**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.

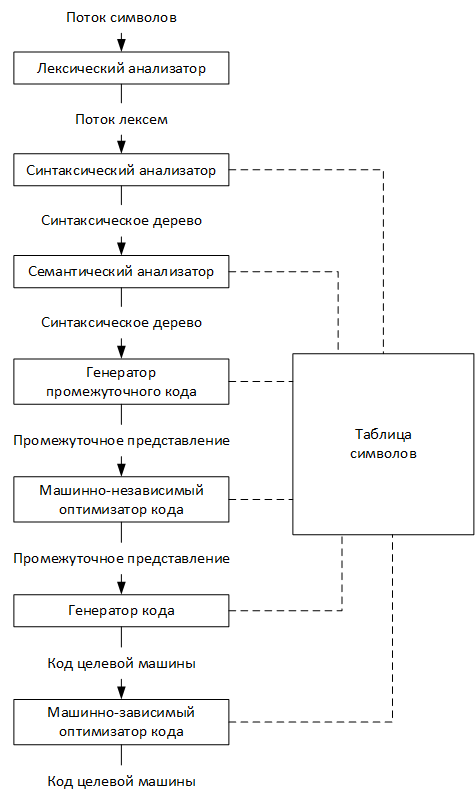


Рисунок 2.1: общая схема устройства компилятора

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.

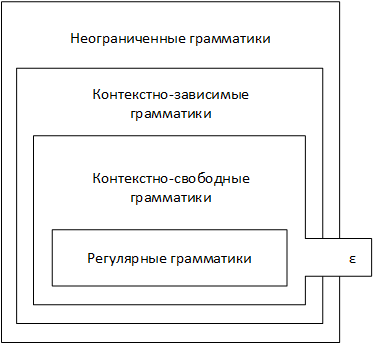


Рисунок 2.2: Классификация типов грамматик по Хомскому

**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.

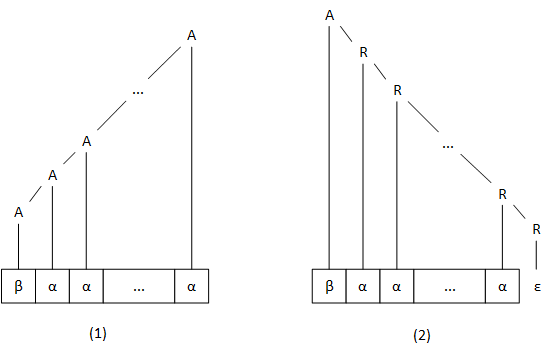


Рисунок 2.3: Схема построения дерева синтаксического разбора по леворекурсивному (1) и праворекурсивному (2) типам

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*).

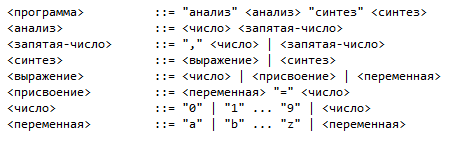
Впервые форма была применена американским программистом Джоном Бэкусом в 1959 году, когда он работал в IBM над языком ALGOL 58. Джону потребовался способ формальной записи синтаксиса разрабатываемого им языка, и он взял за основу классическую математическую нотацию грамматик 2 типа, заменив математические 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.

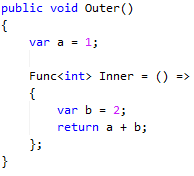
Суть алгоритма заключается в следующем: для каждой функции, в теле которой содержится объявление другой функции, создается специальный класс. Все вложенные объявления функций транслируются в методы этого класса, а локальные переменные, использованные в этих вложенных функциях – в поля. Экземпляр данного 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.

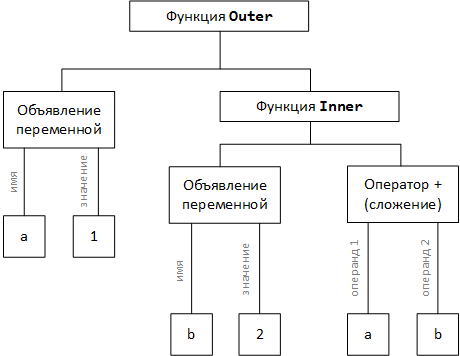
Алгоритм состоит из следующих шагов:

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.

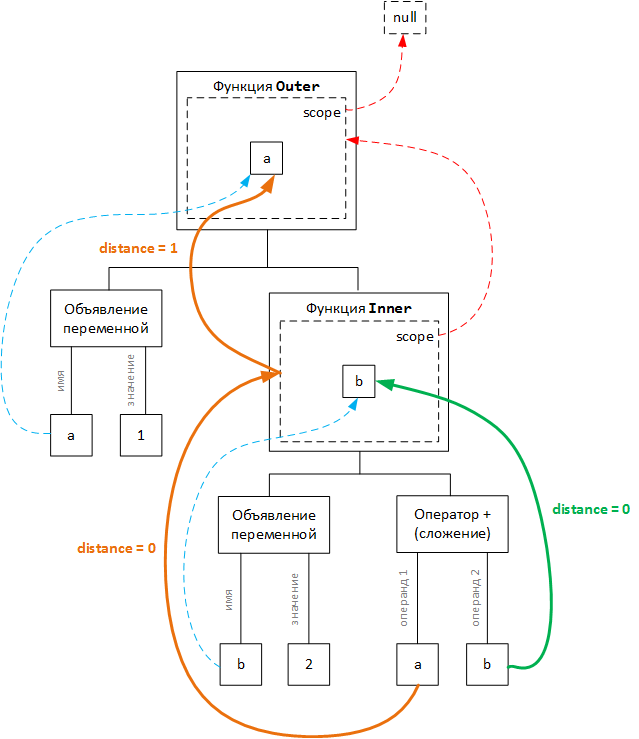


Рисунок 2.5 – схема работы алгоритма поиска замыканий для синтаксического дерева из листинга 2.2

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*.

**2.3 Описание алгоритма автовывода типов Хиндли-Милнера**

Одним из наиболее значимых преимуществ строгой статической типизации является возможность обнаружения целого класса ошибок, связанных с несоответствием типов, на этапе компиляции. Для этого компилятор должен иметь возможность определить тип любого выражения, написанного на исходном языке. Процесс определения типа выражения по типам других выражений и аннотациям называется *выводом типов*.

Существует несколько подходов к выводу типов. В более ранних языках со статической типизацией (таких, как C или Pascal), во многих местах требовались явные аннотации типов. Например, при объявлении переменной требовалось явно указывать ее тип, как показано в листинге 3.1:

**int** x = 10;  
**bool** value = **true**;  
EnhancedQueueStreamProcessor p = **new** EnhancedQueueStreamProcessor();

Листинг 3.1: пример избыточных аннотаций типов

В данном примере видно, что аннотация типа избыточна: число 10 является выражением типа «целое число», которое соответствует типу *int* в языке C. То же самое касается ключевого слова *true*, являющегося одним из двух возможных значений типа *bool*. Алгоритм вывода типов позволил бы определить тип переменной, исходя из типа значения, которым она инициализируется, и тогда код мог бы выглядеть так, как показано в листинге 3.2:

**var** x = 10;  
**var** value = **true**;  
**var** p = **new** EnhancedQueueStreamProcessor();

Листинг 3.2: автоматическое определение типов переменных

В языке C алгоритмы вывода типов не используются. Такой подход проще с точки зрения реализации компилятора, однако делает грамматику языка избыточной, менее гибкой и лаконичной.

В семействе функциональных языков, напротив, принято использовать минимальное количество явных аннотаций, необходимое для обеспечения контроля безопасности типов. Для этого применяется универсальный алгоритм, разработанный британскими учеными Роджером Хиндли и Робином Милнером независимо друг от друга и носящим их имя. Данный алгоритм также иногда называется «Алгоритм W» или «Алгоритм Дамаса-Милнера» в честь француза Луи Дамаса, доказавшего полноту алгоритма и добавившего в него поддержку полиморфических типов.

Алгоритм Хиндли-Милнера (далее ХM) считается быстрым, поскольку позволяет определить типы за время, линейно зависимое от размера исходного кода. Благодаря этому свойству он применяется во многих промышленных компиляторах таких функциональных языков, как Haskell, F# или Nemerle.

Следует заметить, что у ХM существуют и ограничения: он может быть использован не для любой системы типов. В объектно-ориентированных языках с функциональными элементами (вроде Scala) более применим *локальный* вывод типов. Данный алгоритм является модификацией ХМ, но работает только в рамках конкретного метода, выводя типы локальных переменных. Типы аргументов и возвращаемого значения должны быть указаны явно – это требование позволяет избежать случайно модификации интерфейса в результате изменения кода конкретного метода.

В алгоритме ХМ используются следующие обозначения и понятия:

* Типы *T*
* Выражения *e*
* Контексты типизации
* Отношения типизации

Под общим понятием «типы» подразумеваются базовые типы из некоторого зафиксированного множества B, типовые переменные, а также типы, полученные с помощью конструктора (обозначается символом →):

Базовые типы () представляют собой базовые типы, объявленные в языке, например int или bool. Теория λ-исчисления требует только наличие такого множества, но никак не регламентирует список входящих в него элементов. Типовые переменные () описывают типы, для которых конкретное значение еще не выведено. Конструктор типов позволяет описать функцию, переводящую выражение одного типа в выражение другого, и является правоассоциативным.

Суть алгоритма HM заключается в том, чтобы для всех типовых переменных v найти или вычислить конкретное значение. В том случае, если все выражения имеют определенный тип, случай является тривиальным, и наоборот – если в системе уравнений появляются лишние переменные, компилятор сообщает об ошибке и требует явной аннотации.

Выражения (обозначаются буквой e) – константы, абстракции и аппликации из классического типизированного λ-исчисления:

Здесь – константы, имеющие базовые типы. Например:

Под *контекстами типизации* понимается множество *предположений* о том, какие значения могут принимать типовые переменные. Существует также понятие *отношения типизации*, обозначающее подтвержденную гипотезу о типизации выражения относительно контекста. Оно записывается символом «».

Корректность суждения о типе выражения должна быть подтверждена одним из четырех определенных в ХМ *правил вывода типов.* Правила записываются в виде дробей, где в «числителе» - предпосылка для правила (*антецедент*), а в «знаменателе», соответственно, заключение (*консеквент*). Данные правила являются, по сути, следствиями аксиом типизированного λ-исчисления:

1. Константы заведомо типизированы корректно либо одним из базовых типов, либо одной из типовых переменных:
2. Если переменная *x* имеет тип *T* в некотором контексте **Г**, то отношение типизации записывается как *xT*, а *x* – корректно типизирована.
3. Если в контексте Г вместе с предположением *x:T* корректно типизировано выражение *e: σ*, в том же контексте без предположений о типе переменной x можно типизировать абстракцию :
4. Если в контексте Г выражение *e*1 корректно типизировано типом *T*→ *σ*, а выражение *e*2 корректно типизировано типом *T*, в этом же контексте можно корректно типизировать аппликацию (*e*1 *e*2):

Сам алгоритм ХМ состоит из следующих шагов:

1. На основании правил вывода типов порождается система предположений о типах для самого обрабатываемого λ-выражения и всех его подвыражений вплоть до переменных и констант. Особенностью алгоритма ХМ является тот факт, что правила вывода применяются без проверки предусловий.
2. Для предположений о типах всех подвыражений строятся равенства вида T1 = T2 на основании того, что типы T1 и T2 приписаны одному и тому же λ-терму. Весь набор таких равенств и представляет собой систему типовых уравнений.
3. Система типовых уравнений решается методом унификации, т.е. сопоставления типов между собой. Поскольку вариантов типовых сущностей может быть три (константа, переменная и применение), а операция унификации всегда применяется к двум из них, комбинаций может быть всего 32 = 9.

В качестве примера можно рассмотреть выражение применения .Алгоритм построит следующие предположения:

1. – применение правила 2 к переменной *x*
2. – применение того же правила к переменной *y*
3. – применение правила 4 к аппликативному выражению
4. – применение того же правила вывода к переменной *y*
5. – расширение типа *x* на основании применения правила 4

На основании предположений строится система уравнений, содержащая два элемента. Поскольку пример элементарный, система является вырожденной и уже содержит свои решения:

**Выводы**

Для реализации компилятора целесообразно использование метода синтаксически управляемой трансляции. Трансляция будет проходить в четыре прохода: сканирование, синтаксический анализ, семантический анализ и синтез. На втором шаге следует также использовать алгоритм поиска лексических замыканий. Алгоритм вывода типов Хиндли-Милнера же для лучшего взаимодействия с целевой платформой имеет смысл применять не для всей анализируемой исходной программы, а локально.