы этой операции.

Грамматика языка является основой синтаксически управляемого перевода. Она состоит из терминалов, которые используются в лексемах, и правил их сочетания в группы, называемые *нетерминалами*. Более подробно терминалы и нетерминалы будут описаны в разделе 2.1.2.

Синтаксический анализ может выявить несоответствие программы грамматике языка, на котором она написана. Поскольку некорректная программа не может быть преобразована, процесс трансляции завершается, а пользователю показывается сообщение об ошибке.

За третий проход компилятора отвечает *семантический анализатор*. Он использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или таблицах символов, чтобы использовать ее в процессе генерации целевой программы или промежуточного кода.

Важной частью семантического анализатора является *проверка типов*. Эта проверка определяет, соответствуют ли фактические типы аргументов тем, которые ожидает оператор. Например, многие языки программирования требуют, чтобы индекс массива был целым числом; на этапе семантического анализа компилятор должен сообщить об ошибке, если в качестве индекса передано выражение, имеющее тип «строка» или «число с плавающей точкой».

Спецификация языка может позволять определенные преобразования типов, которые называются *приведениями* (англ. coercion). Например, любой бинарный арифметический оператор может быть применен к паре целых чисел, или к паре чисел с плавающей точкой. Если один из операндов является целочисленным, а другой – дробным, компилятор может автоматически выполнить преобразование целого числа к дробному, не требуя явного указания этого в тексте программы. Очевидно, что результат выполнения данного оператора будет иметь наиболее общий тип среди типов его операндов. Преобразование числового типа в более широкий без потерь называется *расширением типа*.

**2.1.2 Теория формальных языков**

Математической базой для синтаксически управляемого перевода является теория формальных языков. В математической логике и информатике формальным языком называется совокупность следующих элементов:

* Терминалы - множество символов ∑, составляющих алфавит языка. Записываются прописными буквами (*a, b*).
* Нетерминалы - множество слов *Ν*, которые можно составить из символов и других слов. Записываются заглавными буквами (*A, B*).
* Правила вывода слов из других слов и символов, записываемые символом →. Слева от символа стрелки указывается выводимый нетерминал, а справа – последовательность терминалов и нетерминалов, в которую этот нетерминал разворачивается. Каждый объявленный нетерминал должен иметь правило его вывода.
* Начальный символ *S*.

В качестве примера рассмотрим простую грамматику, описывающую язык из *n* последовательных символов *а*, после которых идет столько же символов *b*. Математическая запись данной грамматики будет иметь вид:

Множества терминалов и нетерминалов для данной грамматики описаны следующим образом:

В грамматике предусмотрены два возможных порождающих правила:

1. *S* → *aSb*
2. *S* → *ab*

Допустим, на вход подается следующая строка: *aaabb*. Грамматика обрабатывает ее следующим образом:

1. Применяется первое порождающее правило:

1. К оставшемуся выражению повторно применяется первое правило:

1. Поскольку нет правила, которое бы позволяло *S* → *a,* разбор останавливается: строка *aaabb* не соответствует грамматике.

Теория формальных языков изначально существовала как ветвь лингвистики, помогающая систематизировать синтаксические закономерности естественных языков. В информатике она используется для описания языков программирования и формализованных подмножеств естественных языков, в логике и математическом анализе - синтаксиса аксиоматических систем. Следует особо отметить, что теория формальных языков изучает исключительно синтаксическую составляющую, никак не взаимодействуя с семантикой слов и символов рассматриваемого языка.

Для классификации формальных грамматик используется иерархия, предложенная в 1956 году американским ученым Ноамом Хомским, носящая его имя. Он предложил делить грамматики на следующие четыре типа:

1. Тип 0 – *неограниченные* грамматики. Являются самым общим классом и включают в себя все языки, которые можно распознать с помощью машины Тьюринга. Правила генерации имеют вид α → β, где α и β – любые строки из терминалов и нетерминалов грамматики. Неограниченные грамматики также называются *рекурсивно перечислимыми*. Из-за сложности описания языка на практике почти нигде не применяются.
2. Тип 1 – *контекстно-зависимые* грамматики, используются для описания языков, восприимчивых к контексту. Правила генерации имеют вид αAβ → αγβ, где A – нетерминал, а α, γ и β являются строками терминалов и нетерминалов. Данный класс примечателен тем, что описывает все языки, которые может обработать линейно-ограниченный автомат. Контекстно-зависимыми грамматиками обычно описывают естественные языки, однако некоторые эзотерические языки программирования также являются контекстно-зависимыми.
3. Тип 2 – *контекстно-свободные* грамматики. Являются теоретической базой для большинства существующих языков программирования. Правила генерации имеют вид A → γ, то есть в левой части каждого выражения может стоять только один нетерминал, а выражение может содержать сколько угодно терминалов и нетерминалов.
4. Тип 3 – *регулярные* грамматики. Наиболее простой тип грамматик, где все правила имеют вид A → a или A → aB.

На рисунке 2.2 представлена диаграмма включения классов грамматик друг в друга. Она показывает, что каждый класс является подмножеством предыдущего, однако пустая строка может входить только в типы 2 и 3.

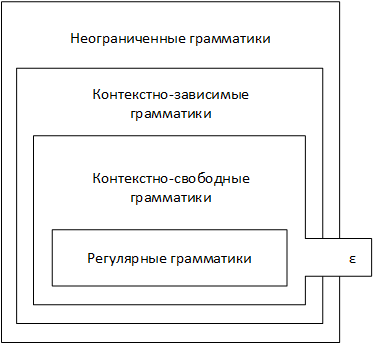


Рисунок 2.2: Классификация типов грамматик по Хомскому

**2.1.3 Классификация синтаксических анализаторов**

Синтаксические анализаторы делятся на три основных типа: универсальные, восходящие и нисходящие. Универсальные методы разбора, такие как алгоритм Эрли (Earley) или Кока-Янгера-Касами (Cocke-Younger-Kasami) могут работать с любой грамматикой, но на практике эти обобщенные методы оказываются слишком неэффективными для того, чтобы использовать их в промышленных компиляторах.

На практике обычно применяются методы *восходящего* (снизу вверх – *bottom-up*) или *нисходящего* (сверху вниз – *top down*) разбора. Как следует из названия, нисходящие синтаксические анализаторы строят дерево разбора от корня к листьям, тогда как восходящие трансляторы идут от листьев к корню. В обоих случаях, сканирование входного потока происходит посимвольно в направлении слева направо.

Наиболее эффективные нисходящие и восходящие методы работают только с подклассами грамматик, однако некоторые из них достаточно выразительны для описания всех синтаксических конструкций языков программирования. К ним относятся в первую очередь *LL-* и *LR-анализаторы*. Аббревиатуры этих анализаторов означают, что они сканируют текст слева направо (**L**eft to Right) и являются левоассоциативными (**L**eft) или правоассоциативными (**R**ight). Написанные вручную анализаторы обычно бывают типа LL, сгенерированные автоматически – LR.

LL-грамматики различаются между собой коэффициентом k, характеризующим максимальное количество символов, на которые анализатор должен заглядывать вперед для корректного распознавания грамматики. Грамматики класса LL(1) не требуют буфера и позволяют определять дальнейший путь построения дерева по текущему символу, поэтому являются наиболее простыми в реализации. Многие современные механизмы для построения синтаксических анализаторов не имеют ограничения на длину буфера и классифицируются как LL(\*).

Синтаксические анализаторы для языков класса LL(\*) используют механизм *отката* (англ. backtracking) в том случае, если сопоставление не прошло. В случае несоответствия входной строки описанию текущего применяемого правила, анализатор возвращается к родительскому правилу и если в нем содержится альтернатива, пытается применить ее. Алгоритм работает рекурсивно и считает входную строку некорректной только тогда, когда к ней были применены все возможные альтернативы, и ни одна из них не оказалась подходящей. Такой подход позволяет описывать куда более выразительные и лаконичные грамматики, нежели при ограниченном коэффициенте k – в этом случае требовалось бы использование ключевых слов или специальных последовательностей символов, по которым парсер может однозначно определить альтернативу, по которой дальше пойдет распознавание.

Помимо простоты реализации, LL и LR-парсеры хороши тем, что позволяют очень эффективно выявлять синтаксические ошибки. Ошибки выявляются на ранней стадии, когда разбор потока лексем становится невозможен в соответствии с существующей грамматикой. Это свойство называется *свойством корректного префикса* (англ. *viable prefix property*) и формально означает, что ошибкой является любой префикс, который не может быть префиксом ни одной корректной строки данного языка.

Основной проблемой LL-парсеров как класса является невозможность обнаружить и разрешить *левую рекурсию* в грамматике: при обнаружении ее сам парсер входит в бесконечную рекурсию распознавания, в результате чего переполняется стек вызовов. Рассмотрим пример грамматики, содержащей такую рекурсию:

Здесь левый символ тела продукции идентичен нетерминалу в заголовке этой же продукции. Предположим, что процедура A принимает решение о применении данной продукции. Так как тело начинается с A, рекурсивно вызывается та же самая процедура. Сканируемый символ изменяется только тогда, когда он соответствует терминалу в теле продукции, следовательно, между вызовами процедуры A не будет происходить никаких изменений, второй вызов отработает точно так же, как и первый – и так до бесконечности.

Левую рекурсию можно устранить, переписав некорректную продукцию. Приведенное выше правило можно разбить на два, превратив запрещенную для LL-парсеров левую рекурсию в разрешенную правую:

Нетерминал R *праворекурсивен*, поскольку содержит ссылку на себя в теле собственной продукции в качестве самого правого символа.

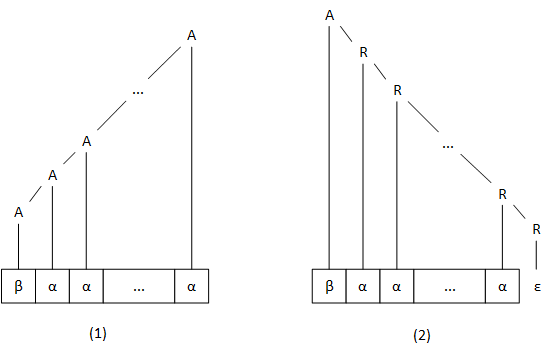


Рисунок 2.3: Схема построения дерева синтаксического разбора по леворекурсивному (1) и праворекурсивному (2) типам

На рисунке 2.3 представлено два вида того, как растут деревья разбора в зависимости от типа применяемой при записи грамматики рекурсии. Вариант 1 соответствует левой рекурсии, вариант 2 – правой.

**2.1.4 Форма Бэкуса-Наура**

Для построения синтаксического анализатора, использующего модель синтаксически управляемого перевода, сначала необходимо формально описать грамматику языка, программы на котором он будет обрабатывать. Одним из наиболее распространенных и широко применяемых в промышленных компиляторах методов для формализации грамматик типа 2 по иерархии Хомского служит *форма Бэкуса-Наура* (БНФ, англ *Backus-Naur Form*, также *Backus Normal Form*).

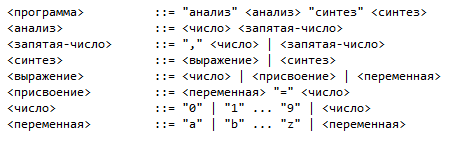
Впервые форма была применена американским программистом Джоном Бэкусом в 1959 году, когда он работал в IBM над языком ALGOL 58. Джону потребовался способ формальной записи синтаксиса разрабатываемого им языка, и он взял за основу классическую математическую нотацию грамматик 2 типа, заменив математические символы на те, которые легче набирать с клавиатуры. Четырьмя годами позже датский ученый Питер Наур описывает и расширяет ее в своем докладе, первоначально называя ее «Нормальной формой Бэкуса». На его доклад Дональд Кнут отвечает, что БНФ не является нормальной формой, и предлагает расшифровывать букву «Н» в аббревиатуре в честь фамилии Питера. В итоге, БНФ имеет следующие правила:

1. Грамматика языка описывается в виде множества строк вида:

<a> ::= <b> | <c> | … | <x>

1. Знак ::= заменяет знак стрелки (→), вертикальная черта означает альтернативу,
2. Терминалы записываются в кавычках, нетерминалы – в треугольных скобках,
3. Справа от знака ::= может стоять только один нетерминал,
4. Любой нетерминал, появившийся в правой части любого выражения, должен иметь описание в виде строки, где он стоит в левой части,
5. Для описания пустого терминала используется специальный символ.

Пример грамматики в записи БНФ приведен в листинге 2.1:



Листинг 2.1 – пример простой грамматики в виде БНФ-записи

Из записи видно, что БНФ не содержит синтаксиса для квантификаторов, т.е. не описывает возможное количество повторений той или иной группы терминалов и нетерминалов. Для описания повторения используется рекурсия. Для того, чтобы сделать запись более лаконичной и емкой, создатель языка Pascal Никлаус Вирт разработал усовершенствованный вариант БНФ, назвав его *расширенной формой Бэкуса-Наура* (РБНФ, англ. EBNF, *Extended Backus-Naur Form*). РБНФ имеет следующие отличия:

1. Для записи нетерминала не используются треугольные скобки,
2. Разделение левой и правой части производится символом «равно»,
3. Добавлены квантификаторы:
   1. Фигурные скобки означают 0 и более вхождений
   2. Квадратные скобки означают 0 или 1 вхождение

Следует заметить, что БНФ не может быть использован для описания абсолютно всех ограничений, накладываемых на синтаксис языка грамматикой. В языке могут существовать ограничения, которые невозможно выражать ни с помощью БНФ, ни с помощью РБНФ. Например, среди функциональных языков широко применяется использование отступов в качестве средства выделения блоков кода, но грамматика типа 2 не предусматривает возможности использовать, например, индекс глубины вложенности для определения размера необходимого отступа. В описании грамматики языка Python этот вопрос решен очень элегантным способом: его создатели применяют специальный *динамический нетерминал* IDENT, который нигде не объявлен в грамматике, но существует в парсере. Его значение увеличивается на 2 пробела при каждом вхождении анализатора в блок кода, и уменьшается при выходе из него. Таким образом обеспечивается точная проверка отступов, позволяющая вывести ошибку в том случае, если их размеры не согласованы.

Некоторые языковые конструкции можно выразить через БНФ, однако это потребует дублирования многих правил. Например, во многих императивных языках есть инструкция break, позволяющая немедленно выйти из цикла. Для того, чтобы позволить ее использование только внутри цикла, придется создать по два варианта всех правил, которые могут содержать блок кода (например, условие, блоки обработки исключений, управления освобождением и блокировкой ресурсов), так чтобы один вариант разрешал использование инструкции break и содержащих ее блоков, а другой – нет. Такой подход, будучи абсолютно верным с формальной точки зрения, на практике вносит дублирование избыточность, которые затрудняют реализацию синтаксического анализатора и модификацию грамматики. Гораздо более простым способом достичь такого же эффекта является наличие счетчика циклов: при входе в тело цикла он увеличивается на единицу, при выходе – уменьшается. В том случае, если парсер дошел до инструкции break, а счетчик циклов равен нулю, программа считается некорректно составленной.

Формализованная запись грамматики в виде БНФ применяется не только в математических выкладках и при сертификации языков программирования. Поскольку форма Бэкуса-Наура обладает достаточной гибкостью и выразительностью, чтобы описать синтаксис самой себя, существует достаточное количество автоматических средств, позволяющих сгенерировать «скелет» лексического и синтаксического анализаторов по передаваемой им на вход БНФ. Для получения работающего транслятора следует заполнить методы обработки каждого из нетерминалов смыслом, например – созданием узлов синтаксического дерева. Также БНФ может использоваться в качестве схемы при написании транслятора вручную.

**2.2 Описание алгоритма поиска и разрешения лексических замыканий**

Большинство языков программирования, в которых функции являются объектами первого рода (т.е. могут быть созданы в любом месте программы, сохранены в переменные и переданы в другие функции в качестве параметров) в том или ином виде поддерживают механизм *замыканий* (англ. closure). *Замыканием* называется метод или функция, объявленная внутри другой функции и имеющая ссылки на локальные переменные из родительской области видимости. В этом заключается основное отличие замыкания от указателя на функцию, который также может быть сохранен в переменную и передан в другую функцию или метод: каждая функция во время работы программы может иметь только один адрес, на который ссылаются все указатели, по которому в памяти размещена первая инструкция ее тела. Замыкание же является объектом, имеющим внутреннее состояние. Этот объект создается и инициализируется каждый раз при вызове родительской функции.

Существует два возможных подхода к реализации замыканий: замыкание *ссылки* и замыкание *значения*. Принципиальная разница между ними состоит в том, что объекты, замкнутые по ссылке, могут быть изменены в процессе работы функции-замыкания, в то время как объекты, замкнутые по значению, являются константами и из изменение не повлияет на изменение значений в родительской области видимости, либо вообще запрещено. Можно провести аналогию между этими типами замыканий и передачей аргументов в функцию по значению и по ссылке. К языкам, поддерживающим ссылочные замыкания, относятся, например, PHP / Javascript / C#, ко вторым – Java, Haskell.

Некоторые языки поддерживают механизм замыканий с помощью встроенных средств. Например, в стандарте языка ECMAScript описано, что для каждой функции создается таблица локальных переменных, которая также имеет ссылку на родительскую область видимость и ее таблицу, а та, в свою очередь, на свою родительскую – и так далее. Такой подход позволяет сильно упростить строение компилятора, поскольку ему не придется делать никакой работы по обнаружению замыканий, однако это снижает скорость обращения к переменным, поскольку при каждом обращении осуществляется поиск по целой цепочке пространств имен.

В статически типизированных языках обычно применяется подход, в котором замыкания представляются в виде *внутренних неявных классов* (англ. *implicit class*). Поскольку разрабатываемый язык ориентирован на платформу .NET, которая является статически типизированной и не имеет встроенных средств для работы с замыканиями, данный алгоритм будет рассмотрен более подробно и применен в разделе 3 при реализации компилятора.

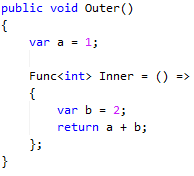
Суть алгоритма заключается в следующем: для каждой функции, в теле которой содержится объявление другой функции, создается специальный класс. Все вложенные объявления функций транслируются в методы этого класса, а локальные переменные, использованные в этих вложенных функциях – в поля. Экземпляр данного неявного класса создается каждый раз при входе в тело метода, содержащего вложенные объявления функций. Внутри этого метода обращение к замкнутым локальным переменным транслируется в обращение к полям данного объекта, а обращение к вложенным функциям – в обращение к его методам.

Алгоритм обнаружения замыканий является отдельным проходом по синтаксическому дереву, предшествующим фазе генерации кода. В результате этого прохода синтаксическое дерево остается неизменным, однако в таблице символов создаются записи для неявных типов и их полей и методов, соответствующих вложенным функциям и замкнутым переменным. Список локальных переменных также модифицируется, чтобы к моменту, когда компилятор начнет генерировать код, можно было точно определить, какая переменная локальна, а какая – замкнута.

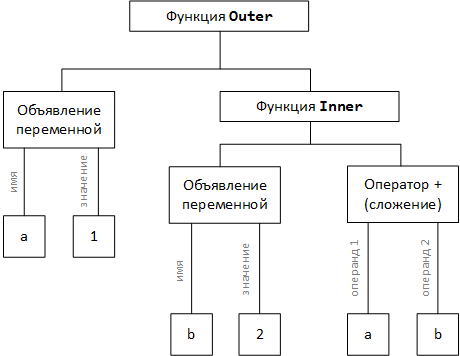
Алгоритм состоит из следующих шагов:

1. Для каждой транслируемой функции создается таблица переменных под названием *scope*, содержащая список локальных переменных и ссылку на аналогичную таблицу родительской области видимости. Для функции верхнего уровня ссылка не проставляется.
2. Переменная *currentScope*, хранящая ссылку на текущую таблицу локальных переменных, инициализируется указателем на scope функции верхнего уровня.
3. Синтаксическое дерево рекурсивно обходится в глубину.
4. При обнаружении вложенной функции текущее значение *currentScope* проставляется в качестве ссылки на родительскую область видимости для этой вложенной функции. После этого значение *currentScope* подменяется на ссылку на *scope* текущей функции, чтобы вложенные функции второго уровня могли быть рекурсивно обработаны тем же образом. После обхода тела вложенной функции восстанавливается оригинальное значение *currentScope*.
5. При обнаружении инструкции объявления переменной ее имя и тип заносятся в список локальных переменных. Флаг замкнутости *isClosured* по умолчанию имеет значение «не выставлен».
6. При обнаружении инструкции, ссылающейся на переменную (получение ее значения или присваивание ей нового), переменная ищется сначала в текущем scope, затем – в родительском, и так далее. Переменная *distance* принимает значение количества просмотренных областей видимости. Если переменная найдена и *distance* > 1, то флаг *isClosured* выставляется. Если переменная не найдена, выдается сообщение об ошибке.

На листинге 2.2 представлен пример функции, содержащей замыкание:



Листинг 2.2 – пример функции, содержащей вложенную функцию

  
Рисунок 2.4 – синтаксическое дерево функции из листинга 2.2

Локальная переменная *a* объявлена во внешней области видимости, а локальная переменная *b* – во внутренней. Действие данной программы довольно элементарно: внутренняя функция возвращает результат сложения двух переменных, одна из которых объявлена локально, а другая – в родительской области видимости.

В данном примере для большей наглядности показан только один уровень вложенности, однако алгоритм поддерживает неограниченное количество вложенных друг в друга функций. Если одно замыкание объявлено внутри другого замыкания, то наряду с полями для представления переменных объект внутреннего замыкания также имеет поле со ссылкой на объект внешнего замыкания. В таком случае, чтобы обратиться из внутреннего замыкания к переменной, объявленной в корневой области видимости, нужно обратиться сначала к ссылке на родительскую область видимости, а она уже будет содержать поле с этой переменной. Для большей глубины вложенности алгоритм применяется рекурсивно.

Работа алгоритма поиска лексических замыканий на данном синтаксическом дереве представлена на рисунке 2.5.

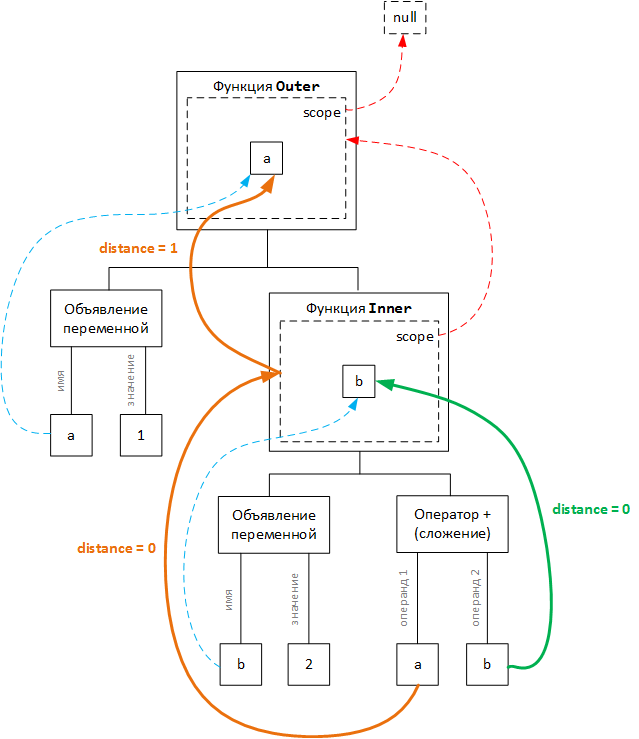


Рисунок 2.5 – схема работы алгоритма поиска замыканий для синтаксического дерева из листинга 2.2

На данном рисунке использованы следующие обозначения: синими пунктирными стрелками показаны объявления переменных в их локальных областях видимости, красными пунктирными – наследование областей видимости. Для области *Outer* родительской не существует.

Зеленая стрелка показывает поиск переменной *b*. Поскольку она находится в текущей области видимости, значение distance для нее равно нулю. Оранжевая стрелка показывает поиск переменной *a*. Изначально он был произведен в текущей области видимости, но там переменной с таким названием не существовало. Алгоритм поднялся на уровень выше, увеличив значение *distance* на единицу, и в родительской области видимости переменная *a* нашлась. Поскольку *distance* > 0, переменная a считается замкнутой и для обращения к ней как в теле функции *Inner*, так и в теле *Outer* будет генерироваться иной машинный код, нежели для обращения к переменной *b*.