Министерство образования и науки РФ
Государственное образовательное учреждение
Высшего профессионального образования
Иркутский Государственный университет
Институт математики, экономики и информатики

Теория вычислительных процессов и структур

учебное пособие

2010

Предназначен для студентов старших курсов специальностей «прикладная математика и информатика» и «математическое обеспечение и администрирование информационных систем».

Составитель: Мезенцев А.В.

Оглавление

Часть 1. Синтаксис языков программирования.

Глава 1.Введение.

1.1. Компиляторы и интерпретаторы.

Транслятор – это программа, которая переводит исходную программу в эквивалентную ей объектную программу. Исходная программа пишется на некотором исходном языке, объектная программа формируется на объектном языке. Выполнение программы самого транслятора происходит во время трансляции.

Если исходный язык является языком высокого уровня, например таким, как Pascal или Basic, и если объектный язык – некоторый машинный язык, то транслятор называется компилятором. Машинный язык иногда называют кодом машины, поэтому и объектная программа иногда называется объектным кодом. Трансляция исходной программы в объектную происходит во время компиляции, а фактическое выполнение объектной программы происходит во время выполнения готовой программы.

Интерпретатор для некоторого исходного языка принимает исходную программу, написанную на этом языке, как входную информацию и выполняет ее. Различие между компилятором и интерпретатором состоит в том, что интерпретатор не порождает объектную программу, которая затем должна выполняться, а непосредственно выполняет ее сам..

Для того, чтобы выяснить, как осуществить выполнение инструкции исходной программы, чистый интерпретатор анализирует ее всякий раз, когда она должна быть выполнена. Конечно же, это не эффективно и используется не очень часто. При программировании интерпретатор обычно разделяют на две фазы. На первой фазе интерпретатор анализирует всю исходную программу, почти так же, как это делает компилятор, и транслирует ее в некоторое внутреннее представление. На второй фазе это

внутреннее представление исходной программы интерпретируется или выполняется. Внутреннее представление исходной программы разрабатывается для того, чтобы свести к минимуму время, необходимое для расшифровки или анализа каждой инструкции при ее выполнении.

1.2. Краткий обзор процесса компиляции.

Компилятор должен сначала выполнить анализ исходной программы, а затем синтез объектной программы. Сначала исходная программа разбивается на ее составные части; затем из них строятся части эквивалентной объектной программы. Для этого на этапе анализа компилятор строит несколько таблиц, которые затем используются как при анализе, так и при синтезе. На рис. 1.1 весь процесс показан более подробно.

Пунктирные стрелки изображают информационные потоки, сплошные стрелки указывают порядок работы программ.

Теперь опишем кратко различные части компилятора.

Информационные таблицы.

При анализе программы из описаний, заголовков процедур, заголовков циклов и т. д. извлекается информация и сохраняется для последующего использования. Эта информация обнаруживается в отдельных точках программы и организуется так, чтобы к ней можно было обратиться из любой части компилятора. Например, при каждом использовании идентификатора необходимо знать, как был описан этот идентификатор и как он использовался в

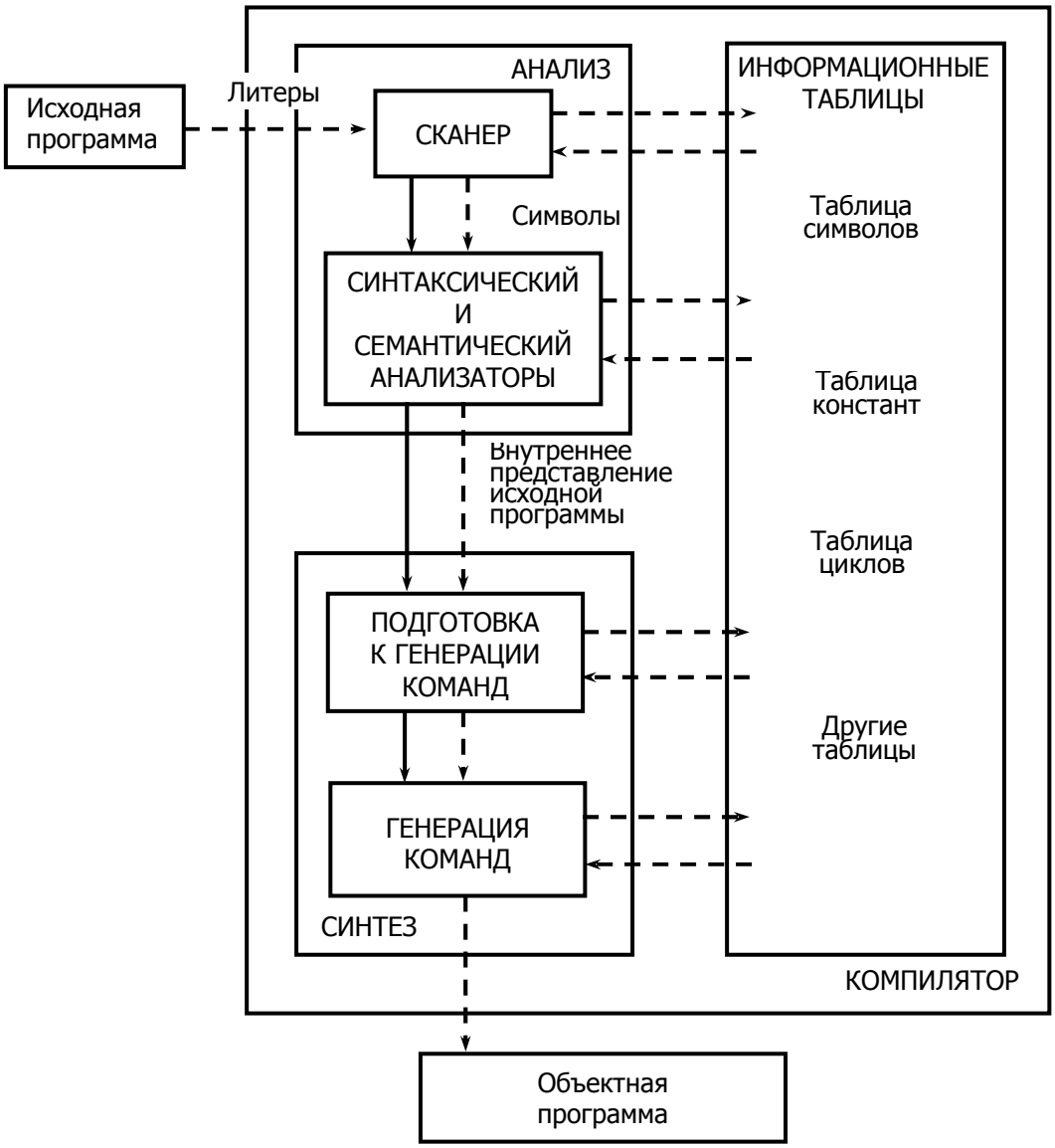


Рис. 1.1. Логические части компилятора.

других местах программы. Что конкретно следует хранить, зависит от исходного языка, объектного языка и сложности компилятора. Но в каждом компиляторе в той или иной форме используется таблица символов (другие названия – таблица идентификаторов, таблица имен). Эта таблица идентификаторов, встречающихся в исходной программе, вместе с их атрибутами. К атрибутам относятся тип идентификатора, его адрес в объектной программе и любая другая информация о нем, которая может понадобиться при генерации объектной программы.

Какую еще информацию следует собирать? Может потребоваться таблица констант, используемых в исходной программе. В эту таблицу будет включена сама константа и соответствующий ей адрес в объектной программе. Может понадобиться таблица заголовков for-циклов, отображающая структуру вложений циклов и хранящая переменные циклов. При разработке компилятора невозможно определить вид и содержание информации, которую следует собирать до тех пор, пока не будут достаточно обстоятельно продуманы команды объектной программы для каждой инструкции исходной программы и сама синтезирующая часть компилятора. Многое зависит от глубины задуманной оптимизации программы.

Сканер.

Сканер – самая простая часть компилятора, иногда также называемая лексическим анализатором. Сканер просматривает литеры исходной программы слева направо и строит символы (лексемы) программы – целые числа, идентификаторы, служебные слова, двухлитерные разделители, такие как >= и := и т. д. Лексемы передаются затем на обработку фактическому анализатору. На этой стадии могут быть исключены комментарии. Сканер также может заносить идентификаторы в таблицу символов и выполнять другую простую работу, которая фактически не требует анализа исходной программы.

Обычно сканер передает лексемы анализатору во внутренней форме. Каждый разделитель (служебное слово, знак операции или знак пунктуации) будет представлен целым числом. Идентификаторы или константы можно представить парой чисел. Первое число, отличное от любого целого числа, использующегося для представления разделителя, характеризует сам «идентификатор» или «константу»; второе число является адресом или индексом идентификатора или константы в

некоторой таблице. Это позволяет в остальных частях компилятора работать эффективно, оперируя с символами фиксированной длины, а не с цепочками литер переменной длины.

Синтаксический и семантический анализаторы.

Анализаторы выполняют действительно сложную работу по расчленению исходной программы на составные части, формированию ее внутреннего представления и занесению информации в таблицу символов и другие таблицы. При этом также выполняется полный синтаксический и семантический контроль программы.

Обычно анализатор представляет собой синтаксически управляемую программу. В действительности стремятся отделить синтаксис от семантики настолько, насколько это возможно. Когда синтаксический анализатор (распознаватель) узнает конструкцию исходного языка, он вызывает соответствующую семантическую процедуру (программу), которая проверяет данную конструкцию с точки зрения семантики и затем запоминает информацию о ней в таблице символов или во внутреннем представлении программы.

Внутреннее представление исходной программы.

Внутреннее представление исходной программы в значительной степени зависит от его дальнейшего использования. Это может быть дерево, отражающее синтаксис исходной программы. Это может быть исходная программа, в так называемой польской записи. Используется еще одна форма – список тетрад (оператор, операнд, операнд, результат) (или список триад (оператор, операнд, операнд)) в порядке их выполнения. Операндами будут не сами символические имена, а указатели на те элементы (или их индексы) в таблице символов, в которых описаны эти операнды.

Подготовка к генерации команд.

Перед генерацией команд обычно необходимо некоторым образом обработать и изменить внутреннюю программу. Кроме того, должна быть распределена память под переменные готовой программы. Одним из важных моментов на этом этапе является оптимизация программы с целью уменьшения времени ее работы.

Генерация команд.

По существу, на этом этапе происходит перевод внутреннего представления исходной программы на машинный язык. Это, по-видимому, наиболее хлопотная и кропотливая часть компилятора, хотя и наиболее понятная. В интерпретаторе эта часть компилятора заменяется программой, которая фактически выполняет (или интерпретирует) внутреннее представление исходной программы. Причем само внутреннее представление в этом случае мало чем отличается от того, которое получается при компиляции.

На рис. 1.1 показаны скорее логические связи между отдельными частями компилятора, чем последовательность их работы. Все четыре логически последовательных процесса: сканирование, анализ, подготовку к генерации и генерацию команд – можно выполнять в том порядке, который показан на рис. 1.1, или их можно выполнять параллельно, с определенной взаимной синхронизацией. Одним из критериев, определяющих выбор в данном случае, является объем доступной памяти. Часто выгодно или даже необходимо иметь несколько проходов (т. е. несколько раз вводить информацию в память компьютера). Другие критерии определяются целями, которые преследуются при разработке компилятора. Насколько быстрым должен быть сам компилятор? Насколько быстрой должна быть готовая программа? Какие средства отладки должны быть предусмотрены в готовой программе? Еще одним фактором является количество людей, занятых в разработке. Чем больше людей, тем больше, по-видимому, будет

проходов, чтобы каждый мог отвечать за отдельную и самостоятельную часть компилятора. Не все этапы должны быть обязательно осуществлены. В однопроходном компиляторе нет необходимости во внутреннем представлении программы, в то время как этапы подготовки и генерации команд растворяются в семантических программах. На рис. 1.2 представлена типичная однопроходная схема.

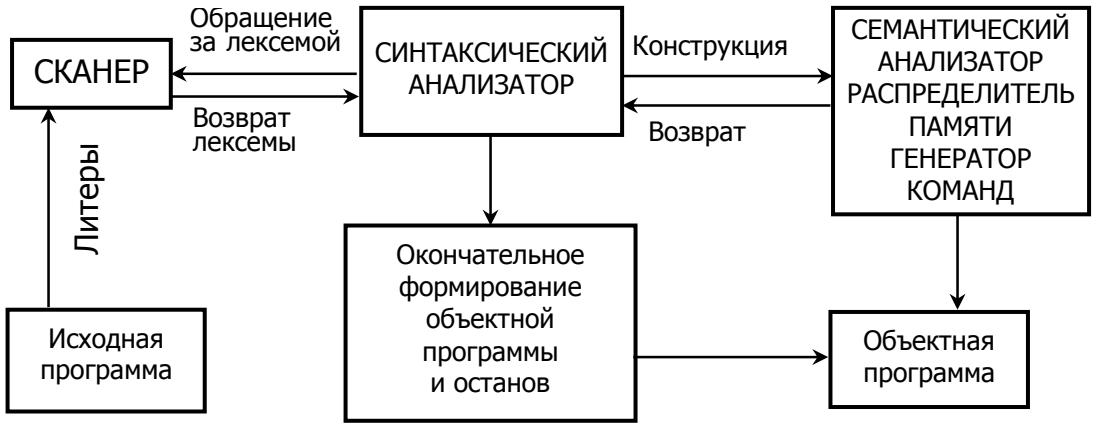


Рис 1.2. Однопроходный компилятор.

Синтаксический анализатор вызывает сканер, когда необходима новая лексема, и вызывает процедуру, когда конструкция распознана. Эта процедура осуществляет семантическую проверку, распределение памяти и генерирует команды, перед тем как возвратиться к разбору. Однако не все языки имеют такую структуру, которая допускает однопроходную трансляцию.

Глава 2. Грамматики и языки.

2.1. Символы и цепочки.

Язык неформально – подмножество множества всех предложений из слов (символов) некоторого словаря (алфавита).

Алфавит – это непустое конечное множество элементов. Элементы алфавита назовем ***символами***. ***Цепочка*** – всякая конечная последовательность символов алфавита. Существует также ***пустая цепочка***, то есть цепочка не содержащая ни одного символа (e). Порядок символов в цепочке важен, так $ab \neq ba$. ***Длина цепочки*** x (записывается как $|x|$) – число символов в цепочке.

$|e|=0$ $|a|=1$ $|abb|=3$

Если x и y цепочки, то цепочка xy называется ***конкатенацией*** (сцеплением) цепочек x и y. Например $x=ab$, $y=cd$ $xy=abcd$.

Для любой цепочки x всегда $xe=ex=x$.

Пусть x, y, z произвольные цепочки в некотором алфавите A. Будем называть x ***префиксом*** (головой) цепочки xy, а y – ***суффиксом*** (хвостом) цепочки xy.

Множество цепочек в алфавите будем обозначать заглавными буквами A, B,…

Произведение AB двух множеств цепочек A и B – $AB=\{xy | x \in A \ \& \ y \in B\}$.

$\{e\}A=A\{e\}=A$ для любого множества цепочек A.

Если x – цепочка, то x^0 – пустая цепочка, $x^1=x$, $x^2=xx$,…, $x^n=xx^{n-1}=x^{n-1}x$ для $n>0$.

Если A – множество цепочек, то $A^0=\{e\}$, $A^1=A$, …, $A^n=AA^{n-1}=A^{n-1}A$ для $n>0$.

Определим ***итерацию*** A^* и ***усеченную*** (позитивную) итерацию A^+ множества цепочек A. $A^+ = A^1 \cup A^2 \cup ... \cup A^n \cup ... = \bigcup_{n\geq 1} A^n$. $A^* = A^0 \cup A^+ = \bigcup_{n\geq 0} A^n$.

Заметим, что $A^+ = AA^* = A^*A$.

2.2. Формальное определение грамматики и языка.

Существуют два основных механизма определения языка – ***порождения*** и ***распознования***. Механизм порождения – это грамматики Хомского.

Продукцией или правилом подстановки называется упорядоченная пара (U, x), которая может записываться как U::=x или $U\rightarrow x$.

Дадим формальное определение грамматики. Грамматикой называется четверка G=(V_N, V_T, P, S), где

(1) V_N – конечное множество нетерминальных символов (нетерминалов)

(2) V_T – конечное множество терминальных символов

(3) P – конечное подмножество множества $(V_N \cup V_T)^* V_N (V_N \cup V_T)^* \times (V_N \cup V_T)^*$ элемент $(\alpha, \beta) \in P$ называется правилом подстановки (продукцией)

(4) $S \in V_N$ выделенный символ, называемый начальным (исходным).

Пример. $G_1 = (\{A, S\}, \{0,1\}, P, S)$, где $P = \{(S, 0A1), (0A, 00A1), (A, e)\}$.

Пусть G грамматика. Будем говорить, что цепочка v ***непосредственно порождает*** цепочку w (w ***непосредственно выводима*** из v) и обозначать это $v\Rightarrow w$, если для некоторых цепочек x и y можно записать $v=xzy$, $w=xuy$, где $(z,u)\in P$.

Будем говорить, что v ***порождает*** w (w ***выводима*** из v) $v\Rightarrow+w$, если $v=u_0\Rightarrow u_1\Rightarrow\ldots\Rightarrow u_n=w$, где $n>0$. Последовательность непосредственных выводов называется выводом длины n.

Будем записывать $v \Rightarrow^* w$, если $v \Rightarrow^+ w$ или $v = w$. Пока в цепочке есть хотя бы один нетерминал, из нее можно вывести новую цепочку.

Пусть $G=(V_N, V_T, P, S)$ – грамматика. Цепочка x называется ***сентенциальной формой***, если x выводима из начального символа S ($S \Rightarrow^* x$). ***Предложение*** – это сентенциальная форма, состоящая из одних терминальных символов ($S \Rightarrow^* x$ $x \in V_T^*$). Язык $L(G)$ – это множество предложений ($L(G)=\{x \mid S \Rightarrow^* x$ & $x \in V_T^*\}$).

Заметим, что несколько различных грамматик могут порождать один и тот же язык.

Пусть $G=(V_N, V_T, P, S)$ – грамматика. И пусть $w=xuy$ – сентенциальная форма (т.е. $S \Rightarrow^* xuy$). Тогда u называется ***фразой сентенциальной формы w для нетерминала U***, если $S \Rightarrow^* xUy$ и $U \Rightarrow^+ u$. u называется ***простой фразой***, если $S \Rightarrow^* xUy$ и $U \Rightarrow u$.

Основой всякой сентенциальной формы называется самая левая простая фраза.

Грамматика описывает бесконечный язык, то есть язык состоящий из бесконечного числа предложений, когда хотя бы одно из ее правил содержит какой-либо нетерминал и в левой, и в правой частях. В общем случае, если $U \Rightarrow^+ \ldots U \ldots$, мы будем говорить, что ***грамматика рекурсивна*** по отношению к U. Если $U \Rightarrow^+ U \ldots$, то имеет место левая рекурсия, если $U \Rightarrow^+ \ldots U$, то имеет место правая рекурсия. Правило называется лево (право) рекурсивным, если оно имеет вид $U ::= U \ldots$ ($U ::= \ldots U$). Если язык бесконечный, то определяющая его грамматика должна быть рекурсивной.

2.3.Синтаксические деревья и неоднозначность.

Синтаксические деревья помогают понять синтаксис предложения.

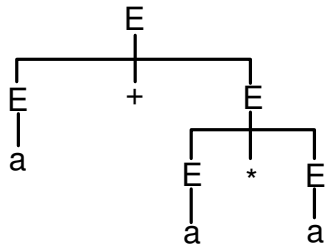
В качестве примера рассмотрим дерево для вывода предложения $a+a*a$ грамматики $G_0=(\{E\}, \{a, +, *, (,)\}, P, E)$, где P состоит из правил:

$E \to E + E$

$E \to E * E$

$E \to (E)$

$E \to a$



$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow a + E * E \Rightarrow a + a * E \Rightarrow a + a * a$

Куст узла – это множество подчиненных ему узлов. Символы узла образуют цепочку, которая заменяет имя куста в непосредственном выводе. Концевые (висячие) узлы синтаксического дерева – это узлы, не имеющие подчиненного куста.

Поддерево состоит из узла дерева (называемого корнем поддерева) вместе с той частью дерева (если она имеется), которая исходит из него. Поддеревья тесно связаны с фразами; концевые узлы образуют фразу для корня данного поддерева.

Можно восстановить вывод по дереву:

$E + a * a \Rightarrow a + a * a$

$E + E * a \Rightarrow E + a * a$

$E + E * E \Rightarrow E + E * a$

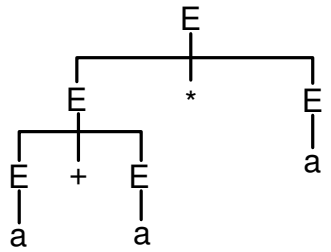
$E + E \Rightarrow E + E * E$

$E \Rightarrow E + E$

Для каждого синтаксического дерева существует, по крайней мере один вывод. Для каждого вывода есть соответствующее синтаксическое дерево (но несколько разных выводов могут иметь одно и тоже дерево).

Заметим, что выводы отличаются лишь порядком применения правил и что синтаксическое дерево не определяет точный порядок.

Предложение грамматики называется ***неоднозначным***, если для его вывода существует два синтаксических дерева. Грамматика ***неоднозначна***, если она допускает неоднозначные предложения. Грамматика G_0 неоднозначна. Действительно, мы можем привести для вывода рассмотренного выше предложения $a + a * a$ еще одно синтаксическое дерево:

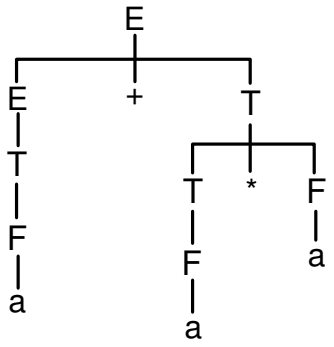


Если сентенциальная форма неоднозначна, то она имеет более чем одно синтаксическое дерево и поэтому в общем случае более чем одну основу. Изменяя неоднозначную грамматику, но не изменяя ее предложений, можно иногда получить однозначную грамматику для того же самого множества предложений. Модифицируем нашу грамматику G_0 в грамматику G_1:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid a$



Заметим, что мы называем неоднозначной грамматику, а не сам язык. Изменяя неоднозначную грамматику, но не изменяя ее предложения, можно иногда получить однозначную грамматику для того же самого множества

предложений. Однако есть языки, для которых не существует однозначной грамматики. Такие языки называются ***существенно неоднозначными***. Было доказано, что проблема распознавания неоднозначности алгоритмически неразрешима.

2.4. Задача разбора.

Разбор сентенциальной формы означает построение вывода и возможно синтаксического дерева.

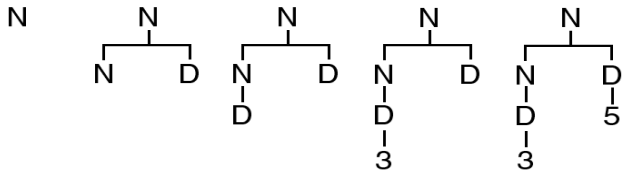
Различают две категории алгоритмов разбора: ***нисходящий*** (сверху вниз) и ***восходящий*** (снизу вверх), что соответствует способу построения синтаксических деревьев.

Рассмотрим обе категории на примере следующей грамматики:

$N \rightarrow D \mid N\ D$

$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

При нисходящем разборе дерево строится от корня (начального символа). На каждом шаге нисходящего разбора самый левый нетерминал текущей сентенциальной формы заменяется на правую часть правила, в результате получается следующая сентенциальная форма. Проблема заключается в том, что надо получить ту сентенциальную форму, которая совпадает с заданной цепочкой.



$N \Rightarrow N\ D \Rightarrow D\ D \Rightarrow 3\ D \Rightarrow 3\ 5$

При восходящем разборе дерево строится от листьев (концевых узлов) к корню. То есть, отправляясь от заданной цепочки, пытаются привести ее к начальному символу. На каждом шаге восходящего разбора основа текущей сентенциальной формы приводится (редуцируется) к нетерминалу (левой части правила, в котором основа является правой частью).

$N \Rightarrow N\ D \quad \Rightarrow N\ 5 \quad \Rightarrow D\ 5 \quad \Rightarrow 3\ 5$

Мы расположили деревья на рисунке справа налево, потому что такое расположение лучше иллюстрирует построенный вывод, который теперь читается, как обычно, слева направо. Заметим, что выводы, построенные при разных разборах, отличаются, но имеют одно и то же синтаксическое дерево.

Рассмотрим теперь главные проблемы разбора.

1. Предположим, что при нисходящем разборе надо заменить самый левый нетерминал V, и пусть имеется n правил:

   $V ::= x_1 \mid x_2 \mid \ldots \mid x_n$.

   Как узнать, какой цепочкой x_i следует заменить V?
2. При восходящем разборе на каждом шаге редуцируется основа. Как найти основу и то, к чему она должна приводиться?

На эти вопросы не всегда легко ответить. Одним из решений будет выбор одного из возможных вариантов наугад в предположении, что он верен. Если позднее выяснится, что он не верен, то мы должны будем вернуться назад и попытаться применить другой вариант. Такой разбор называется разбором с возвратами. Очевидно, что он будет не очень эффективным. Другим решением будет просмотр контекста вокруг обрабатываемой в данный момент подцепочки.

2.5.Классификация грамматик по Хомскому.

Хомский в 1956 году определил четыре основных класса языков в терминах грамматик $G=(V_N,V_T,P,S)$. Различие заключается в форме правил подстановки, допустимых в P.

(1) Говорят, что G – это грамматика типа 0 или грамматика ***с фразовой структурой***, если правила имеют вид:

$u \to v$, где $u \in (V_N \cup V_T)^+$, $v \in (V_N \cup V_T)^*$

(2) Грамматика типа 1 или ***контекстно-чувствительная*** (контекстно-зависимая) грамматика, если правила имеют вид:

$xUy \to xuy$, где $U \in V_N$, $u \in (V_N \cup V_T)^+$, $x, y \in (V_N \cup V_T)^*$

(3) Грамматика типа 2 или ***контекстно-свободная*** грамматика (КС-грамматика):

$U \to u$, где $U \in V_N$, $u \in (V_N \cup V_T)^*$

Может появиться правило вида $U \to e$, однако известно что по заданной КС-грамматике G_0 всегда можно построить неукорачивающую КС-грамматику G_1, такую что $L(G_1) = L(G_0) - \{e\}$

(4) Грамматика типа 3 или регулярная (***автоматная***, А-грамматика):

$U \to N$ или $U \to WN$, где $N \in V_T$, $U, W \in V_N$

Вводя все большие ограничения, мы определили четыре класса грамматик. Таким образом, есть языки с фразовой структурой, которые не являются контекстно-чувствительными, контекстно-чувствительные языки, которые не являются контекстно-свободными, и контекстно-свободные, которые не являются регулярными. В большинстве работ по теории формальных языков изучаются контекстно-свободные или регулярные языки, или их подмножества, так как именно этими языками являются большинство языков программирования.

2.6. Некоторые отношения применительно к грамматикам.

С грамматиками связано несколько важных символьных множеств. Эти множества легко определяются в терминах совсем простых отношений и их транзитивных замыканий.

Если задана грамматика и нетерминальный символ U, нам может потребоваться знать множество головных символов в цепочках, выводимых

из U. Назовем это множество $HEAD(U)$ и определим его следующим образом:

$HEAD(U) = \{S \mid U \Rightarrow^{+} S\ldots\}$

В большинстве случаев такое определение могло бы быть удовлетворительным. Заметим, однако, что в определении используется отношение $\Rightarrow+$, заданное на бесконечном множестве цепочек, и хотя само множество $HEAD(U)$ конечно, при его построении могут возникнуть трудности. Поэтому мы переопределим множество $HEAD(U)$, используя другое отношение, заданное на конечном множестве правил вывода.

Определим отношение $FIRST$ в конечном объединенном алфавите $V_N \cup V_T$ грамматики следующим образом:

$U\ FIRST\ S$ тогда и только тогда, когда существует правило $U ::= S\ldots$.

$U\ FIRST^{+}\ S$ тогда и только тогда, когда существует непустая последовательность правил $U ::= S_1\ldots$, $S_1 ::= S_2\ldots$, …, $S_n ::= S\ldots$.

Из данного определения, очевидно, вытекает, что

$U\ FIRST^{+}\ S$ тогда и только тогда, когда $U \Rightarrow^{+} S\ldots$.

Заметим, что если отношение $FIRST^{+}$ рассматривать как множество, то оно полностью определяет множество $HEAD(U)$: $HEAD(U) = \{S \mid (U, S) \in FIRST^{+}\}$.

Второе множество – это множество символов, которыми оканчиваются цепочки, выводимые из некоторого символа U. Оно определяется для всех символов U через отношение $LAST^{+}$, являющегося транзитивным замыканием отношения $LAST$.

$U\ LAST\ S$ тогда и только тогда, когда существует правило $U ::= \ldots S$.

Глава 3. Лексический анализ.

3.1. Сканер.

Сканер представляет собой ту часть компилятора, которая читает литеры первоначальной исходной программы и строит символы (лексемы) исходной программы (идентификаторы, служебные слова, целые числа, одно- или двухлитерные разделители, такие, как *, +, **, /*). Сканер выполняет простой лексический анализ исходной программы в отличие от синтаксического анализа, и поэтому сканер называют также лексическим анализатором.

Почему лексический анализ нельзя объединить с синтаксическим анализом? Есть несколько серьезных доводов в пользу отделения лексического анализа от синтаксического.

1. Значительная часть времени компиляции тратится на сканирование литер. Выделение же позволяет сконцентрировать внимание на сокращении этого времени.
2. Синтаксис лексем можно описать в рамках очень простых грамматик. Можно разработать эффективную технику разбора, наилучшим образом учитывающую особенности этих грамматик.
3. Так как сканер выдает лексемы вместо литер, синтаксический анализ на каждом шаге получает больше информации том, что надо делать. Более того, некоторые специфические проверки контекста, необходимые при разборе лексем, проще и уместнее делать в сканере, чем в формальном синтаксическом анализаторе.

Сканер можно запрограммировать как отдельный проход, на котором выполняется полный лексический анализ исходной программы и который выдает синтаксическому анализатору таблицу, содержащую исходную программу во внутреннем представлении. В другом варианте это может быть подпрограмма, к которой обращается синтаксический анализатор всякий раз, когда ему необходима новая лексема. В ответ на каждый вызов подпрограмма распознает следующую лексему исходной программы и отсылает ее синтаксическому анализатору. Второй вариант, вообще говоря,

лучше, так как нет необходимости строить целиком всю внутреннюю исходную программу и хранить ее в таком виде в памяти.

Большинство лексем в языках программирования попадают в один из следующих классов:

- идентификаторы;
- служебные слова (которые являются подмножеством идентификаторов);
- целые числа;
- однолитерные разделители (+, -, (,), / и т.д.);
- двулитерные разделители (//, /*, **, := и т.д.).

Эти лексемы могут быть описаны следующими простыми правилами:

<идентификатор> ::= БУКВА | <идентификатор> БУКВА

| <идентификатор> ЦИФРА

<целое> ::= ЦИФРА | <целое> ЦИФРА

<разделитель> ::= + | - | (|) | / | …

<разделитель> ::= <слэш> / | <слэш> * | <звездочка> * | <двоеточие> =

<слэш> ::= /

<звездочка> ::= *

<двоеточие> ::= :

Конечно, правила могли бы быть еще проще, но мы записали их в таком виде, чтобы грамматика была автоматной. Синтаксис лексем большинства языков программирования можно задать в такой форме. Следовательно, целесообразно найти эффективный способ разбора предложений автоматной грамматики.

3.2. Конечные автоматы.

В этом параграфе мы рассмотрим некоторые результаты, которые не будем формально доказывать. Нас главным образом будет интересовать вопрос, как программировать сканеры, а для этого доказательства не потребуются.

Рассмотрим автоматную грамматику:

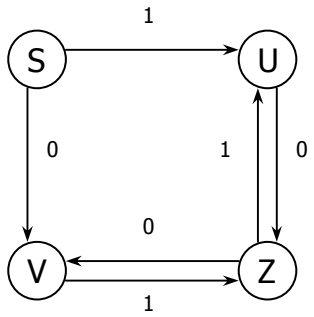
Z ::= U0 | V1

U ::= Z1 | 1

V ::= Z0 | 0

Легко видеть, что порождаемый ею язык состоит из последовательностей, образуемых парами 01 или 10, т.е. $L(G) = \{B^n \mid n>0\}$, где $B = \{01, 10\}$.

Чтобы облегчить распознавание предложений грамматики *G*, нарисуем диаграмму состояний:



В этой диаграмме каждый нетерминал грамматики *G* представлен узлом или состоянием; кроме того, есть начальное состояние *S* (предполагается, что грамматика не содержит нетерминала *S*). Каждому правилу *Q* ::= *T* в *G* соответствует дуга с пометкой *T*, направленная от начального состояния *S* к состоянию *Q*. Каждому правилу *Q* ::= *RT* соответствует дуга с пометкой *T*, направленная от состояния *R* к состоянию *Q*.

Мы используем диаграмму состояний, чтобы распознавать или разобрать цепочку x следующим образом:

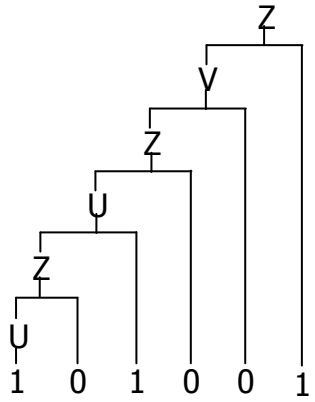
1. Первым текущим состоянием считать начальное состояние *S*. Начать с самой левой литеры в цепочке *x* и повторять шаг 2 до тех пор, пока не будет достигнут правый конец *x*.
2. Считать следующую литеру *x*, продвинуться по дуге, помеченной этой литерой, переходя к следующему текущему состоянию.

Если при каком-то повторении шага 2 такой дуги не оказывается, то цепочка *x* не является предложением и происходит останов. Если мы достигаем

конца *x*, то *x* – предложение тогда и только тогда, когда последнее текущее состояние есть *Z*.

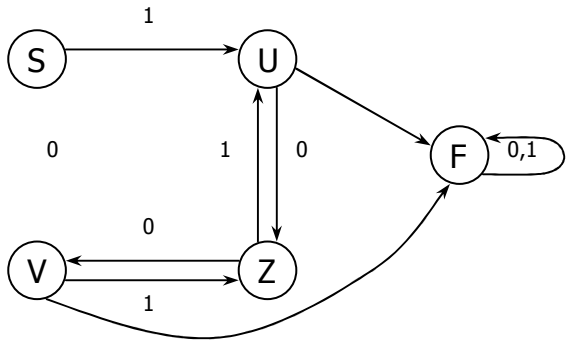
В этих действиях легко узнать восходящий разбор. На каждом шаге (кроме первого) основой является имя текущего состояния, за которым следует входной символ. Символ, к которому приводится основа, будет именем следующего состояния. В качестве примера проведем разбор предложения 101001.

Шаг	*Текущее состояние*	*Остаток цепочки x*
1	*S*	101001
2	*U*	01001
3	*Z*	1001
4	*U*	001
5	*Z*	01
6	*V*	1
7	*Z*	*e*



В этом примере разбор выглядит таким простым благодаря простому виду правил. Так как нетерминалы встречаются лишь как первые символы правой части, на первом шаге первый символ предложения всегда приводит к нетерминалу. На каждом последующем шаге первые два символа *UT* сентенциальной формы *UTt* приводятся нетерминалу *V*, при этом используется правило *V* ::= *UT*. При выполнении этой редукции имя текущего состояния – *U*, а имя следующего состояния – *V*. Так как каждая правая часть единственна, то единственным оказывается и символ, к которому она приводится.

Чтобы избавиться от проверки на каждом шаге, есть ли дуга с соответствующей пометкой, можно добавить еще одно состояние, называемое *F* (неудача), и добавить все необходимые дуги от всех состояний к *F*. Добавляется также дуга, помеченная всеми возможными литерами и ведущая из *F* обратно в *F*.



Конечный автомат является одним из простейших распознавателей. Бесконечной памяти у него нет. Обычно конечный автомат состоит только из входной ленты и управляющего устройства с конечной памятью. Входная головка является односторонней, т.е. входная лента может двигаться только справа налево.

Дадим теперь формальное определение конечного автомата.

Детерминированный конечный автомат – это пятерка $M = (Q, V_T, \delta, q_0, F)$, где

(1) Q – конечное множество состояний;

(2) V_T – конечное множество допустимых входных символов;

(3) δ - отображение множества $Q \times V_T$ в множество Q (функция переходов);

(4) $q_0 \in Q$ – начальное состояние;

(5) $F \subseteq Q$ – множество заключительных состояний.

Работа конечного автомата представляет собой некоторую последовательность шагов, или тактов. ***Такт*** определяется текущим состоянием управляющего устройства и входным символом, обозреваемым в данный момент входной головкой. Сам шаг состоит из изменения состояния и сдвига входной головки на одну ячейку вправо.

Для того чтобы определить будущее поведение конечного автомата, надо знать лишь

(1) текущее состояние управляющего устройства и

(2) цепочку символов на входной ленте, состоящую из символа под головкой и всех символов, расположенных вправо от него.

Пару $(q, u) \in Q \times V_T^*$ будем называть ***конфигурацией*** конечного автомата M.

Конфигурация (q_0, w), где w входная цепочка, называется ***начальной***.

Конфигурация (q,e), где $q \in F$, называется ***допускающей***.

Такт автомата M представляется бинарным отношением $\vdash_M$ (или $\vdash$, если M подразумевается), определенным на конфигурациях. Если $\delta(q,a)$ содержит q', то $(q,aw) \vdash (q',w)$ для всех $w \in V_T^*$.

Запись $C \vdash^0 C'$ означает, что $C = C'$, а $C_0 \vdash^k C_k$ (для $k \geq 1$) – что существуют такие конфигурации $C_1, C_2, \ldots, C_{k-1}$, что $C_i \vdash C_{i+1}$ для всех $0 \leq i < k$. $C \vdash^* C'$ означает, что $C \vdash^k C'$ для некоторого $k \geq 0$.

Говорят, что автомат M ***допускает*** цепочку w, если $(q_0,w) \vdash^* (q,e)$ для некоторого $q \in F$.

Языком, определяемым (распознаваемым, допускаемым) автоматом M, называется множество входных цепочек, допускаемых автоматом M, т.е.

$L(M) = \{w \mid w \in V_T$ и $(q_0,w) \vdash^* (q,e)$ для некоторого $q \in F\}$

Пример. Пусть $M = (\{p, q, r\}, \{0, 1\}, \delta, p, \{r\})$ – конечный автомат, где δ задается следующей таблицей:

δ		Входные символы	
		0	1
Состояния	p	q	p
	q	r	p
	r	r	r

M допускает все цепочки из 0 и 1, содержащие два рядом стоящих 0.

Рассмотрим, как работает автомат с входной цепочкой 01001:

$(p,01001) \vdash (q,1001) \vdash (p,001) \vdash (q,01) \vdash (r,1) \vdash (r,e)$, т.е. $01001 \in L(M)$.

3.3. Недетерминированный конечный автомат.

В предыдущем параграфе, при построении конечного автомата по грамматике, все у нас получилось хорошо, благодаря тому, что наша грамматика не содержала разных правил с одинаковыми правыми частями. Проблема возникает, когда грамматика G содержит два правила $V \to UT$ и $W \to UT$. Это значит, что в диаграмме состояний есть две дуги, помеченные T и исходящие из узла U, т.е. отображение δ оказывается неоднозначным. Таким образом, мы приходим к понятию недетерминированного конечного автомата.

Недетерминированный конечный автомат – это пятерка

$M=(Q, V_T, \delta, q_0, F)$, где

(1) Q – конечное множество состояний;

(2) V_T – конечное множество допустимых входных символов;

(3) δ - отображение множества $Q \times T$ в множество всех подмножеств множества Q - функция переходов;

(4) $q_0 \in Q$ – начальное состояние;

(5) $F \subseteq Q$ – множество заключительных состояний.

Пример. Пусть $G(Z)=(\{Z, U, V, Q\}, \{0, 1\}, P, Z)$ – грамматика, где

$P = \{Z \to U1|V0|Z0|Z1$

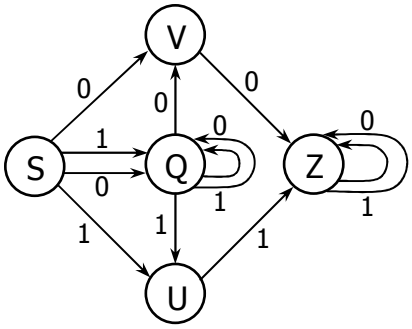
$U \to Q1|1$

$V \to Q0|0$

$Q \to Q0|Q1|0|1\ \}$

Краткое рассмотрение показывает, что язык $L(G)$ представляет собой множество цепочек из 0 и 1, содержащих, по крайней мере, два смежных 0 или две смежные 1.

Построим по заданной грамматике конечный автомат в виде диаграммы состояний:



и функцию переходов в виде таблицы:

δ		*Входные символы*	
		0	1
Состояния	S	$\{Q, V\}$	$\{Q, U\}$
	V	$\{Z\}$	$\varnothing$
	Q	$\{Q, V\}$	$\{Q, U\}$
	Z	$\{Z\}$	$\{Z\}$
	U	$\varnothing$	$\{Z\}$

Проблема состоит в том, что теперь на каждом шаге может быть более одной дуги, помеченной следующей входной литерой, так что мы не знаем, какой путь выбрать. На каждом шаге известна основа текущей сентенциальной формы, но не известно, к чему она приводится.

Рассмотрим работу нашего автомата на примере входной цепочки 10110:

$(S,10110) \vdash (Q,0110) \vdash (Q,110) \vdash (Q,10) \vdash (Q,0) \vdash (Q,\text{e})$

(V,e)

$(U,0) \vdash (F,e)$

$(U,10) \vdash (Z,0) \vdash (Z,e)$ *

$(V,110) \vdash (F,10) \vdash (F,0) \vdash (F,e)$

$(U,0110) \vdash (F,110) \vdash (F,10) \vdash (F,0) \vdash (F,e)$

Мы видим, что цепочка 10110 допускается определенным выше автоматом, если на каждом шаге выбирается правильный путь (помечен *).

Заметим, что для любой автоматной грамматики G можно построить диаграмму состояний и, следовательно, недетерминированный конечный автомат.

3.4. Построение детерминированного конечного автомата из недетерминированного.

Покажем, как из недетерминированного конечного автомата построить детерминированный конечный автомат, который как бы параллельно проверяет все возможные пути разбора и отбрасывает тупиковые. Если в НКА имеется, к примеру, выбор из трех состояний *X*, *Y* и *Z*, то в КА будет одно состояние [*XYZ*], которое представляет все три.

Пусть $M = (Q, V_T, \delta, q_0, F)$ – НКА. Построим КА $M' = (Q', V_T, \delta', q'_0, F')$ следующим образом:

(1) Множество состояний Q' состоит из множества всех подмножеств множества Q. Будем обозначать элемент $\{S_1, S_2,\ldots,S_k\}$ множества Q' через $S_1S_2\ldots S_k$. Будем предполагать, что состояния $S_1, S_2, \ldots, S_k$ расположены в некотором каноническом порядке таким образом, что, например, состояние из Q' для подмножества $\{S_1, S_2\} = \{S_2, S_1\}$ – суть S_1S_2.

(2) Множество входных литер V_T для M и M' одно и тоже.

(3) Отображение δ' определим следующим образом. $\delta'(S_1S_2\ldots S_k, T) = R_1R_2\ldots R_m$, если $\delta(\{S_1, S_2, \ldots, S_k\}, T) = \cup\delta(\{S_i\}, T) = \{R_1, R_2, \ldots, R_m\}$.

(4) Если $q_0=\{S_1, S_2, \ldots, S_k\}$, то $q'_0=S_1S_2\ldots S_k$.

(5) Если $F=\{ S_1, S_2, \ldots, S_k\}$, то $F'=S_1S_2\ldots S_k$.

3.5. Построение минимального автомата.

Для каждого конечного автомата существует бесконечное число других конечных автоматов, которые распознают то же множество цепочек. Для каждого конечного автомата существует единственный конечный автомат, распознающий то же самое множество цепочек, при этом число его состояний не больше числа состояний любого другого конечного автомата для данного множества цепочек.

Употребление слова “единственный” требует некоторого пояснения. Для любого автомата можно получить новый автомат с таким же числом состояний, просто переименовав его состояния. Однако имена состояний не имеют никакого значения для распознавания цепочек. Поэтому автоматы, которые различаются лишь именами состояний, можно считать одинаковыми. Этот единственный автомат будем называть минимальным. Естественно, что нас будет интересовать этот единственный минимальный автомат.

В этом параграфе мы рассмотрим методы построения по заданному конечному автомату минимальный автомат. Суть их в том, что минимальный автомат – это компактный вариант автоматов большего объема, а не просто еще один автомат, у которого случайно оказалось меньше состояний.

Среди состояний автомата могут быть такие, которые не достижимы из начального состояния ни для какой входной цепочки. Такие состояния называются недостижимыми. Строки, соответствующие этим состояниям, можно удалить из таблицы переходов, получив тем самым таблицу переходов автомата, который эквивалентен исходному, но имеет меньшее число состояний.

Для любого заданного автомата довольно просто составить список достижимых состояний.

Шаг 1. Начать список начальным состоянием.

Шаг 2. Для каждого состояния, уже внесенного в список, добавить все еще не занесенные в него состояния, которые могут быть достигнуты из этого нового состояния под действием одного входного символа.

Если эта процедура перестает давать новые состояния, то все достижимые состояния получены, а остальные состояния можно удалить из автомата.

Следующим шагом в минимизации автоматов будет введение понятия ***эквивалентности состояний***. Неформально два состояния эквивалентны,

если они одинаково реагируют на все возможные продолжения входной цепочки.

Будем называть два состояния q_1 и q_2 КА M эквивалентными, если КА, начав работу в любом из этих состояний, будет допускать в точности те же самые цепочки.

Для построения метода проверки эквивалентности состояний нам больше подойдет другое, более конструктивное, утверждение.

Состояния q и q' эквивалентны тогда и только тогда, когда выполняются следующие два условия:

(1) ***Условие подобия***. Состояния должны быть либо оба допускающими, либо оба отвергающими.

(2) ***Условие преемственности***. Для всех входных символов состояния q и q' должны переходить в эквивалентные состояния, т.е. их преемники должны быть эквивалентны.

На этом утверждении основан следующий метод, называемый «методом разбиения».

Шаг 1. Множество всех состояний разбивается на два класса: допускающие состояния и отвергающие состояния.

Шаг 2. Рассматриваем состояния из одного класса. Если q и q' под действием входных символов переходят в состояния из различных классов, то их необходимо разнести по различным классам.

Повторять шаг 2 до тех пор, пока возможны разбиения.

Шаг 3. Каждому классу дать новое имя состояния.

Мы будем говорить, что автомат минимальный или приведенный, если он не содержит недостижимых состояний и никакие два его состояния не эквивалентны друг другу.

Глава 4. Нисходящий разбор.

4.1 Нисходящий разбор с возвратами.

Алгоритм нисходящего разбора строит синтаксическое дерево, начиная с корня, постепенно спускаясь до уровня предложения, как было показано в параграфе 2.4.. Там демонстрировалась простота и наглядность основной идеи. Описание усложняется главным образом из-за вспомогательных операций, которые необходимы для того, чтобы выполнять возвраты с твердой уверенностью, что все возможные попытки построения дерева были предприняты. Чтобы свести к минимуму сложности, опишем этот алгоритм образно.

Вообразим, что на любом этапе разбора, в каждом узле уже построенной части дерева, находится по одному человеку. Люди, находящиеся в терминальных узлах, занимают места соответствующие символам предложения.

Некоему человеку M предстоит провести разбор предложения x. Ему необходимо отыскать вывод $Z \Rightarrow^{+} x$, где Z – начальный символ. Первым непосредственным выводом должен быть вывод $Z \Rightarrow y$, где $Z ::= y$ – некоторое правило. Пусть для Z существуют правила

$Z ::= X_1X_2\ldots X_n \mid Y_1Y_2\ldots Y_m \mid Z_1Z_2\ldots Z_k$

Сначала человек пытается применить правило $Z ::= X_1X_2\ldots X_n$. Если нельзя построить дерево, используя это правило, он пытается применить второе правило $Z ::= Y_1Y_2\ldots Y_m$. И так далее.

Как ему определить, правильно ли он выбрал непосредственный вывод $Z \Rightarrow X_1X_2\ldots X_n$? Заметим, что если вывод правилен, то для некоторых цепочек x_i будет иметь место $x = x_1x_2\ldots x_n$, где $X_i \Rightarrow^{*} x_i$ для $1 \le i \le n$.

Для осуществления разбора M возьмет себе приемного сына M_1, который должен найти вывод $X_1 \Rightarrow^{*} x_1$, для любого x_1, такого, что $x = x_1\ldots$. Если сыну M_1 удалось найти такой вывод, он закрывает цепочку x_1 в предложении x и сообщает своему отцу об успехе. M усыновляет M_2, чтобы тот нашел вывод $X_2 \Rightarrow^{*} x_2$, где $x = x_1x_2\ldots$, и ждет ответа от него, и т.д. Как только

сообщил об успехе сын M_{i-1}, M усыновит еще и M_i, чтобы тот нашел вывод $X_i \Rightarrow^* x_i$. Сообщение об успехе, пришедшее от сына M_n, означает, что разбор предложения окончен.

Как быть, если сыну M_i не удалось найти вывод $X_i \Rightarrow^* x_i$? В этом случае M_i сообщает о неудаче своему отцу; тот от него отрекается и дает старшему брату M_i, M_{i-1} распоряжение найти другой вывод. Если M_{i-1} сумеет найти другой вывод, он вновь сообщает об успехе, и все продолжается по-прежнему. Если придется отречься даже от M_1, значит, непосредственный вывод $Z \Rightarrow X_1X_2\ldots X_n$ был неверен, и необходимо попытаться воспользоваться другим.

Как действует каждый из M_i? Пусть его целью является терминал X_i. Входная цепочка имеет вид $x=x_1x_2\ldots x_{i-1}T\ldots$, где символы в $x_1x_2\ldots x_{i-1}$ уже закрыты другими людьми. M_i проверяет, совпадает ли очередной незакрытый символ T с его целью X_i. Если это так, он закрывает этот символ и сообщает об успехе. Если не совпадает, сообщает о неудаче. Если цель M_i – нетерминал X_i, то M_i поступает так же, как и его отец. Он начинает проверять правые части правил, относящиеся к нетерминалу, и, если необходимо, тоже усыновляет или отрекается от сыновей. Если все его сыновья сообщают об успехе, то M_i в свою очередь сообщает об успехе своему отцу. Если отец просит M_i найти другой вывод, а целью является терминальный символ, то M_i сообщает о неудаче, так как другого вывода не существует, В противном случае M_i просит своего младшего сына найти другой вывод и реагирует на его ответ так же, как и раньше. Если все сыновья сообщат о неудаче M_i, он сообщит о неудаче своему отцу.

Привлекательность этого метода в том, что каждый человек должен помнить лишь о своей цели, о своем отце, о своих сыновьях, а также о своем месте в грамматике и во входной цепочке. И никому не нужны сведения о том, что происходит в других местах.

Для имитации усыновления и отречения от сыновей в алгоритме используется стек типа LIFO (last in first out), или как его иногда называют, «магазин». Стек – основной механизм, используемый почти во всех типах распознавателей. Действительно, всякий раз, когда говорят о рекурсии, говорят о стеках.

Для реализации стека мы будем использовать динамический массив `S` пользовательского типа состоящий из пяти компонент. Будем предполагать, что грамматика задана в виде строки символов `Grammar`:

Z E # | $ E T + E | T | $ T F * T | F | $ F (E) | i | $

1 2 3 4 5 6 7 8 9 10 11 13 15 17 19 21 23 25 27 29

12 14 16 18 20 22 24 26 28

```
Private Type Stack
   Goal As String     ‘Цель, т.е. символ, который ищется
   Ind As Integer     ‘индекс в массиве Grammar, указывающий
                      ‘на тот символ в правой части правила
                      ‘для Goal, который рассматривается в
                      ‘данный момент
   Father As Integer  ‘Отец (индекс в стеке соответствующий
                      ‘отцу
   Son As Integer     ‘Индекс самого последнего из сыновей
   Brother As Integer ‘Индекс старшего брата
‘  Нуль в любом из полей означает, что данная величина
‘  отсутствует
End Type
Sub TopDownParser(Input As String,Grammar As String)
‘ Input – входная цепочка
‘ Grammar строка в которой хранятся правила грамматики
Dim S() As Stack         ‘Стек имитирующий усыновления
Dim CurMan As Integer    ‘Ссылка на работающего в данный
                         ‘момент человека
Dim StackTop As Integer ‘Указатель на вершину стека
Dim LockInd As Integer   ‘Индекс самого левого (незакрытого)
                         ‘символа входной цепочки
'Начальная установка
StackTop = 1
ReDim S(StackTop)
With S(StackTop)         ‘Первое усыновление.
   .Goal = ”Z”           ‘Цель усыновленного –
   .Ind = 0              ‘- начальный символ Z
   .Father = 0
   .Son = 0
   .Brother = 0
End With
CurMan = 1: LockInd = 1
```

```
'Новый человек
Do
   With S(CurMan)          ‘Новый человек изучает свою цель
      If IsTerminal(.Goal) Then         ‘Цель – терминал
         If Input(LockInd) = .Goal Then
'           Если цель совпадает с символом из предложения,
‘           человек закрывает этот символ и сообщает об успехе.
            LockInd = LockInd + 1
‘           Успех
            CurMan = .Father
            .Ind = .Ind + 1
         Else
'           Цель не совпадает – сообщает о неудаче
'           Неудача
            Do
‘              Сообщить о неудаче своему отцу. Он от вас
‘              отречется и попросит вашего старшего брата
‘              предпринять еще одну попытку
               CurMan = .Father
               .Son = S(.Son).Brother
               StackTop = StackTop – 1
               ReDim Preserve S(StackTop)
‘              Еще раз
               Do
‘                 Есть ли у вас сын, который может
‘                 предпринять еще одну попытку?
                  If .Son = 0 Then
‘                    Нет. Тогда пропустите эту правую часть –
‘                    – это не та, которая нужна
                     Do While Grammar(.Ind) <> “|”
                        .Ind = .Ind + 1
                     Loop
                     .Ind = .Ind + 1
                     Goto Met1
                  End If
‘                 У вас есть сын. Попросите его
'                 повторить попытку
                  .Ind = .Ind – 1
                  CurMan = .Son
'                 Пока его цель нетерминал, он может попытаться
'                 добиться успеха еще раз
               Loop While Not IsTerminal(.Goal)
'              Его цель – терминал. Попытка не приведет к успеху
               LockInd = LockInd – 1
            Loop
         End If
      Else
‘        Цель нового человека – нетерминал
'        Положить Ind = номер в Grammar правой части для Goal
         .Ind=RigthPart(.Goal, Grammar)
‘        Цикл
'        Просмотр правой части
Met1:    Do
            If Grammar(.Ind) = “|” Then
‘              Вы достигли конца правой части
               If .Father <> 0 Then
```

```
'                  Сообщите об успехе своему отцу
                   CurMan = .Father
                   .Ind = .Ind + 1
                Else
'                  Отца нет. Разбор окончен. Предложение
                   Message("Предложение")
                   Exit Sub
                End If
             ElseIf Grammar(.Ind) = "$" Then
'                Правых частей для цели больше нет
                If .Father <> 0 Then
'                  Сообщите о неудаче своему отцу
                   GoTo Неудача
                Else
'                  Отца нет. Разбор окончен. Не предложение
                   Message("Не предложение")
                   Exit Sub
                End If
             Else
'                Grammar(.Ind) - другая цель,
'                которую можно попытаться найти
                StackTop = StackTop + 1
                With S(StackTop)
                   .Goal = Grammar(S(CurMan).Ind)
                   .Ind = 0
                   .Father = CurMan
                   .Son = 0
                   .Brother = S(CurMan).Son
                End With
                .Son = StackTop
                CurMan = StackTop
                GoTo Новый человек
             End If
          Loop
       End If
   End With
Loop
End Sub
```

4.2.Проблемы нисходящего разбора и их решение.

Прямая левосторонняя рекурсия.

В алгоритме, описанном в параграфе 4.1, есть серьезный недостаток, который проявляется, когда цель определена с использованием левосторонней рекурсии. Если X - наша цель, а первое же правило для X имеет вид X→X…, то процесс "усыновлений" немедленно зацикливается и для решения этой задачи не хватит населения Китая.

По этой причине в грамматике использованы правила с применением правосторонней рекурсии вместо более привычной левосторонней. Лучший способ избавиться от прямой левосторонней рекурсии – записывать правила, используя итеративные и факультативные обозначения.

Фигурные скобки (итеративные обозначения).

Мы будем использовать фигурные скобки { и } как метасимволы там, где необходимо указать, что цепочка, заключенная в фигурные скобки, может либо отсутствовать, либо повторяться любое число раз.

Например, E→E+T|T можно записать как E→T{+T}.

Оба способа записи считаются эквивалентными.

Фигурные скобки позволяют нам использовать метасимвол | внутри них для указания на возможный выбор.

Например, T→T∗F|T/F|F можно записать как T→F{∗F|/F}.

Квадратные скобки.

Квадратные скобки [и] часто используют для того, чтобы указать факультативную цепочку, т.е. цепочка, заключенная в квадратые скобки, может либо присутствовать, либо отсутствовать.

Круглые скобки как метасимволы.

В правых частях правил оператор конкатенации имеет более высокий приоритет чем оператор выбора. Поэтому AB|C означает либо AB, либо C. Когда необходимо отказаться от обычного порядка, можно использовать круглые скобки почти так же, как это делается в арифметических выражениях. Таким образом, A(B|C) означает либо AB, либо AC. Круглые скобки как метасимволы будут полезны при факторизации в правых частях. Однако возникает проблема, еогда круглые скобки используются как терминальные символы языка.

Факторизация.

Если круглые скобки используются как метасимволы, можно записать правила

$$U \rightarrow xy|xw|\ldots|xz \text{ как } U \rightarrow x(y|w|\ldots|z),$$

где общая для всех альтернатив голова x вынесена за скобки. Скобки могут быть вложеными на какую угодно глубину, точно так же как в арифметических выражениях.

Рассмотрим два принципа, на основании которых правила грамматики, включающие прямую левостороннюю рекурсию, преобразуются в эквивалентные правила, использующие итерацию.

(1) Факториация. Если существуют правила вида $U \rightarrow xy|xw|\ldots|xz$, то их надо заменить на $U \rightarrow x(y|w|\ldots|z)$, где скобки являются метасимволами.

Допустима факторизация и в более общей форме. Например если $y=y_1y_2$ и $w=y_1w_2$, можно заменить $U \rightarrow x(y_1y_2|y_1w_2|\ldots|z)$ на $U \rightarrow x(y_1(y_2|w_2)|\ldots|z)$.

Заметим, что исходные правила $U \rightarrow x|xy$ мы преобразуем к виду $U \rightarrow x(y|e)$, где e – пустая цепочка. Когда бы ни использовалось подобное преобразование, e всегда помещается как последняя альтернатива, так как мы принимаем условие, что если цель – e, то эта цель всегда сопоставляется.

Помимо того, что факторизация позволяет нам исключить прямую рекурсию, ее использование сокращает размеры грамматики и позволяет проводить разбор более эффективно.

После факторизации в грамматике останется не более одной правой части с прямой левосторонней рекурсией для каждого нетерминала. Если такая правая часть есть, мы заменяем рекурсивные правила на эквивалентные правила с итерацией следующим образом.

(2) Пусть $U \rightarrow x|y|\ldots|z|Uv$ – правила, у которых осталась леворекурсивная правая часть.Эти правила означают, что членом синтаксического класса U является x, y или z, за которыми либо ничего не следует, либо следует сколько-то v. Тогда преобразуем эти правила к виду $U \rightarrow (x|y|\ldots|z)\{v\}$.

Общая левосторонняя рекурсия.

Мы не решили всей проблемы левосторонней рекурсии: с прямой левосторонней рекурсией покончено, но общая левосторонняя рекурсия еще осталась. Таким образом, правила $U \to Vx$ и $V \to Uy|w$ дают вывод $U \Rightarrow^{+} Uyx$. Избавиться от этого не так просто, но обнаружить такую ситуацию можно. Исключим из исходной грамматики все правила с прямой левосторонней рекурсией. Символ U, получившейся в результате этих преобразований грамматики, может быть леворекурсивным тогда и только тогда, когда U $FIRST^{+}$ U.

Разбор без возвратов.

Программа разбора в компиляторе ни в коем случае не должна прибегать к возвратам. Мы должны быть уверены в том, что каждая предполагаемая цель верна. Это необходимо потому, что нам предстоит связать семантику с синтаксисом, и по мере того, как мы будем прогнозировать и находить цели, эти символы будут обрабатываться семантически. Например, при обработке описаний переменных их имена помещаются в таблицу символов; при обработке арифметических выражений проверяется, совместимы ли типы операндов.

Если произошел возврат из-за того, что прогнозируемая цель неверна, придется уничтожить результаты семантической обработки, проделанной во время поисков этой цели. Сделать это не так-то просто, поэтому надо стараться проводить грамматический разбор без возвратов.

Для того чтобы избавиться от возвратов, в компиляторах в качестве контекста обычно используется следующий незакрытый символ исходной программы. Тогда на грамматику налагается следующее ограничение: если есть альтернативы x|y|…|z, то множества символов, которыми могут начинаться выводимые из x, y, …, z слова, должны быть попарно различны. То есть если $x \Rightarrow^{*} Au$ и $y \Rightarrow^{*} Bv$, то $A \neq B$. Если это требование выполнено,

можно довольно просто определить, какая из альтернатив x, y или z – наша цель. Заметим, что факторизация оказывает здесь большую помощь. Если есть правило U→xy|xz, то преобразование этого правила к виду U→x(y|z) помогает сделать множества первых символов для разных альтернатив непересекающимися. В следующем разделе рассмотрим применение этого метода.

4.3. Рекурсивный спуск.

В некоторых компиляторах синтаксический анализатор содержит по одной рекурсивной процедуре для каждого нетерминала U. Каждая такая процедура осуществляет разбор фраз, выводимых из U. Процедуре сообщается, с какого места данной входной цепочки следует начинать поиск фразы, выводимой из U. Следовательно, такая процедура – целенаправленная, или прогнозирующая. Мы предполагаем, что сможем найти такую фразу. Процедура ищет эту фразу, сравнивая входную цепочку в указанном месте с правыми частями правил для U и вызывая по мере необходимости другие процедуры для распознавания промежуточных целей. В действительности во время этого разбора дерево строится точно так же, как в алгоритме разбора, описанном в разделе 4.1 (только без возвратов). Чтобы проиллюстрировать этот способ, напишем процедуры для нетерминальных символов следующей грамматики:

<инструкция>::=<переменная> := <выражение>|

|`IF` <выражение> `THEN` <инструкция>|

|`IF` <выражение> `THEN` <инструкция> `ELSE` <инструкция>

<переменная>::=`i`|`i`(<выражение>)

<выражение> ::=<терм>|<выражение> + <терм>

<терм> ::=<множитель>|<терм> * <множитель>

<множитель> ::=<переменная>|(<выражение>)

Чтобы удобнее было работать, перепишем грамматику следующим образом:

<инструкция>::=<переменная> := <выражение>|

|IF <выражение> THEN <инструкция> [ELSE <инструкция>]

<переменная>::=i|i(<выражение>)

<выражение> ::=<терм> {+ <терм>}

<терм> ::=<множитель> {* <множитель>}

<множитель> ::=<переменная>|(<выражение>)

Мы полагаем, что сможем провести разбор без возвратов. Для того чтобы возвратов не было, в качестве контекста будет использоваться единственный символ, следующий за уже разобранной частью фразы. Мы запишем процедуры на языке Visual Basic с соблюдением следующих условий:

1. Глобальная переменная `NextSymbol` всегда содержит тот символ исходной программы (входной цепочки), который будет обрабатываться следующим. При вызове процедуры для поиска новой цели первый символ, который она должна исследовать, уже находится в `NextSymbol`.
2. Перед тем как выйти из процедуры с сообщением об успехе, символ, следующий за уже разобранной подцепочкой, помещается в `NextSymbol`.
3. Процедура `Scaner` готовит очередной символ исходной программы и помещает его в `NextSymbol`.
4. Процедура `CallError` вызывается в тех случаях, когда обнаружена ошибка. Она выдает сообщение об ошибке и передает управление обратно. После возврата работа продолжается так, как будто бы никакой ошибки не было.
5. Для того чтобы начать синтаксический анализ инструкции, надо вызвать процедуру `Scaner`, которая поместит первый символ в `NextSymbol`, а затем вызвать процедуру `State`.

```
Option Explicit
```

```
Dim NextSymbol As String
Sub State()
If NextSymbol = "IF" Then
   Scaner
   Expression
   If NextSymbol <> "THEN" Then
      CallError
   Else
      Scaner
      State
      If NextSymbol = "ELSE" Then
         Scaner
         State
      End If
   End If
Else
   Variable
   If NextSymbol <> ":=" Then
      CallError
   Else
      Scaner
      Expression
   End If
End If
End Sub
Sub Variable()
If NextSymbol <> "i" Then
   CallError
Else
   Scaner
   If NextSymbol = "(" Then
      Scaner
      Expression
      If NextSymbol <> ")" Then
         CallError
      Else
         Scaner
      End If
   End If
End If
End Sub
Sub Expression()
Term
Do While NextSymbol = "+"
   Scaner
   Term
Loop
End Sub
Sub Term()
Factor
```

Процедура для <инструкция>. Мы полагаем, что можно определить вид инструкции по первому символу.

Вызвать процедуру для обработки выражения. Следующим символом должен быть “THEN”, затем инструкция.

Рекурсивный вызов.

Если встретится символ “ELSE”, провести его анализ.

Это не условная инструкция. Должна быть инструкция присваивания. Перейти к разбору переменной, проверить, есть ли символ :=, и затем анализировать выражение.

Конец процедуры для <инструкция>.

Процедура для <переменная>.

Переменная должна начинаться с i.

Взять символ, следующий за i. В том и только в том случае, когда это открывающая скобка, перейти к разбору выражения и проверить, следует ли за выражением закрывающая скобка.

Занести в NextSymbol символ, следующий за обработанной конструкцией.

Процедура для <выражение>.

Выражение должно начинаться термом.

После этого ожидается любое количество конструкций вида “+<терм>”. Возврат, если следующий символ не “+”.

Процедура для <терм>.

```
Do While NextSymbol = "*"
   Scaner
   Factor
Loop
End Sub
Sub Factor()
If NextSymbol = "(" Then
   Scaner
   Expression
   If NextSymbol <> ")" Then
      CallError
   Else
      Scaner
   End If
Else
   Variable
End If
End Sub
```

Процедура аналогична процедуре для <выражение> и не требует пояснений.

Процедура для <множитель>.
По первому символу выбирается альтернатива. Это выражение, заключенное в скобки. Проверить наличие закрывающей скобки и сканировать следуюший символ.

Это переменная.

Во всех этих процедурах необычно то, что они не требуют локальных переменных. Фактически единственная используемая переменная – это глобальная переменная `NextSymbol`. По существу, мы просто используем обычный стековый механизм, связывающий процедуры во время выполнения для того, чтобы имитировать стек, используемый в разделе 4.1. Преимущества этого метода очевидны. Программируя, можно реорганизовать правила так, чтобы они согласовывались с процедурами. Предполагается, что автор компилятора настолько хорошо знаком с исходным языком, что может провести реорганизацию, которая избавляет от возвратов. Метод сохраняет свою гибкость и по отношению к семантической обработке. С этой целью в любое место процедуры можно включить группу команд, не откладывая семантическую обработку до тех пор, пока будет обнаружена вся фраза.

Основной недостаток состоит в том, что на программирование и отладку затрачивается больше усилий, чем в частично автоматизированных системах. Тем не менее, это разумный метод и используется он во многих компиляторах.

Глава 5. Грамматики простого предшествования.

5.1.Отношения предшествования.

Рассмотрим схему разбора, в которой применяется восходящий метод. Напомним, что при использовании этого метода в текущей сентенциальной форме повторяется поиск основы (самой левой простой фразы *и*), которая в соответствии с правилом $U \to u$ приводится к нетерминалу U. При применении любого из методов восходящего разбора возникает вопрос – как найти основу и выяснить, к какому нетерминалу нужно ее приводить. Мы рассмотрим как этот вопрос решается для определенного класса грамматик, называемых грамматиками (простого) предшествования. Для всех грамматик это проблемы не решает.

Как же найти основу, если задана сентенциальная форма x? Хотелось бы, двигаясь слева направо и рассматривая только два соседних символа одновременно, определить, нашли ли мы хвост основы. Затем, продвигаясь назад к левому концу сентенциальной формы, найти голову основы, принимая каждый раз решение только по двум соседним символам. То есть мы сталкиваемся с такой проблемой: если задана цепочка $\ldots RS \ldots$, как выяснить в каких случаях R является хвостом основы, или RS вместе входят в основу, или S является головой основы? Хотелось бы, не приступая еще к разбору, исследовать грамматику и принять решение относительно каждой пары символов R и S.

Рассмотрим теперь два символа R и S из объединенного словаря $V_T \cup V_N$ грамматики G. Предположим, что существует (каноническая) сентенциальная форма $\ldots RS \ldots$. На некотором этапе канонического разбора либо R, либо S (или оба символа одновременно) должны войти в основу. При этом возникают следующие три возможности:

(1) R – часть основы, а S нет. Эту ситуацию мы будем записывать как $R \circ > S$ и говорить, что R больше S или что R предшествует S, поскольку символ R будет редуцирован раньше, чем S. Заметим, что R должен быть

последним символом в правой части некоторого правила $U \rightarrow \ldots R$, и, поскольку основа находится слева от S, S должен быть терминалом.

(2) Оба символа R и S входят в основу. Будем записывать это как $R \doteq S$. У них одинаковое значение предшествования, и они должны редуцироваться одновременно. Очевидно, в грамматике должно быть правило $U \rightarrow \ldots RS \ldots$.

(3) S – часть основы, а R нет. Отношение между ними будем записывать как $R <\circ S$ и говорить, что R меньше, чем S. Символ S должен быть первым в правой части некоторого правила $U \rightarrow S \ldots$.

Если (канонической) сентенциальной формы $\ldots RS \ldots$ не существует, будем считать, что между упорядоченной парой символов (R, S) не определено никакое отношение (предшествования). Заметим, что ни одно из трех определенных выше отношений предшествования не является симметричным.

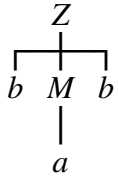
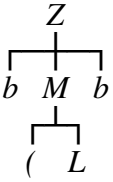
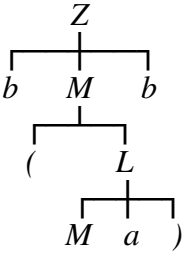
Рассмотрим в качестве примера грамматику G:

$Z \rightarrow bMb$

$M \rightarrow (L \mid a$

$L \rightarrow Ma)$

Языку $L(G)$ принадлежат цепочки bab, $b(aa)b$, $b((aa)a)b$, $b(((aa)a)a)b$ и т.д. В каждом столбце приведенной ниже таблицы показана сентециальная форма, ее синтаксическое дерево, основа и отношения, которые можно из этого дерева получить.

Сентенциальная форма:	bab	$b(Lb$	$b(Ma)b$
Синтаксическое дерево:			
Основа:	a	$(L$	$Ma)$
Отношения,	$b <\circ a$	$b <\circ ($	$(<\circ M$

определяемые деревом:	$a \circ\!> b$	$(\doteq L$ $L \circ\!> b$	$M \doteq a$ $a \doteq)$ $) \circ\!> b$

Матрица предшествования для грамматики G – матрица, в которой указываются все отношения предшествования. Элемент этой матрицы B_{ij} содержит отношение между парой символов (S_i, S_j). Пустой элемент матрицы свидетельствует о том, что между соответствующими двумя символами отношение предшествования не определено.

	Z	*B*	*M*	*L*	*a*	(	)
Z							
b			$\doteq$		$<\circ$	$<\circ$	
M		$\doteq$			$\doteq$		
L		$\circ>$			$\circ>$		
a		$\circ>$			$\circ>$		$\doteq$
(			$<\circ$	$\doteq$	$<\circ$	$<\circ$	
)		$\circ>$			$\circ>$		

Может показаться, что трудно найти все отношения предшествования между символами грамматики. Складывается впечатление, что придется просмотреть довольно много синтаксических деревьев, чтобы найти *все* отношения. Да и как гарантировать, что все отношения найдены? В следующем параграфе мы переопределим отношения так, что их можно будет без особого труда построить для любой грамматики.

Как воспользоваться отношениями предшествования при разборе предложений? Если между какой-либо парой символов (*R, S*) определено более чем одно отношение, они бесполезны. Если же между любой парой символов определено не более одного отношения, отношения предшествования позволят найти основу любой сентенциальной формы. Основой любой сентенциальной формы $S_1S_2...S_n$ является самая левая подцепочка $S_j...S_i$, такая, что

$S_{j-1} <^{\circ} S_j$

$S_j \doteq S_{j+1} \doteq S_{j+2} \doteq \ldots \doteq S_i$

$S_i {}^{\circ}> S_{i+1}$

(Чтобы учесть и тот случай, когда символ S_i (S_j) основы является самым первым (последним) символом сентенциальной формы, последнее утверждение придется слегка изменить; мы это сделаем позже).

Приведем для примера разбор предложения *b(aa)b* грамматики *G* с использованием матрицы предшествования.

Шаг	*Сентенциальная форма*	*Основа*	*Привести основу к*	*Построенный непосредственный вывод*
1	*b (a a) b* $<^{\circ}\ <^{\circ}\ {}^{\circ}>\ \doteq\ {}^{\circ}>$	*b*	*M*	$b(Ma)b \Rightarrow b(aa)b$
2	*b (M a) b* $<^{\circ}\ <^{\circ}\ \doteq\ \doteq\ {}^{\circ}>$	*Ma)*	*L*	$b(Lb \Rightarrow b(Ma)b$
3	*b (L b* $<^{\circ}\ \doteq\ {}^{\circ}>$	*(L*	*M*	$bMb \Rightarrow b(Lb$
4	*b M b* $\doteq\ \doteq$	*bMb*	*Z*	$Z \Rightarrow bMb$

Упражнения.

1. Используя для нахождения основ матрицу предшествования, заданную в параграфе 5.1, проведите разбор следующих предложений грамматики: *bab*, *b(aa)b*, *b(((aa)a)a)b*.
2. Попытайтесь найти все отношения предшествования для грамматики:

   Z::=E#

   E::=T+E|T

   T::=F∗T|F

   F::=(E)|i.

5.2.Определение и построение отношений.

В предыдущем разделе мы определили три отношения предшествования в терминах синтаксических деревьев сентенциальных форм. Теперь мы хотим определить их заново, основываясь только на правилах грамматики.

Если задана грамматика G, то отношения предшествования из словаря $V_T \cup V_N$ определяются следующим образом:

(1) $R \doteq S$ тогда и только тогда, когда в G есть правило $U \to \ldots RS \ldots$.

(2) $R <\circ S$ тогда и только тогда, когда существует правило $U \to \ldots RV \ldots$, такое, что справедливо отношение $V\ FIRST^{+}\ S$.

(3) $R \circ > S$ тогда и только тогда, когда S – терминал и существует правило $U \to \ldots VW \ldots$, такое, что справедливы соотношения $V\ LAST^{+}\ R$ и $W\ FIRST^{*}\ S$.

Грамматику G называют грамматикой (простого) предшествования или грамматикой (1, 1) предшествования, если

1) между любыми двумя символами из словаря определено не более чем одно отношение;
2) ни у каких двух продукций нет одинаковых правых частей.

Запись (1, 1) означает, что для принятия решения о том, действительно ли предполагаемая основа является основой, мы используем по одному символу слева и справа от нее. То есть, если в основу входит символ R, мы по одному только следующему за ним символу S определяем, является ли R хвостом основы; аналогично решается вопрос о голове основы.

Следует не упустить из виду случай, когда в основу входит символ S_1 – первый символ сентенциальной формы, так как в этом случае символа S_0 такого, что $S_0 <\circ S_1$, нет. Решим эту проблему следующим образом:

> Каждая сентенциальная форма заключена между символами “#” и “#” (при этом предполагается, что “#” не есть символ из данной грамматики). Более того, будем считать, что $\#<\circ S$ и $S \circ >\#$ для любого символа S из грамматики.

Построение отношения простого предшествования $\doteq$ не требует практически никаких объяснений. Нужно только просмотреть правые части правил и установить отношение $R \doteq S$ для всех R и S, таких, что $\ldots RS \ldots$ - правая часть.

Установление отношений $<\circ$ и $\circ>$ - дело более трудоемкое, но и их вычислить несложно.

Отношение $<\circ$ равно произведению отношений $\doteq$ и $FIRST^+$, то есть

$$<\circ = (\doteq)(FIRST^+)$$

Пусть I – единичное отношение. $R \circ > S$ тогда и только тогда, когда S – терминал и

$$R\ ((LAST^+)^{-1})(\doteq)(I+FIRST^+)\ S$$

Упражнения.

1. Покажите, что следующая грамматика не является грамматикой предшествования:

   Z::=bEb, E::=E+T|T
2. Покажите, что следующая грамматика не является грамматикой предшествования:

   Z::=b E1 b, E1::=E|E+T|T|i, T::=i|(E1)
3. Напишите на каком-нибудь языке высокого уровня и отладьте программу для построения и вывода отношений предшествования. Входной информацией для программы должна быть заданная в удобной форме грамматика.

5.3.Алгоритм разбора.

При практическом применении отношений предшествования для распознавания предложений нам потребуется способ их компактного

представления. Обычно для этой цели служит матрица P, элементы которой имеют значения:

P_{ij}=0, если S_i и S_j несравнимы

P_{ij}=1, если $S_i <\circ S_j$

P_{ij}=2, если $S_i \doteq S_j$

P_{ij}=3, если $S_i \circ> S_j$

Для грамматики предшествования такое представление возможно, так как известно, что между любыми двумя символами определено не более одного отношения.

Сами правила должны находиться в таблице, имеющей такую структуру, которая позволяет по заданной правой части найти содержащие ее правила, а затем указать соответствующую левую часть.

Алгоритм разбора работает следующим образом. Символы входной цепочки просматриваются слева направо и заносятся в стек S до тех пор, пока не окажется, что верхний символ стека находится в отношении $\circ>$ к следующему входному символу. Это означает, что верхний символ стека является хвостом основы и, следовательно, вся основа уже в стеке. Затем в стеке ищется голова основы (она находится в отношении $<\circ$ с предыдущим символом). После этого основу надо найти в списке правил, и заменить ее в стеке тем нетерминалом, к которому ее надлежит привести. Процесс повторяется до тех пор, пока в стеке не останется один символ “Z” (Z – начальный символ грамматики) и следующим входным символом не станет “#”.

```
Sub PrecedenceParser(Input() As String)
‘ Алгоритм разбора для грамматики простого предшествования
Dim S() As String ‘ Стек
Dim i As Integer  ‘ Счетчик стека
Dim j As Integer  ‘ Индекс для адресации нескольких элементов
                  ‘ в стеке
Dim k As Integer  ‘ Номер очередного входного символа
Dim R As String   ‘ Вспомогательная переменная для хранения
                  ‘ основы
Dim Q As String   ‘ Вспомогательная переменная для хранения
```

```
                    ‘ нетерминала к которому приводится основа
Dim F As Boolean    ‘ Флаг = True если существует правило с
                    ‘ соответствующей правой частью
i = 1
k = 1
ReDim S(i)
S(i) = “#”          ‘ Заносим в стек специальный символ
Do
‘ Заносим в стек символы до тех пор, пока там не окажется
‘ вся основа
   Do While Not IsGreater(S(i), Input(k))
      i = i + 1
      k = k + 1
      ReDim Preserve S(i)
      S(i) = Input(k)
   Loop
   j = i
   R = “”
   ‘ Продвигаясь назад в стеке, формируем основу в R
   Do While Not IsLess(S(j – 1), S(j))
      R =S(j) & R
      j = j – 1
   Loop
   ‘ Проверяем есть ли правило с правой частью равной R
   ‘ Если есть, то поместить соответствующую левую часть в Q
   F = RightPart(Grammar, R, Q)
   If Not F Then
      If i = 2 And S(i) = “Z” And Input(k) = “#” Then
         Message(“Предложение”)
      Else
         Message(“Не предложение”)
      End If
      Exit Sub
   End If
   ‘ Исключаем из стека основу и помещаем туда нетерминал,
   ‘ к которому она приводится
   i = j
   ReDim Preserve S(i)
   S(i) = Q
Loop
End Sub
```

В этом алгоритме функции `IsGreater` и `IsLess` реализуют отношения ◦> и <◦ соответственно. Функция `RightPart` проверяет есть ли в грамматике `Grammar` правило с заданной правой частью и, если есть, возвращает в своем третьем параметре соответствующую левую часть. Подробности реализации этой функции мы опускаем.

В следующей таблице приведем для примера разбор цепочки *b(aa)b*.

Шаги	*Стек* `S(1),…,S(i)`	*Отношение*	*Входная цепочка* `Input(k),…`
1	#	<◦	*b (a a) b #*
2	*# b*	<◦	*(a a) b #*
3	*# b (*	<◦	*a a) b #*
4	*# b (a*	◦>	*a) b #*
5	*# b (M*	≐	*a) b #*
6	*# b (M a*	≐	*) b #*
7	*# b (M a)*	◦>	*b#*
8	*# b (L*	◦>	*b#*
9	*# b M*	≐	*b#*
10	*# b M b*	◦>	#
11	*# Z*		#

Упражнения.

1. Используя алгоритм из параграфа 5.3, и матрицу предшествования из параграфа 5.1, попытайтесь разобрать следующие цепочки (не все они являются предложениями):
   #*aa*#, #((*aa*)*a*)#, #()#, #((((*aa*)*a*)*a*)*a*)#.

5.4. Функции предшествования.

Матрица предшествования может занимать слишком большой объем памяти. Если в языке 100 символов, понадобится матрица, состоящая из 100×100 элементов. Однако во многих случаях информацию, задаваемую в матрице, можно представить двумя функциями f и g, такими, что

из R≐S следует f(R)=g(S)

из R<◦S следует f(R)<g(S)

из R◦>S следует f(R)>g(S)

для всех символов грамматики. Это называется *линеаризацией* матрицы. Для матрицы предшествования рассмотренной в параграфе 5.1., функции f и g таковы:

	Z	*B*	*M*	*L*	*a*	*(*	*)*
f	1	4	7	8	9	2	8
g	1	7	4	2	7	5	9

Отметим, что эти функции не единственны – если для данной матрицы найдется хотя бы одна пара функций f и g, то для той же матрицы существует бесконечное количество таких функций. Однако есть много матриц предшествования, для которых эти функции не существуют. Нельзя линеаризовать, например, даже такую матрицу предшествования размером 2×2 для символов S_1 и S_2:

$\triangleq$ $\circ>$

$\triangleq$ $\triangleq$

Если бы функции существовали, то из их определения и матрицы следовало бы

$$f(S_1)>g(S_2),\ g(S_2)=f(S_2)$$

$$f(S_2)=g(S_1),\ g(S_1)=f(S_1),$$

что ведет к противоречивому утверждению $f(S_1)>f(S_1)$.

Функции предшествования (если они существуют) строятся следующим образом:

1. Начертить ориентированный граф с 2×n узлами, помеченными $f_1,\ldots,f_n$ и $g_1,\ldots,g_n$. Если $S_i\circ>=S_j$, провести дугу от f_i к g_j. Если $S_i<\circ=S_j$, провести дугу от g_j к f_i.
2. Каждому узлу поставить в соответствие число, равное числу узлов, в которые можно попасть из данного узла (включая и этот узел). Число, поставленное в соответствие f_i, и есть $f(S_i)$; число, поставленное в соответствие g_i, и есть $g(S_i)$.
3. Проверить, не противоречат ли значения построенных функций f и g исходным отношениям. Если нет противоречий, то функции f и g построены правильно. Если противоречие есть, то функций предшествования не существует.

Когда компилятор обнаруживает ошибку в программе, он должен попытаться нейтрализовать ее и продолжать разбор, чтобы найти другие ошибки (иногда такое благое намерение приводит к большому числу так

называемых «наведенных» ошибок). Поэтому важно обнаружить ошибку как можно раньше, чтобы можно было ее нейтрализовать. Из-за применения функций процесс обнаружения ошибок может затянуться. Линеаризация ведет к потере информации, потому что неизвестно, существует ли в действительности отношение между двумя символами. Для иллюстрации проведем разбор цепочки *ba)))))b*, используя сначала матрицу, а затем функции.

Разбор с использованием матрицы:

Шаг	*Стек*	*Отношение*	*Входная цепочка*
1	#	<◦	*ba)))))b#*
2	*#b*	<◦	*a)))))b#*
3	*#ba*	≗	*)))))b#*
4	*#ba)*	*Нет отношения*	*))))b#*

Разбор с использованием функций:

Шаг	*Стек*	*f Отношение g*	*Входная цепочка*
1	#	1<7	*ba)))))b#*
2	*#b*	4<7	*a)))))b#*
3	*#ba*	9=9	*)))))b#*
4	*#ba)*	8<9	*))))b#*
5	*#ba))*	8<9	*)))b#*
6	*#ba)))*	8<9	*))b#*
7	*#ba))))*	8<9	*)b#*
8	*#ba)))))*	8>7	*b#*

При использовании функций до обнаружения ошибки было выполнено на четыре шага больше, чем в первом случае.

5.5. Трудности, возникающие при построении грамматик предшествования.

Очень часто между двумя символами определено более чем одно отношение. Единственное, что мы можем сделать, - это изменить грамматику так, чтобы обойти конфликт.

Одна из таких проблем может возникнуть как следствие левосторонней рекурсии. Предположим, что существует некоторое правило U::=U… . Если есть другое правило вида V::=…SU…, получается, что одновременно S≗U и S<◦U. Иногда можно избавиться от такого конфликта, вводя еще один

нетерминал W и промежуточное правило. Заменим V::=…SU… на V::=…SW…, W::=U, где W – новый символ, при этом получим, что S≐W и S<◦U. Такой прием называют *стратификацией* или *разделением*. При правосторонней рекурсии те же проблемы возникают с отношениями ◦> и ≐. Стратификация не всегда спасает, так как она часто приводит к конфликтам иного рода. Если одновременно R<◦S и R◦>S, то лучший выход – применить другую технику.

Покажем, как делается стратификация, воспользовавшись следующей грамматикой для арифметических выражений:

E::=E+T|T

T::=T∗F|F

F::=(E)|*i*

Из первого правила следует, что +≐T, а так как символ T леворекурсивен, получаем также и +<◦T. Та же самая проблема возникает и с символами) и E. Без ущерба для структуры предложений и не меняя языка, можно заменить эту грамматику на такую:

$E::=E_1$

$E_1::=E_1+T_1|T_1$

T::=T∗F|F

$T_1::=T$

F::=(E)|*i*

Для этой грамматики матрица и функции предшествования имеют следующий вид:

	E	E_1	T_1	T	F	*i*	(	+	*	)	f	g
E										≗	2	2
E_1								≗		◦>	4	3
T_1								◦>		◦>	5	4
T								◦>	≗	◦>	6	5
F								◦>	◦>	◦>	7	6
i								◦>	◦>	◦>	7	7
(	≗	<◦	<◦	<◦	<◦	<◦	<◦				2	7
+			≗	<◦	<◦	<◦	<◦				4	4
*					≗	<◦	<◦				6	6
)								◦>	◦>	◦>	7	2

Этот пример может создать впечатление, что изменения не столь значительны, однако если в грамматике 100 правил и около 100 с лишним символов, то даже искушенный человек затратит немало времени на то, чтобы переделать такую грамматику в грамматику предшествования.

Часть 2. Формальная семантика языков программирования.

Глава 1. Введение.

Целью этой части является описание некоторых основных идей и методов, используемых в формальной семантике, иллюстрация этих идей на интересных приложениях и исследование взаимосвязи между различными методами.

Основу формальной семантики составляет строгое описание смысла или поведения программ, аппаратных частей компьютеров и т.д. Необходимость в строгости возникает потому, что она может раскрыть двусмысленность и скрытые сложности в кажущихся совершенно четко определенными документах (например, в руководствах по языкам программирования), и она формирует основу для реализации, анализа и верификации (в частности доказательство правильности).

1.1. Методы описания семантики.

Принято различать синтаксис и семантику языков программирования. Синтаксис занимается грамматической структурой программ. Так синтаксический анализ программы

$z := x; x := y; y := z$

покажет, что она состоит из трех операторов разделенных символом точка с запятой. Каждый из операторов состоит из переменной, знака присваивания и выражения, которое является переменной.

Семантика занимается смыслом грамматически правильных программ. Таким образом, смысл приведенной выше программы – поменять значения переменных x и y (и установить значения z равным y). Если мы собираемся разобрать смысл более детально, то мы должны будем рассматривать грамматическую структуру программы и использовать объяснения смысла

- последовательности операторов разделенных точкой с запятой;
- оператора состоящего из переменной, знака присваивания и выражения.

Объяснения могут быть формализованы различными способами. Мы рассмотрим три наиболее известных подхода.

Операционная семантика: смысл конструкции определяется вычислениями которые она вызывает когда выполняется на компьютере. Операционная семантика интересуется тем КАК достигается результат выполнения программы.

Денотационная семантика: смысл моделируется математическими объектами, которые представляют результат выполнения программы. Таким образом, денотационная семантика интересуется только результатом выполнения программы, а не тем как он достигается.

Аксиоматическая семантика: специфические свойства результата выполнения программы выражаются как утверждения. Таким образом могут существовать такие аспекты выполнения, которые игнорируются данным подходом.

Чтобы почувствовать различия в этих подходах рассмотрим, как в каждом из них выражается смысл приведенного выше примера.

О п е р а ц и о н н а я с е м а н т и к а.

Операционный подход к смыслу конструкции говорит о том как выполнить ее:

- чтобы выполнить последовательность операторов разделенных точкой с запятой, мы должны выполнить каждый оператор один за другим слева направо;
- чтобы выполнить оператор присваивания, мы определяем значение второй переменной и присваиваем его первой.

Мы запишем выполнение программы в состоянии в котором x имеет значение 5, y имеет значение 7 и z имеет значение 0 следующей последовательностью выводов:

$\langle z := x; x := y; y := z, [x\textbf{:}\ 5, y\textbf{:}\ 7, z\textbf{:}\ 0]\rangle \Rightarrow$

$\langle x := y; y := z, [x\textbf{:}\ 5, y\textbf{:}\ 7, z\textbf{:}\ 5]\rangle \Rightarrow$

$\langle y := z, [x\textbf{:}\ 7, y\textbf{:}\ 7, z\textbf{:}\ 5]\rangle \Rightarrow$

$[x\textbf{:}\ 7, y\textbf{:}\ 5, z\textbf{:}\ 5]$

На первом шаге мы выполняем оператор z := x и значение z изменяется на 5, тогда как x и y не изменяются. Оставшаяся программа теперь x := y; y := z. После второго шага значение x – 7 и оставшаяся программа y := z. Третий и последний шаг изменяет значение y на 5. Таким образом, начальные значения x и y обменены используя z в качестве вспомогательной переменной.

Это объяснение дает нам абстракцию того, как программа выполняется на компьютере. Важно отметить, что это действительно абстракция: мы игнорируем детали выполнения, такие как использование регистров и адресов переменных. Таким образом, операционная семантика не зависит от архитектуры ЭВМ и стратегий реализации.

Во второй главе формализуется такой вид операционной семантики, которая часто называется структурной операционной семантикой (или семантикой малых шагов).

Другой вид операционной семантики называется натуральной семантикой (или семантикой больших шагов) и отличается от структурной операционной семантики тем, что скрывает еще больше деталей выполнения программы. В натуральной семантике выполнение нашей программы представляется следующим деревом вывода:

$$\dfrac{\dfrac{\langle z := x, s_0\rangle \to s_1 \quad \langle x := y, s_1\rangle \to s_2}{\langle z := x; x := y, s_0\rangle \to s_2} \qquad \langle y := z, s_2\rangle \to s_3}{\langle z := x; x := y; y := z, s_0\rangle \to s_3}$$

где

$s_0 = [x\textbf{:}\ 5, y\textbf{:}\ 7, z\textbf{:}\ 0]$

$s_1 = [x: 5, y: 7, z: 5]$

$s_2 = [x: 7, y: 7, z: 5]$

$s_3 = [x: 7, y: 5, z: 5]$

Читается следующим образом: выполнение $z := x$ в состоянии s_0 имеет своим результатом состояние s_1; выполнение $x := y$ в состоянии s_1 приводит к состоянию s_2. Таким образом, выполнение $z := x ; x := y$ в состоянии s_0 приводит к состоянию s_2. Далее, выполнение $y := z$ в состоянии s_2 приводит к состоянию s_3. Это записывается

$$\langle z := x; x := y; y := z, s_0\rangle \rightarrow s_3$$

но теперь мы скрыли объяснение того, как это на самом деле достигнуто.

В третьей главе мы используем натуральную семантику как основу для доказательства правильности реализации простого языка программирования.

Д е н о т а ц и о н н а я с е м а н т и к а.

В денотационной семантике мы концентрируемся на результате выполнения программ и моделируем это математическими функциями:

- результатом последовательности операторов будет функциональная композиция результатов отдельных операторов;
- результатом оператора присваивания будет функция переводящая одно состояние в другое, отличающееся от исходного значением первой переменной.

Для нашего примера мы имеем функции

$$\mathcal{S}[z := x],\ \mathcal{S}[x := y],\ \mathcal{S}[y := z]$$

для каждого из операторов присваивания. Для всей программы имеем функцию

$$\mathcal{S}[z := x; x := y; y := z] = \mathcal{S}[y := z] \circ \mathcal{S}[x := y] \circ \mathcal{S}[z := x]$$

Заметим, что порядок операторов изменился потому что мы используем композицию функций, где $(f \circ g)\ s$ означает $f(g\ s)$. Если мы хотим определить результат выполнения программы в каком-либо состоянии, тогда мы можем

применить функцию к этому состоянию и вычислить результирующее состояние.

$\mathcal{S}[z := x; x := y; y := z]([x: 5, y: 7, z: 0]) =$

$(\mathcal{S}[y := z] \circ \mathcal{S}[x := y] \circ \mathcal{S}[z := x])([x: 5, y: 7, z: 0]) =$

$\mathcal{S}[y := z](\mathcal{S}[x := y](\mathcal{S}[z := x]([x: 5, y: 7, z: 0]))) =$

$\mathcal{S}[y := z](\mathcal{S}[x := y]([x: 5, y: 7, z: 5])) =$

$\mathcal{S}[y := z]([x: 7, y: 7, z: 5]) =$

$[x: 7, y: 5, z: 5]$

Заметим, что мы манипулировали только математическими объектами и не касались выполнения программы. Разница может показаться незначительной для программы содержащей только операторы присваивания и сотавные операторы, но для программ содержащих более сложные конструкции она будет существенной. Польза денотационного подхода обусловлена в основном тем фактом, что он абстрагируется от того как выполняется программа. Таким образом, становится легче рассуждать о программах просто как о некотором количестве математических объектов. Однако, необходимым условием для этого является установка твердой математической основы для денотационной семантики, а это не совсем тривиальная задача.

Денотационный подход может быть легко адаптирован к выражению другого вида свойств программ. Например:

- определить инициализированы ли все переменные перед использованием;
- определить является ли значение выражения в программе константой;
- определить, достижимы ли все части программы.

Мы предпочитаем денотационный подход когда рассуждаем о программе и операционный подход когда осуществляем реализацию языка. Таким

образом, очень интересным является вопрос о том эквивалентно ли денотационное определение операционному.

А к с и о м а т и ч е с к а я с е м а н т и к а.

Очень часто нас интересует частичная корректность программ. Программа называется частично корректной если из того факта, что программа, начав свою работу в состоянии удовлетворяющем предусловию, завершится следует что конечное состояние должно удовлетворять постусловию.

Для нашего примера:

$$\{ x = n \ \& \ y = m \}\ z := x\ ;\ x := y\ ;\ y := z\ \{x = m \ \& \ y = n\}$$

где $x = n \ \& \ y = m$ является предусловием, а $x = m \ \& \ y = n$ – постусловием. n и m используются для того, чтобы «запомнить» начальные значения x и y. Состояние [x**:** 5, y**:** 7, z**:** 0] удовлетворяет предусловию, принимая $n = 5$ и $m = 7$. Когда мы доказываем частичную корректность, мы должны будем доказать что, если программа завершится, то ее конечное состояние будет таким, что $x = 7$ и $y = 5$. Однако частичная корректность не гарантирует того, что программа завершится, хотя это достаточно ясно для нашего примера. Аксиоматическая семантика использует математическую логику для доказательства частичной корректности программ. Доказательство частичной корректности нашего примера может быть выражено следующим деревом вывода:

$$\dfrac{\dfrac{\{ p_0\}z := x\{ p_1\} \qquad \{ p_1\}x := y\{ p_2\}}{\{ p_0\}z := x;\ x := y\{ p_2\}} \qquad \{ p_2\}y := z\{ p_3\}}{\{ p_0\}z := x;\ x := y;\ y := z\{ p_3\}}$$

где

$p_0 = (x = n \ \& \ y = m)$

$p_1 = (z = n \ \& \ y = m)$

$p_2 = (z = n \ \& \ x = m)$

$p_3 = (x = m \ \& \ y = n)$

Мы можем рассматривать аксиоматический подход как спецификацию только некоторых аспектов семантики. Он не покрывает всех аспектов по той простой причине, что частичная корректность различных программ с точки зрения логики может быть совершенно идентичной, но мы не можем рассматривать такие программы как эквивалентные:

1. *{x = n & y = m} z := x; x := y; y :=z {x = m & y = n}*
2. *{x = n & y = m} if x = y then skip else (z := x; x := y; y :=z) {x = m & y = n}*
3. *{x = n & y = m} while true do skip {x = m & y = n}*

Польза аксиоматического подхода состоит в том, что логика предоставляет легкий способ доказательства свойств программ и, в известной степени, его можно автоматизировать.

Важно отметить, что эти виды семантик являются не конкурирующими, а различными подходами, предназначенными для различных целей и, в некоторой степени, для различных языков программирования.

1.2 Язык программирования WHILE.

Мы будем рассматривать различные подходы в семантике на примере очень простого императивного языка программирования. В качестве первого шага мы должны описать его синтаксис.

Выпишем сначала все синтаксические категории и метапеременные, значения которых будут принадлежать этим синтаксическим категориям.

n метапеременная для категории **Num** для чисел (целых);

x метапеременная для категории **Var** для переменных;

a метапеременная для категории **Aexp** для арифметических выражений;

b метапеременная для категории **Bexp** для булевых выражений;

S метапеременная для категории **Stm** для операторов.

Мы будем использовать индексы и штрихи для различения метапеременных принадлежащих одной и той же синтаксической категории.

Для простоты будем считать, что синтаксис чисел и переменных уже задан, например, число может быть строкой десятичных цифр, а переменная

строкой латинских букв и цифр начинающейся с буквы. Синтаксис остальных конструкций следующий:

$a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$

$b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$

$S ::= x := a \mid \text{skip} \mid S_1;S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S$

Семантика языка WHILE задается определением так называемых семантических функций для каждой синтаксической категории. Идея заключается в том, что семантическая функция использует синтаксическую конструкцию в качестве аргумента и возвращает ее смысл в качестве результата.

1.3 Семантика выражений.

Рассмотрим сначала числа в двоичной системе счисления.

$n ::= 0 \mid 1 \mid n\,0 \mid n\,1$

Чтобы определить значение представленное числом мы определим функцию

$\mathcal{N}: \mathbf{Num} \to \mathbf{Z}$

Она называется семантической функцией, так как она определяет семантику чисел. Если $n \in \mathbf{Num}$, то мы будем писать $\mathcal{N}[\![n]\!]$ для применения $\mathcal{N}$ к n. Мы будем использовать синтаксические скобки **[ ]** вместо более обычных ( ). Семантическая функция $\mathcal{N}$ определяется следующими уравнениями:

$\mathcal{N}[\![0]\!] = \mathbf{0}$

$\mathcal{N}[\![1]\!] = \mathbf{1}$

$\mathcal{N}[\![n\,0]\!] = \mathbf{2} \times \mathcal{N}[\![n]\!]$

$\mathcal{N}[\![n\,1]\!] = \mathbf{2} \times \mathcal{N}[\![n]\!] + \mathbf{1}$

Заметим, что нули и единицы стоящие в левых и правых частях уравнений различаются. В левых частях стоят синтаксические единицы, в то время как в правых частях используются целые числа $\mathbf{0}, \mathbf{1} \in \mathbf{Z}$, $\times$ и $+$ обычные арифметические операции над целыми числами. Приведенное нами

определение семантической функции является примером композиционного определения. Это значит, что для каждого из способов конструирования n оно указывает как соответствующее значение получается из смыслов (значений) его составляющих.

Пример.

$$\mathcal{N}[101] = \mathbf{2} \times \mathcal{N}[10] + \mathbf{1} =$$

$$= \mathbf{2} \times (\mathbf{2} \times \mathcal{N}[1]) + \mathbf{1} =$$

$$= \mathbf{2} \times (\mathbf{2} \times \mathbf{1}) + \mathbf{1} =$$

$$= \mathbf{5}$$

Заметим, что строка 101 разбиралась согласно синтаксису *n*.

Общую технику, которую мы применили в определении синтаксиса и семантики числа можно суммировать следующим:

Композиционное определение

1. В определении синтаксической категории выделяются базовые элементы и составные (композитные) элементы. Составные элементы единственным образом разлагаются на их непосредственно составляющие.
2. Семантика описывается композиционным определением функции: записывается семантическое уравнение для каждого из базовых элементов синтаксической категории; записывается семантическое уравнение для каждого метода конструирования составного элемента (т.е. количество семантических уравнений равно числу правил в определении синтаксиса). Семантика составных элементов определяется в терминах семантики непосредственно составляющих.

Технику доказательства тесно связанную с нашим подходом к определению семантических функций можно суммировать следующим:

Структурная индукция

1. Доказать, что некоторое утверждение имеет место для всех базовых элементов синтаксической категории.
2. Доказать, что это утверждение выполняется для всех составных элементов синтаксической категории: для этого предположить что это утверждение выполняется для всех непосредственных составляющих составного элемента (индукционная гипотеза), а затем доказать что из этого следует, что утверждение выполняется для самого элемента.

В дальнейшем мы будем считать, что все числа записаны в десятичной системе счисления и имеют обычный смысл (например, $\mathcal{N}$**[**137**] = 137** $\in$ **Z**).

Тем не менее, важно различать ЧИСЛО (что является синтаксической единицей) и ЗНАЧЕНИЕ ЧИСЛА (что является семантической единицей), даже в десятичной нотации.

Семантические функции.

Значение (смысл) выражения зависит от значений связанных с переменными участвующими в этом выражении. Например, если x связана с 3, то арифметическое выражение $x + 1$ имеет значение 4, но если x связана с 7, то значением выражения будет 8. Таким образом, мы приходим к определению понятия состояние. Каждой переменной состояние ставит в соответствие ее текущее значение. Мы будем представлять состояние как функцию из множества всех переменных в множество значений, т.е элемент множества

State = Var $\rightarrow$ **Z**

Каждое состояние *s* определяет некоторое значение (пишется *s x*), для каждой переменной $x \in$ **Var**. Так если *s x* = 3, то значением выражения $x + 1$ в состоянии *s* будет 4.

На самом деле это только один из нескольких способов представления состояния. Другими возможностями являются использование таблицы:

x	5
y	7

z	0

или списка вида [x**:** 5,y**:** 7,z**:** 0].

Во всех случаях мы должны быть уверенными в том, что с каждой переменной ассоциируется в точности одно значение.

Для заданного арифметического выражения a и состояния s мы можем определить значение выражения. Для этого мы определим смысл (значение) арифметического выражения как функцию $\mathcal{A}$, которая имеет два аргумента: синтаксическую конструкцию и состояние.

$$\mathcal{A} \textbf{ : Aexp} \rightarrow \textbf{(State} \rightarrow \textbf{Z)}$$

Такая запись означает, что $\mathcal{A}$ берет свои параметры по одному. То есть мы можем применить $\mathcal{A}$ к ее первому параметру, скажем $x + 1$ и получим в качестве результата функцию $\mathcal{A}$ **[**$x + 1$**]** функциональность которой определяется отображением **State** $\rightarrow$ **Z** и только когда мы применим ее к состоянию (которое тоже является функцией, но в данном случае это не имеет значения) мы получим значение выражения $x + 1$.

Предполагая существование функции $\mathcal{N}$определяющей значение чисел, мы можем определить функцию $\mathcal{A}$ задавая семантическое уравнение для каждой правой части правил для арифметического выражения a и состояния s.

$$\mathcal{A}\,[n]\,s = \mathcal{N}[n]$$

$$\mathcal{A}\,[x]\,s = s\,x$$

$$\mathcal{A}\,[a_1 + a_2]\,s = \mathcal{A}\,[a_1]\,s + \mathcal{A}\,[a_2]\,s$$

$$\mathcal{A}\,[a_1 * a_2]\,s = \mathcal{A}\,[a_1]\,s \times \mathcal{A}\,[a_2]\,s$$

$$\mathcal{A}\,[a_1 - a_2]\,s = \mathcal{A}\,[a_1]\,s - \mathcal{A}\,[a_2]\,s$$

Таблица 1.1 : Семантика арифметических выражений.

Уравнение для n отражает тот факт, что значение n в любом состоянии есть $\mathcal{N}[n]$. Значением переменной x в состоянии s является значение, связанное с x в s, т.е. $s\ x$. Значением составного выражения $a_1 + a_2$ в состоянии s является сумма значения a_1 в состоянии s и значения a_2 в состоянии s. Заметим, что +, ×, – участвующие в правых частях уравнений – это обычные арифметические операции, тогда как +, ∗, - в левых частях – синтаксические единицы.

Пример 1.5 Пусть $s\ x = 3$, тогда

$$\begin{aligned}\mathcal{A}[x+1]\ s &= \mathcal{A}[x]\ s + \mathcal{A}[1]\ s = \\ &= (s\ x) + \mathcal{N}[1] = \\ &= 3 + 1 = \\ &= 4\end{aligned}$$

Пример 1.6 Добавим арифметическое выражение $-a$ к нашему языку. Соответствующее семантическое уравнение будет

$$\mathcal{A}[-a]\ s = 0 - \mathcal{A}[a]\ s$$

тогда как альтернативное уравнение $\mathcal{A}[-a]\ s = \mathcal{A}[0 - a]\ s$ будет противоречить требованиям композиционного определения.

Значением булевых выражений являются истиностные значения, таким образом мы можем определить их значения функцией из множества состояний **State** в множество **T**:

$$\mathcal{B}: \mathbf{Bexp} \rightarrow (\mathbf{State} \rightarrow \mathbf{T})$$

Здесь **T** состоит из истиностных значений ***tt*** (для истины) и ***ff*** (для лжи).

Используя функцию $\mathcal{A}$, мы можем определить $\mathcal{B}$ как показано в таблице 1.2.

$\mathcal{B}$ [true] s	=	***tt***
$\mathcal{B}$ [false] s	=	***ff***
$\mathcal{B}[a_1 = a_2]\ s$	=	***tt***, если $\mathcal{A}[a_1]\ s = \mathcal{A}[a_2]\ s$

$$
\begin{array}{lcl}
 & & \textbf{\textit{ff}}, \text{ если } \mathcal{A}[a_1]\, s \neq \mathcal{A}[a_2]\, s \\
\mathcal{B}[a_1 \leq a_2]\, s & = & \begin{cases} \textbf{\textit{tt}}, \text{ если } \mathcal{A}[a_1]\, s \leq \mathcal{A}[a_2]\, s \\ \textbf{\textit{ff}}, \text{ если } \mathcal{A}[a_1]\, s > \mathcal{A}[a_2]\, s \end{cases} \\
\mathcal{B}[\neg b]\, s & = & \begin{cases} \textbf{\textit{tt}}, \text{ если } \mathcal{B}[b]\, s = \textbf{\textit{ff}} \\ \textbf{\textit{ff}}, \text{ если } \mathcal{B}[b]\, s = \textbf{\textit{tt}} \end{cases} \\
\mathcal{B}[b_1 \wedge b_2]\, s & = & \begin{cases} \textbf{\textit{tt}}, \text{ если } \mathcal{B}[b_1]\, s = \textbf{\textit{tt}} \text{ и } \mathcal{B}[b_2]\, s = \textbf{\textit{tt}} \\ \textbf{\textit{ff}}, \text{ если } \mathcal{B}[b_1]\, s = \textbf{\textit{ff}} \text{ или } \mathcal{B}[b_2]\, s = \textbf{\textit{ff}} \end{cases}
\end{array}
$$

Таблица 1.2: Семантика булевых выражений.

Упражнение 1.8. Пусть s $x = 3$. Вычислить $\mathcal{B}[\neg(x = 1)]\, s$.

Упражнение 1.10. Пусть синтаксическая категория **Bexp'** определяется как следующее расширение **Bexp**:

$$b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2 \mid a_1 < a_2 \mid a_1 > a_2 \mid \neg b \mid$$
$$\mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid b_1 \Rightarrow b_2 \mid b_1 \Leftarrow b_2$$

Дайте композиционное расширение семантической функции $\mathcal{B}$ из таблицы 1.2.

Два булевых выражения b_1 и b_2 являются эквивалентными если для любого состояния s, $\mathcal{B}[b_1]\, s = \mathcal{B}[b_2]\, s$.

Покажите что для любого $b' \in$ **Bexp'** существует булево выражение $b \in$ **Bexp** такое, что b' и b эквивалентны.

1.4 Свойства семантики выражений.

Далее нас будут интересовать два свойства выражений. Первое заключается в том, что значения выражений не зависят от значений переменных не участвующих в них. Второе состоит в том, что если мы заменим переменную подвыражением, то должны будем сделать соответствующее изменение в

состоянии. Мы формализуем эти свойства ниже и докажем что они имеют место.

Свободные переменные.

Свободные переменные арифметического выражения a можно определить как множество переменных участвующих в этом выражении.

Формально, мы можем дать композиционное определение множества $\mathrm{FV}(a) \subset \mathbf{Var}$ следующим образом:

$$\mathrm{FV}(n) = \varnothing$$

$$\mathrm{FV}(x) = \{x\}$$

$$\mathrm{FV}(a_1 + a_2) = \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)$$

$$\mathrm{FV}(a_1 * a_2) = \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)$$

$$\mathrm{FV}(a_1 - a_2) = \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)$$

Например, $\mathrm{FV}(x + 1) = \{x\}$ и $\mathrm{FV}(x + y * x) = \{x,y\}$.

Очевидно, что только переменные из $\mathrm{FV}(a)$ могут влиять на значение a. Формально это выражается следующей леммой.

Лемма 1.11. Пусть s и s' состояния такие, что $s\ x = s'\ x$ для всех $x \in \mathrm{FV}(a)$. Тогда $\mathcal{A}\ [\![a]\!]\ s\ =\ \mathcal{A}\ [\![a]\!]\ s'$.

Доказательство: Мы дадим достаточно детальное доказательство леммы используя структурную индукцию на арифметическом выражении.

Сначала базовые элементы **Aexp**:

n : Из таблицы 1.1 имеем $\mathcal{A}\ [\![n]\!]\ s = \mathcal{N}[\![n]\!]$ и $\mathcal{A}\ [\![n]\!]\ s' = \mathcal{N}[\![n]\!]$. Таким образом $\mathcal{A}\ [\![n]\!]\ s = \mathcal{A}\ [\![n]\!]\ s'$.

x : Из таблицы 1.1 имеем $\mathcal{A}\ [\![x]\!]\ s = s\ x$ $\mathcal{A}\ [\![x]\!]\ s' = s'\ x$. Из посылки леммы $s\ x = s'\ x$ так как $x \in \mathrm{FV}(a)$.

Отсюда $\mathcal{A}\ [\![x]\!]\ s = \mathcal{A}\ [\![x]\!]\ s'$.

Далее рассмотрим составные элементы **Aexp**:

$a_1 + a_2$ **:** Из таблицы 1.1 имеем $\mathcal{A}\,[\![a_1 + a_2]\!]\,s = \mathcal{A}\,[\![a_1]\!]\,s + \mathcal{A}\,[\![a_2]\!]\,s$ и $\mathcal{A}\,[\![a_1 + a_2]\!]\,s' = \mathcal{A}\,[\![a_1]\!]\,s' + \mathcal{A}\,[\![a_2]\!]\,s'$. Так как a_1 и a_2 являются непосредственными составляющими $a_1 + a_2$ и $\mathrm{FV}(a_1) \subseteq \mathrm{FV}(a_1 + a_2)$, $\mathrm{FV}(a_2) \subseteq \mathrm{FV}(a_1 + a_2)$ мы можем применить индукционную гипотезу к a_i и получить $\mathcal{A}\,[\![a_i]\!]\,s = \mathcal{A}\,[\![a_i]\!]\,s'$.

Отсюда легко видеть, что лемма имеет место для $a_1 + a_2$.

Случаи $a_1 - a_2$ и $a_1 * a_2$ аналогичны. Лемма доказана.

Подобно арифметическому выражению мы можем определить множество $\mathrm{FV}(b)$ свободных переменных в булевом выражении b.

$$\mathrm{FV}(\mathrm{true}) = \varnothing$$

$$\mathrm{FV}(\mathrm{false}) = \varnothing$$

$$\mathrm{FV}(a_1 = a_2) = \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)$$

$$\mathrm{FV}(a_1 \leq a_2) = \mathrm{FV}(a_1) \cup \mathrm{FV}(a_2)$$

$$\mathrm{FV}(\neg b) = \mathrm{FV}(b)$$

$$\mathrm{FV}(b_1 \wedge b_2) = \mathrm{FV}(b_1) \cup \mathrm{FV}(b_2)$$

Упражнение 1.12. Пусть s и s' состояния такие, что $s\,x = s'\,x$ для всех $x \in \mathrm{FV}(b)$. Докажите, что $\mathcal{B}\,[\![b]\!]\,s = \mathcal{B}\,[\![b]\!]\,s'$.

Подстановки.

Ниже нам будет интересен случай, когда каждое вхождение переменной y в арифметическое выражение a заменяется другим арифметическим выражением a_0. Будем называть это подстановкой и записывать $a[y\mathbf{:}\ a_0]$.

Формальное определение:

$$n[y\mathbf{:}\ a_0] = n$$

$$x[y\mathbf{:}\ a_0] = \begin{cases} a_0, \text{ если } x = y \\ x, \text{ если } x \neq y \end{cases}$$

$$(a_1 + a_2)[y\mathbf{:}\ a_0] = (a_1[y\mathbf{:}\ a_0]) + (a_2[y\mathbf{:}\ a_0])$$

$$(a_1 * a_2)[y\mathbf{:}\ a_0] = (a_1[y\mathbf{:}\ a_0]) * (a_2[y\mathbf{:}\ a_0])$$

$(a_1 - a_2)[y: a_0] = (a_1[y: a_0]) - (a_2[y: a_0])$

Например $(x + 1)[x: 3] = 3 + 1$

$(x + y * x)[x: y - 5] = (y - 5) + y * (y - 5)$.

Мы также будем записывать подстановки для состояний. Определим $s[y: v]$ как состояние равное s, за исключением того, что значение связанное с переменной y есть v, то есть

$$(s[y: v])x = \begin{cases} v, \text{ если } x = y \\ s\,x, \text{ если } x \neq y \end{cases}$$

Взаимосвязь между двумя этими понятиями показана в следующем упражнении.

Упражнение 1.13. Докажите, что $\mathcal{A}[\![a[y: a_0]]\!]\, s = \mathcal{A}[\![a]\!](s[y: \mathcal{A}[\![a_0]\!]\, s])$ для всех состояний s.

Глава 2. Операционная семантика.

Назначение операторов в языке WHILE состоит в том, чтобы изменять состояния. Например, если x связана с 3 в состоянии s и мы выполняем оператор $x := x + 1$, то мы получим новое состояние в котором x связана с 4. Таким образом, если семантики арифметических и булевых выражений только используют состояния для того, чтобы определить значение выражения, семантика операторов изменяет состояния.

Напомним, что в операционной семантике нам важно как выполняется программа, а не результат выполнения. Более того, нас интересует как состояния изменяются во время выполнения операторов. Мы рассмотрим два различных подхода к операционной семантике:

- натуральная семантика – ее целью является описание того, как достигается общий результат выполнения;
- структурная семантика – ее целью является описание отдельных шагов выполнения.

Мы увидим, что для языка WHILE можно легко определить обе разновидности семантики и что они эквивалентны.

Однако, мы рассмотрим также примеры конструкций для которых можно отдать явное предпочтение одному из подходов.

Для обеих операционных семантик значения операторов могут быть определены системой переходов. Она имеет два вида конфигураций:

- $\langle S,s\rangle$ означающей, что оператор S должен быть выполнен в состоянии s;
- s означающей терминальное (конечное) состояние.

Заключительной конфигурацией может быть только конфигурация второго вида. Отношение перехода описывает как происходит выполнение. Различие между двумя подходами к операционной семантике заключается в различных подходах к определению отношения перехода.

2.1 Натуральная семантика.

В натуральной семантике мы сосредотачиваемся на отношении между начальным и конечным состояниями выполнения. Таким образом, отношение перехода можно определить как отношение между начальным и конечным состояниями для каждого оператора.

Мы будем записывать переход как

$\langle S, s\rangle \rightarrow s'$.

Интуитивно это означает, что выполнение оператора S в состоянии s завершится и результирующим состоянием будет s'.

$[ass_{ns}]$	$\langle x := a, s\rangle \rightarrow s[x: \mathcal{A}[a]\ s]$	
$[skip_{ns}]$	$\langle \text{skip}, s\rangle \rightarrow s$	
$[comp_{ns}]$	$\dfrac{\langle S_1, s\rangle \rightarrow s', \langle S_2, s'\rangle \rightarrow s''}{\langle S_1; S_2, s\rangle \rightarrow s''}$	
$[if^{tt}_{ns}]$	$\dfrac{\langle S_1, s\rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \rightarrow s'}$	, если $\mathcal{B}[b]\ s = \mathbf{tt}$

$[\text{if}^{\text{ff}}_{\text{ns}}]$	$\dfrac{\langle S_2, s\rangle \to s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \to s'}$	, если $\mathcal{B}[b]\, s = \boldsymbol{ff}$
$[\text{while}^{\text{tt}}_{\text{ns}}]$	$\dfrac{\langle S, s\rangle \to s', \langle \text{while } b \text{ do } S, s'\rangle \to s''}{\langle \text{while } b \text{ do } S, s\rangle \to s''}$	, если $\mathcal{B}[b]\, s = \boldsymbol{tt}$
$[\text{while}^{\text{ff}}_{\text{ns}}]$	$\langle \text{while } b \text{ do } S, s\rangle \to s$	, если $\mathcal{B}[b]\, s = \boldsymbol{ff}$

Таблица 2.1: Натуральная семантика языка WHILE.

Правила для отношения переходов приведены в таблице 2.1. Правило имеет общий вид:

$$\frac{\langle S_1, s\rangle \to s_1, \ \dots\ , \langle S_{n-1}, s_{n-1}\rangle \to s_n, \langle S_n, s_n\rangle \to s}{\langle S, s\rangle \to s'} \text{ , если } \dots$$

где $S_1, \dots, S_n$ непосредственные составляющие S или операторы построенные из непосредственных составляющих S. Правило имеет некоторое количество посылок (записываются над чертой) и одно заключение (записывается под чертой). Правило в общем случае может также иметь условие (записывается справа от черты) при выполнении которого правило может применяться. Правила с пустым множеством посылок называются аксиомами и черта в них опускается.

Интуитивно, аксиома $[\text{ass}_{\text{ns}}]$ говорит, что выполнение оператора $x := a$ в состоянии s приводит к конечному состоянию $s[x: \mathcal{A}[a]\, s]$, которое отличается от s только тем, что x имеет значение $\mathcal{A}[a]\, s$. В действительности это ***схема аксиом***, так как x, a и s метапеременные замещающие действительные переменную, арифметическое выражение и состояние, но для простоты мы будем называть ее аксиомой. Мы получаем ***частный случай аксиомы*** выбирая конкретные переменную, арифметическое выражение и состояние. Например, если s_0 – состояние, которое задает значение 0 всем переменным, то

$$\langle x := x + 1, s_0 \rangle \rightarrow s_0[x\mathbf{:}\ 1]$$

является частным случаем [ass_{ns}] потому что x является частным случаем x, $x + 1$ – частный случай a, s_0 – частный случай s и значение $\mathcal{A}$ [$x + 1$] s_0 равно 1.

Аналогично, [skip_{ns}] является аксиомой и, интуитивно, говорит о том, что skip не изменяет состояние.

Правило [comp_{ns}] говорит, что для выполнения S_1; S_2 в состоянии s мы должны сначала выполнить S_1 в состоянии s. Предполагая, что это выполнение приведет к состоянию s', мы должны затем выполнить S_2 в состоянии s'. Посылки этого правила относятся к двум операторам S_1 и S_2, в то время как заключение выражает свойство составного оператора.

Частный случай этого правила:

$$\frac{\langle \text{skip}, s_0 \rangle \rightarrow s_0, \quad \langle x := x + 1, s_0 \rangle \rightarrow s_0[x\mathbf{:}\ 1]}{\langle \text{skip}; x := x + 1, s_0 \rangle \rightarrow s_0}$$

Здесь skip – частный случай S_1, $x := x + 1$ – частный случай S_2, s_0 – частный случай s и s', и $s_0[x\mathbf{:}\ 1]$ – частный случай s".

Другой пример:

$$\frac{\langle \text{skip}, s_0 \rangle \rightarrow s_0, \quad \langle x := x + 1, s_0 \rangle \rightarrow s_0[x\mathbf{:}\ 1]}{\langle \text{skip}; x := x + 1, s_0 \rangle \rightarrow s_0}$$

Это тоже частный случай правила [comp_{ns}], хотя и менее интересный, потому, что его посылки не выводятся ни из каких аксиом и правил таблицы 2.1.

Для if-конструкции мы имеем два правила. Первое, [$\text{if}^{\text{tt}}_{\text{ns}}$] говорит, что для выполнения оператора if b then S_1 else S_2 мы выполняем S_1 если значением b является ***tt*** в заданном состоянии.

Второе, [$\text{if}^{\text{ff}}_{\text{ns}}$] говорит, что если значением b является ***ff***, то для выполнения оператора if b then S_1 else S_2 мы должны выполнить оператор S_2.

Наконец, мы имеем одно правило и одну аксиому выражающие то, как выполнить конструкцию while.

Правило [while^{tt}_{ns}] говорит о том, что когда значением b является ***tt*** мы должны выполнить S, а затем снова выполнить while b do S.

Аксиома [while^{ff}_{ns}] говорит, что если значением b является ***ff***, то мы завершаем выполнение конструкции while оставляя состояние неизменным.

Заметим, что правило [while^{tt}_{ns}] определяет смысл конструкции while в терминах смысла той же самой конструкции и, таким образом, это определение не является композиционным.

Когда мы используем аксиомы и правила для того, чтобы вывести переход $\langle S, s\rangle \rightarrow s'$ мы строим дерево вывода. Корнем дерева вывода является $\langle S, s\rangle \rightarrow s'$, а листьями частные случаи аксиом. Внутренние узлы дерева являются заключениями частных случаев правил, и они имеют в качестве своих непосредственных потомков соответствующие посылки. Все условия частных случаев аксиом и правил должны удовлетворяться.

Дерево вывода называется простым, если оно является частным случаем аксиомы, иначе оно называется составным.

Рассмотрим теперь задачу построения дерева вывода для данных оператора S и состояния s. Наилучший способ достичь этого состоит в том, чтобы попытаться построить дерево из корня вверх. Мы начинаем с того, что ищем аксиому или правило с заключением левая часть которого совпадает с конфигурацией $\langle S, s\rangle$. Имеется две возможности:

- Если это аксиома и, если условие аксиомы удовлетворяется, то мы можем определить заключительное состояние и построение дерева вывода завершено.
- Если это правило, тогда следующим шагом будет попытка построить деревья вывода для посылок этого правила. Когда это сделано, нужно проверить выполнение условия этого правила – только в этом случае

мы можем определить конечное состояние соответствующее начальной кофигурации $\langle S, s\rangle$.

Часто бывает, что существует больше чем одна аксиома или правило, которые удовлетворяют заданной конфигурации и тогда различные возможности должны быть исследованы с целью найти дерево вывода. Мы увидим позже, что для языка WHILE существует по крайней мере одно дерево вывода для каждого перехода $\langle S, s\rangle \rightarrow s'$, но это не обязательно так для расширений языка WHILE.

Пример 2.2 Рассмотрим программу вычисляющую факториал:

$$y := 1; \text{ while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1)$$

и пусть s состояние в котором $s\ x = 3$. Мы покажем что

$$\langle y := 1; \text{ while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1), s\rangle \rightarrow s[y\mathbf{:}\ 6][x\mathbf{:}\ 1]. \quad (*)$$

Чтобы сделать это, мы должны показать что (*) может быть получена из системы переходов таблицы 2.1. Для этого нужно построить дерево вывода с переходом (*) в качестве корня.

Вместо того, чтобы строить сразу все дерево вывода T, мы будем строить его по частям. Вначале, нам известно только, что корень T имеет вид:

$$\langle y := 1; \text{ while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1), s\rangle \rightarrow s_{61}$$

Оператор $y := 1; \text{ while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1)$ имеет вид $S_1;S_2$, таким образом единственное правило применение которого приводит к корню T является $[\text{comp}_{\text{ns}}]$. Следовательно T должно иметь вид:

$$\frac{\langle y \mathbf{:=} 1, s\rangle \rightarrow s_{13} \qquad\qquad T_1}{\langle y := 1; \text{ while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1), s\rangle \rightarrow s_{61}}$$

для некоторого (пока не известного нам) состояния s_{13} и дерева вывода T_1, которое имеет корень:

$$\langle \text{while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1), s_{13}\rangle \rightarrow s_{61} \qquad (**)$$

Так как $\langle y := 1, s\rangle \to s_{13}$ является частным случаем аксиомы $[\text{ass}_{ns}]$ мы получаем, что $s_{13} = s[y: 1]$.

Отсутствующая часть T_1 дерева T – это дерево вывода с корнем (**). Так как оператор имеет вид while b do S дерево вывода T_1 должно быть простроено применением либо правила $[\text{while}^{tt}_{ns}]$, либо аксиомы $[\text{while}^{ff}_{ns}]$. Так как $\mathcal{B}[\![\neg(x = 1)]\!]\, s_{13} = \mathit{tt}$ мы видим, что только правило $[\text{while}^{tt}_{ns}]$ может быть применено и таким образом T_1 должно иметь вид:

$$\frac{T_2 \qquad\qquad T_3}{\langle \text{while } \neg(x = 1) \text{ do } (y := y * x;\ x := x - 1),\ s_{13}\rangle \to s_{61}}$$

где T_2 – дерево вывода с корнем $\langle y := y * x;\ x := x - 1,\ s_{13}\rangle \to s_{32}$, и T_3 – дерево вывода с корнем

$$\langle \text{while } \neg(x = 1) \text{ do } (y := y * x;\ x := x - 1),\ s_{32}\rangle \to s_{61} \qquad (***)$$

для некоторого состояния s_{32}.

Используя тот факт, что вид оператора $y := y * x;\ x := x - 1$ - $S_1;S_2$ легко видеть, что дерево вывода T_2:

$$\frac{\langle y := y * x,\ s_{13}\rangle \to s_{33} \qquad \langle x := x - 1,\ s_{33}\rangle \to s_{32}}{\langle y := y * x;\ x := x - 1,\ s_{13}\rangle \to s_{32}}$$

где $s_{33} = s[y: 3]$ и $s_{32} = s[y: 3][x: 2]$. Листьями T_2 являются частные случаи аксиомы $[\text{ass}_{ns}]$ и они объединяются используя $[\text{comp}_{ns}]$. Таким образом T_2 полностью построено.

Подобным образом мы можем построить дерево вывода T_3 с корнем (***):

$$\frac{\dfrac{\langle y := y * x,\ s_{32}\rangle \to s_{62} \qquad \langle x := x - 1,\ s_{62}\rangle \to s_{61}}{\langle y := y * x;\ x := x - 1,\ s_{32}\rangle \to s_{61}} \qquad T_4}{\langle \text{while } \neg(x = 1) \text{ do } (y := y * x;\ x := x - 1),\ s_{32}\rangle \to s_{61}}$$

где $s_{62} = s[y: 6][x: 2]$, $s_{61} = s[y: 6][x: 1]$ и T_4 имеет корень

$$\langle \text{while } \neg(x = 1) \text{ do } (y := y * x;\ x := x - 1),\ s_{61}\rangle \to s_{61}.$$

Мы видим, что T_4 является частным случаем аксиомы [while$^{ff}_{ns}$], потому что $\mathcal{B}[\neg(x = 1)]\ s_{61} = \mathit{ff}$.

Упражнение 2.3 Рассмотрим пример

$z := 0$; while $y \leq x$ do $(z := z + 1; x := x - y)$

Постройте дерево вывода для этого оператора в состоянии, в котором x имеет значение 17, а y – 5.

Введем следующие понятия. Будем говорить, что выполнение оператора S в состоянии s

- ***завершается*** если и только если имеется состояние s' такое, что $\langle S, s\rangle \rightarrow s$' и
- ***зацикливается*** если и только если не существует такого состояния s', что $\langle S, s\rangle \rightarrow s$'.

Мы будем говорить, что оператор S ***всегда завершается***, если его выполнение завершается для любого состояния s, и ***всегда зацикливается***, если его выполнение зацикливается для любого состояния s.

Упражнение 2.4 Для следующих операторов

- while $\neg(x = 1)$ do $(y := y * x; x := x - 1)$
- while $1 \leq x$ do $(y := y * x; x := x - 1)$
- while true do skip

определите не являются ли они всегда завершающимися или всегда зацикливающимися. Постарайтесь доказать это используя аксиомы и правила таблицы 2.1.

Семантические свойства.

Система переходов дает нам способ рассуждения об операторах и их свойствах. В качестве примера мы рассмотрим вопрос о том являются ли два оператора S_1 и S_2 семантически эквивалентными (под этим мы будем

понимать, что для любых состояний s и s' $\langle S_1, s\rangle \to s$' тогда и только тогда, когда $\langle S_2, s\rangle \to s$'.

Лемма 2.5 Оператор while *b* do *S* семантически эквивалентен оператору if *b* then (*S*; while *b* do *S*) else skip.

Доказательство.

Проведем его в два этапа. Сначала докажем что если

$$\langle \text{while } b \text{ do } S, s\rangle \to s'', \qquad (*)$$

то

$$\langle \text{ if } b \text{ then } (S; \text{ while } b \text{ do } S) \text{ else skip}, s\rangle \to s'' \qquad (**)$$

Таким образом, если выполнение цикла завершится, то завершится также его одноуровневая развертка. Позже покажем что если развернутый цикл завершится, то завершится и сам цикл. Объединение этих двух результатов и будет доказательством леммы.

Пусть теперь (*) выполняется, откуда должно существовать дерево вывода T. Оно может иметь два вида в зависимости от тог, используется ли при его построении правило $[\text{while}^{\text{tt}}_{\text{ns}}]$, или аксиома $[\text{while}^{\text{ff}}_{\text{ns}}]$. В первом случае дерево вывода T имеет вид:

$$\frac{T_1 \qquad T_2}{\langle \text{while } b \text{ do S}, s\rangle \to \text{s}''}$$

где T_1 дерево вывода с корнем $\langle S, s\rangle \to s$', а T_2 дерево вывода с корнем $\langle \text{while } b \text{ do } S, s'\rangle \to s''$. Кроме того $\mathcal{B}[\![b]\!]\, s = \boldsymbol{tt}$.

Используя деревья вывода T_1 и T_2 в качестве посылок для правила $[\text{comp}_{\text{ns}}]$ мы можем построить дерево вывода:

$$\frac{T_1 \qquad T_2}{\langle S; \text{ while } b \text{ do } S, s\rangle \to s''}$$

Учитывая, что $\mathcal{B}[\![b]\!]\, s = \boldsymbol{tt}$ мы можем использовать правило $[\text{if}^{\text{tt}}_{\text{ns}}]$ для построения дерева вывода:

$$\dfrac{\dfrac{T_1 \qquad T_2}{\langle S;\ \text{while } b \text{ do } S,\ s\rangle \to s''}}{\langle\ \text{if } b \text{ then } (S;\ \text{while } b \text{ do } S) \text{ else skip},\ s\rangle \to s''}$$

которое показывает, что (**) выполняется.

Во втором случае $\mathcal{B}[\![b]\!]\ s = \boldsymbol{ff}$ и $s'' = s$. Таким образом, T имеет вид

$\langle \text{while } b \text{ do } S,\ s\rangle \to s$.

Используя аксиому $[\text{skip}_{ns}]$ получим дерево вывода $\langle \text{skip},\ s\rangle \to s''$.

Теперь можно использовать правило $[\text{if}^{ff}_{ns}]$ для построения дерева вывода для (**):

$$\dfrac{\langle \text{skip},s\rangle \to s''}{\langle\ \text{if } b \text{ then } (S;\ \text{while } b \text{ do } S) \text{ else skip},\ s\rangle \to s''}$$

Это завершает первую часть нашего доказательства.

На втором этапе доказательства мы принимаем, что (**) выполняется и должны доказать, что (*) выполняется. То есть мы имеем дерево вывода T для (**) и должны построить дерево вывода для (*).

Только два правила могут использоваться при построении T, а именно $[\text{if}^{tt}_{ns}]$ и $[\text{if}^{ff}_{ns}]$.

В первом случае $\mathcal{B}[\![b]\!]\ s = \boldsymbol{tt}$ и мы имеем дерево вывода

$$\dfrac{T_1}{\langle\ \text{if } b \text{ then } (S;\ \text{while } b \text{ do } S) \text{ else skip},\ s\rangle \to s''}$$

где T_1 дерево вывода с корнем $\langle S;\ \text{while } b \text{ do } S,\ s\rangle \to s''$.

Оператор имеет вид $S_1; S_2$ и единственное правило которое может быть использовано это - $[\text{comp}_{ns}]$. Таким образом мы имеем поддеревья T_2 и T_3

$\langle S,\ s\rangle \to s'$ и $\langle \text{while } b \text{ do } S,\ s\rangle \to s''$

для некоторого состояния s'. Это прямо приводит нас к использованию правила $[\text{while}^{tt}_{ns}]$ для объединения T_2 и T_3 в дерево вывода для (*).

Во втором случае $\mathcal{B}[\![b]\!]\, s = \mathit{ff}$ и T строится используя правило $[\text{if}^{\text{ff}}_{\text{ns}}]$. Это означает, что мы имеем дерево вывода для $\langle \text{skip}, s\rangle \to s$" и согласно аксиоме $[\text{skip}_{\text{ns}}]$ должно быть $s = s$". Но тогда мы можем использовать аксиому $[\text{while}^{\text{ff}}_{\text{ns}}]$ при построении дерева вывода для (*). Доказательство завершено.

Упражнение 2.6 Докажите, что операторы $S_1;(S_2;S_3)$ и $(S_1;S_2);S_3$ семантически эквивалентны. Постройте оператор показывающий, что $S_1;S_2$ в общем случае не эквивалентен $S_2;S_1$.

Упражнение 2.7 Добавьте к языку WHILE оператор repeat S until b и определите отношение $\to$ для него. (Семантика оператора repeat не должна зависеть от существования оператора while в языке).

Докажите, что repeat S until b и S; if b then skip else (repeat S until b) семантически эквивалентны.

Упражнение 2.8 Добавьте к языку WHILE оператор for $x := a_1$ to a_2 do S и определите отношение $\to$ для него. Вычислите значение оператора

$y := 1$; for $z := 1$ to x do ($y := y * x$; $x := x - 1$)

в состоянии s таком, что $s\, x = 5$. (Семантика оператора for не должна зависеть от существования оператора while в языке).

В приведенном выше доказательстве леммы мы использовали таблицу 2.1 для исследования структуры дерева вывода когда некоторый переход имеет место. В доказательстве следующего результата мы объединим этот способ с индукцией по форме дерева вывода.

Индукция по форме дерева вывода.

1. Доказать, что некоторое утверждение имеет место для всех простых деревьев вывода. Для этого надо доказать, что оно имеет место для всех аксиом системы переходов.
2. Доказать, что утверждение имеет место для всех сложных (составных) деревьев вывода. Для каждого правила, предполагая что утверждение имеет место для посылок (индукционная гипотеза) доказать, что оно

имеет место для заключения правила в том случае, если условие применения правила выполняется.

Определение.

Будем говорить, что семантика детерминированная, если при любом выборе S, s, s', s'', из $\langle S, s\rangle \to s'$ и $\langle S, s\rangle \to s''$ следует, что $s' = s''$.

Это означает, что для любого оператора S и начального состояния s мы можем едиственным образом определить конечное состояние s' если (и только если) выполнение S завершится.

Теорема 2.9.

Натуральная семантика из таблицы 2.1 детерминированная.

Доказательство.

Пусть $\langle S, s\rangle \to s'$ и $\langle S, s\rangle \to s''$. Надо показать, что $s' = s''$. Докажем это методом индукции по форме дерева вывода для $\langle S, s\rangle \to s'$.

Случай $[\text{ass}_{\text{ns}}]$: $S - x := a$ и $s' - s[x\mathbf{:}\ \mathcal{A}\ [a]\ s]$. Единственной аксиомой или правилом для $\langle x := a,s\rangle \to s''$ может быть $[\text{ass}_{\text{ns}}]$, таким образом, s'' должно быть $s[x\mathbf{:}\ \mathcal{A}\ [a]\ s]$ и, следовательно $s' = s''$.

Случай $[\text{skip}_{\text{ns}}]$: Аналогично.

Случай $[\text{comp}_{\text{ns}}]$: Предположим, что $\langle S_1; S_2,s\rangle \to s'$ имеет место, потому что $\langle S_1, s\rangle \to s_0$ и $\langle S_2, s_0\rangle \to s'$ для некоторого s_0. Единственным правилом, которое можно применить чтобы получить $\langle S_1; S_2,s\rangle \to s''$ является $[\text{comp}_{\text{ns}}]$, таким образом, существеет состояние s_1 такое, что $\langle S_1, s\rangle \to s_1$ и $\langle S_2, s_1\rangle \to s''$.

Применяя индукционную гипотезу к посылкам $\langle S_1, s\rangle \to s_0$ и $\langle S_1, s\rangle \to s_1$, получим $s_0 = s_1$.

Подобным образом $\langle S_2, s_0\rangle \to s'$ и $\langle S_2, s_0\rangle \to s''$ дают $s' = s''$.

Случай $[\text{if}^{\text{tt}}_{\text{ns}}]$: Предположим, что $\langle$if b then S_1 else S_2, $s\rangle \to$ s' имеет место, потому что $\mathcal{B}\ [b]\ s = \mathbf{tt}$ и $\langle S_1, s\rangle \to s'$.

Из $\mathcal{B}[\![b]\!]\, s = \mathbf{tt}$ мы имеем, что единственное правило которое можно применить, чтобы получить $\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \rightarrow$ s'' – это $[\text{if}^{\text{tt}}_{\text{ns}}]$. В таком случае должно быть $\langle S_1, s\rangle \rightarrow s$''.

Применяя индукционную гипотезу к посылкам $\langle S_1, s\rangle \rightarrow s$'и $\langle S_1, s\rangle \rightarrow s$'', получим $s' = s$''.

Случай $[\text{if}^{\text{ff}}_{\text{ns}}]$: Аналогично.

Случай $[\text{while}^{\text{tt}}_{\text{ns}}]$: Предположим, что $\langle \text{while } b \text{ do } S, s\rangle \rightarrow s$' имеет место, потому что

$\mathcal{B}[\![b]\!]\, s = \mathbf{tt}$, $\langle S, s\rangle \rightarrow s_0$ и $\langle \text{while } b \text{ do } S, s_0\rangle \rightarrow s'$.

Единственным правилом, которое можно применить для получения $\langle \text{while } b \text{ do } S, s\rangle \rightarrow s$'' является $[\text{while}^{\text{tt}}_{\text{ns}}]$ потому, что $\mathcal{B}[\![b]\!]\, s = \mathbf{tt}$, а это значит что $\langle S, s\rangle \rightarrow s_1$ и $\langle \text{while } b \text{ do } S, s_1\rangle \rightarrow s$'' для некоторого s_1. Применяя индукционную гипотезу к посылкам $\langle S, s\rangle \rightarrow s_0$ и $\langle S, s\rangle \rightarrow s_1$, получим $s_0 = s_1$.

Таким образом, имеем $\langle \text{while } b \text{ do } S, s_0\rangle \rightarrow s$'и $\langle \text{while } b \text{ do } S, s_1\rangle \rightarrow s$''.

Так как они являются посылками (частных случаев) $[\text{while}^{\text{tt}}_{\text{ns}}]$ мы можем применить к ним индукционную гипотезу и получим $s' = s$''.

Случай $[\text{while}^{\text{ff}}_{\text{ns}}]$: Прямо следует из аксиомы.

Теорема доказана.

Упражнение 2.10* Докажите, что repeat S until $\underline{\text{b}}$ (как определено в упражнении 2.7) семантически эквивалентна S; while $\neg b$ do S. Покажите, что это означает, что рассширенная семантика детерминированная.

Важно отметить, что мы не можем доказать теорему 2.9 используя структурную индукцию. Причина в том, что правило $[\text{while}^{\text{tt}}_{\text{ns}}]$ определяет семантику while b do S в терминах самого себя. Структурная индукция работает отлично, когда семантика определяется композиционно (как например $\mathcal{A}$ и $\mathcal{B}$).

В своей основе индукция по форме деревьев вывода является одним из видов структурной индукции на деревьях вывода. В случае индукционной базы мы показываем, что утверждение имеет место для простых деревьев вывода. В случае индукционного шага мы предполагаем, что утверждение имеет место для непосредственных составляющих дерева вывода и показываем, что оно имеет место также для сложного (составного) дерева вывода.

Семантическая функция $\mathcal{S}_{ns}$.

Мы можем теперь определить смысл операторов как функцию (частичную) из **State** в **State**.

$\mathcal{S}_{ns}$ **: Stm** $\rightarrow$ (**State** $\rightarrow$ **State**).

Это значит, что для любого оператора S мы имеем частичную функцию

$\mathcal{S}_{ns}$ **[***S***]** $\in$ **State** $\rightarrow$ **State**

которая определяется следующим образом

$$\mathcal{S}_{ns}\,[\![S]\!]s \quad = \quad \begin{cases} s' \text{ если } \langle S, s\rangle \rightarrow s' \\ \underline{\text{undef}} \text{ в противном случае} \end{cases}$$

То, что это действительно частичная функция можно показать на примере оператора while true do skip, который всегда зацикливается (см. упражнение 2.4):

$\mathcal{S}_{ns}$ **[**while true do skip**]***s* = undef для всех состояний *s*.

Упражнение 2.11. Семантика арифметических выражений задается функцией $\mathcal{A}$. Мы можем также использовать операционный подход и определить натуральную семантику для арифметических выражений. Она имеет два вида конфигураций:

$\langle a, s\rangle$ обозначающая, что *a* должно быть вычислено в состоянии *s*, и

z обозначающая конечное значение (элемент **Z**).

Отношение перехода $\rightarrow_{\text{Aexp}}$ имеет вид

$\langle a, s\rangle \rightarrow_{\text{Aexp}} z$

И говорит о том, что значением a в состоянии s является z.

Примеры аксиом и правил:

$$\langle a, s\rangle \to_{\text{Aexp}} \mathcal{N}[n]$$

$$\langle x, s\rangle \to_{\text{Aexp}} s\ x$$

$$\frac{\langle a_1, s\rangle \to_{\text{Aexp}} z_1, \langle a_2, s\rangle \to_{\text{Aexp}} z_2}{\langle a_1 + a_2, s\rangle \to_{\text{Aexp}} z}$$, где $z = z_1 + z_2$

Закончите определение системы переходов. Используйте структурную индукцию по **Aexp** для доказательства того, что значение выражения a определенное этим отношением совпадает с тем, которое определяется $\mathcal{A}$.

Упражнение 2.12. Подобным образом мы можем определить натуральную семантику для булевых выражений. Переходы будут иметь вид $\langle b, s\rangle \to_{\text{Bexp}} t$, где $t \in \mathbf{T}$.

Определите систему переходов и докажите, что значение b определенное таким образом совпадает с тем, которое определяется $\mathcal{B}$.

Упражнение 2. 13. Выясните являются ли семантически эквивалентными операторы S_1 и S_2, если $\mathcal{S}_{ns}[S_1] = \mathcal{S}_{ns}[S_2]$.

2.2 Структурная операционная семантика.

В структурной операционной семантике особое значение уделяется индивидуальным шагам выполнения, таким как присваивание и проверка условия. Отношене перехода имеет вид

$$\langle S, s\rangle \Rightarrow \gamma$$

где γ имеет вид или $\langle S, s\rangle$, или s'. Переход показывает первый шаг выполнения S в состоянии s. Могут быть две возможности:

- если γ имеет вид $\langle S', s'\rangle$, тогда выполнение S в состоянии s не завершено и оставшееся выполнение выражается конфигурацией $\langle S', s'\rangle$.

- если γ имеет вид s', то выполнение S в состоянии s завершено, и заключительным состоянием является s'.

Мы будем говорить, что конфигурация $\langle S, s\rangle$ тупиковая, если не существует γ такого, что $\langle S, s\rangle \Rightarrow \gamma$.

Определение отношения $\Rightarrow$ задается аксиомами и правилами таблицы 2.2.

$[ass_{sos}]$	$\langle x := a, s\rangle \Rightarrow s[x: \mathcal{A}[a]\, s]$	
$[skip_{sos}]$	$\langle skip, s\rangle \Rightarrow s$	
$[comp^1_{sos}]$	$\dfrac{\langle S_1, s\rangle \Rightarrow \langle S'_1, s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S'_1; S_2, s'\rangle}$	
$[comp^2_{sos}]$	$\dfrac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$	
$[if^{tt}_{sos}]$	$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \Rightarrow \langle S_1, s\rangle$	, если $\mathcal{B}[b]\, s = \boldsymbol{tt}$
$[if^{ff}_{sos}]$	$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \Rightarrow \langle S_2, s\rangle$	, если $\mathcal{B}[b]\, s = \boldsymbol{ff}$
$[while_{sos}]$	$\langle \text{while } b \text{ do } S, s\rangle \Rightarrow \langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, s\rangle$	

Таблица 2.2 Структурная операционная семантика языка WHILE.

Аксиомы $[ass_{sos}]$ и $[skip_{sos}]$ не изменились, потому что операторы присваивания и skip полностью выполняются за один шаг.

Правила $[comp^1_{sos}]$ и $[comp^2_{sos}]$ выражают тот факт, что для выполнения $S_1; S_2$ в состоянии s мы сначала выполняем один шаг S_1 в состоянии s. Далее существуют две возможности:

- Если выполнение S_1 не завершено, мы должны завершить его прежде чем переходить к выполнению S_2.
- Если выполнение S_1 завершено, мы можем начинать выполнение S_2.

Из аксиом $[if^{tt}_{sos}]$ и $[if^{ff}_{sos}]$ мы видим, что первый шаг в выполнении оператора if состоит в вычислении условия и выборке соответствующей ветки.

Наконец, аксиома $[while_{sos}]$ показывает, что первый шаг выполнения оператора while состоит в его развертке, то есть переписывании его как

условного оператора. Проверка условия будет выполнена на втором шаге (где будет применяться одна из аксиом if-конструкции).

Определение.

Последовательность вывода оператора S начинающегося в состоянии s – это либо

- конечная последовательность $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_k$ конфигураций таких, что $\gamma_0 = \langle S, s\rangle$, $\gamma_i \Rightarrow \gamma_{i+1}$ для $0 \le i < k$, $(k \ge 0)$, где γ_k либо конечная, либо тупиковая конфигурация; либо
- бесконечная последовательность $\gamma_0, \gamma_1, \gamma_2, \ldots$ конфигураций таких, что $\gamma_0 = \langle S, s\rangle$, $\gamma_i \Rightarrow \gamma_{i+1}$ для $i \ge 0$.

Мы будем писать $\gamma_0 \Rightarrow^i \gamma_i$ если существуют i шагов и $\gamma_0 \Rightarrow^* \gamma_i$ когда существует конечное число шагов.

Заметим, что $\gamma_0 \Rightarrow^i \gamma_i$ и $\gamma_0 \Rightarrow^* \gamma_i$ не обязательно являются последовательностями вывода: они будут ими тогда и только тогда, когда γ_i будет либо конечной, либо тупиковой конфигурацией.

Пример 2.15 Пусть s x = 3. Первый шаг выполнения в конфигурации

$$\langle y := 1;\ \text{while}\ \neg(x = 1)\ \text{do}\ (y := y * x;\ x := x - 1),\ s\rangle$$

даст конфигурацию

$$\langle \text{while}\ \neg(x = 1)\ \text{do}\ (y := y * x;\ x := x - 1),\ s[y: 1]\rangle$$

Мы получили это используя аксиому [ass_{sos}] и правило [$\text{comp}^2_{\text{sos}}$] как показывает следующее дерево вывода:

$$\frac{\langle y := 1, s\rangle \Rightarrow s[y: 1]}{\langle y := 1;\ \text{while}\ \neg(x = 1)\ \text{do}\ (y := y * x;\ x := x - 1),\ s\rangle \Rightarrow \langle \text{while}\ \neg(x = 1)\ \text{do}\ (y := y * x;\ x := x - 1),\ s[y: 1]\rangle}$$

Следующим шагом выполнения будет переписывание цикла в условный оператор используя аксиому [$\text{while}_{\text{sos}}$] и, таким образом мы получаем конфигурацию:

$$\langle \text{if}\ \neg(x = 1)\ \text{then}\ ((y := y * x;\ x := x - 1);$$

$$\text{while } \neg(x = 1) \text{ do } (y := y * x;\ x := x - 1)) \text{ else skip}, s[y: 1]\rangle$$

Следующий шаг – проверка условия и получение (согласно $[\text{if}^{\text{tt}}_{\text{sos}}]$) конфигурации:

$$\langle (y := y * x;\ x := x - 1);\ \text{while } \neg(x = 1) \text{ do } (y := y * x;\ x := x - 1), s[y: 1]\rangle$$

Мы можем далее использовать $[\text{ass}_{\text{sos}}]$, $[\text{comp}^2{}_{\text{sos}}]$ и $[\text{comp}^1{}_{\text{sos}}]$ для получения конфигурации:

$$\langle x := x - 1;\ \text{while } \neg(x = 1) \text{ do } (y := y * x;\ x := x - 1), s[y: 3]\rangle$$

как показывает следующее дерево вывода:

$$\frac{\dfrac{\langle y := y * x, s[y: 1]\rangle \Rightarrow s[y: 3]}{\langle y := y * x;\ x := x - 1, s[y: 1]\rangle \Rightarrow \langle x := x - 1, s[y: 3]\rangle}}{\langle (y := y * x;\ x := x - 1);\ \text{while } \neg(x = 1) \text{ do } (y := y * x;\ x := x - 1), s[y: 1]\rangle \Rightarrow \langle x := x - 1;\ \text{while } \neg(x = 1) \text{ do } (y := y * x;\ x := x - 1), s[y: 3]\rangle}$$

Используя $[\text{ass}_{\text{sos}}]$ и $[\text{comp}^2{}_{\text{sos}}]$ получим следующую конфигурацию:

$\langle \text{while } \neg(x = 1) \text{ do } (y := y * x;\ x := x - 1), s[y: 3][x: 2]\rangle$

Продолжая, мы в конце концов получим конечное состояние $s[y: 6][x: 1]$.

Упражнение 2. 16 Постройте последовательность вывода для оператора

$$z := 0;\ \text{while } y \le x \text{ do } (z := z + 1;\ x := x - y)$$

в состоянии в котором x имеет значение 17 и y – 5.

Для заданных оператора S языка WHILE и состояния s всегда возможно найти как минимум одну последовательность вывода, которая начинается с конфигурации $\langle S, s\rangle$ просто применяя аксиомы и правила до тех пор, пока не будет достигнута заключительная или тупиковая конфигурация.

Из таблицы 2.2 ясно, что в языке WHILE нет тупиковых конфигураций, а упражнение 2.22 ниже показывает, что на самом деле существует только одна последовательность вывода начинающаяся с $\langle S, s\rangle$. Некоторые конструкции расширяющие WHILE рассмотренные в параграфе 2.4 могут

иметь тупиковые конфигурации или более чем одну последовательность вывода, которая начинается в заданной конфигурации.

По аналогии с терминологией предыдущего параграфа мы будем говорить, что выполнение оператора S в состоянии s

- ***завершается***, тогда и только тогда, когда существует конечная последовательность вывода начинающаяся с $\langle S, s\rangle$ и
- ***зацикливается***, тогда и только тогда, когда существует бесконечная последовательность вывода начинающаяся с $\langle S, s\rangle$.

Мы будем говорить, что выполнение S в состоянии s ***завершается успешно***, если $\langle S, s\rangle \Rightarrow^* s'$ для некоторого состояния s'.

В WHILE выполнение завершается успешно, если оно завершается, так как не существует тупиковых конфигураций.

Наконец, мы будем говорить, что оператор S ***всегда завершается***, если он завершается для любого состояния и ***всегда зацикливается***, если он зацикливается для любого состояния.

Упражнение 2. 17 Добавьте в WHILE конструкцию repeat S until b и опишите структурную операционную семантику для него (семантика для repeat-конструкции не должна зависеть от существования while-конструкции).

Упражнение 2. 18 Добавьте в WHILE конструкцию for $x := a_1$ to a_2 do S и опишите структурную операционную семантику для него (семантика для for-конструкции не должна зависеть от существования while-конструкции).

Семантические свойства.

Для структурной операционной семантики часто бывает удобным проводить доказательства методом индукции по длине последовательности вывода:

1. Доказать, что некоторое утверждение имеет место для любой последовательности вывода длиной 0.
2. Доказать, что это утверждение имеет место для любой другой последовательности вывода: предположить, что утверждение имеет место для всех последовательностей вывода длины меньше или равной

k (индукционная гипотеза) и показать, что оно имеет место для последовательности вывода длины $k + 1$.

Лемма 2. 19 Пусть $\langle S_1; S_2, s\rangle \Rightarrow^k s''$, тогда существует состояние s' и натуральные числа k_1 и k_2 такие, что $\langle S_1, s\rangle \Rightarrow^{k1} s'$ и $\langle S_2, s'\rangle \Rightarrow^{k2} s$ и $k = k_1 + k_2$.

Интуитивно, лемма выражает тот факт, что выполнение составного оператора распадается на две части, одна – выполнение S_1, другая – S_2.

Будем говорить, что семантика детерминированная, если для любых S, s, γ, γ' из $\langle S, s\rangle \Rightarrow \gamma$ и $\langle S, s\rangle \Rightarrow \gamma'$ следует, что $\gamma = \gamma'$.

Структурная операционная семантика из таблицы 2.2 детерминированная.

Операторы S_1 и S_2 семантически эквивалентны, если для любого состояния s

- $\langle S_1, s\rangle \Rightarrow^* \gamma$ тогда и только тогда, когда $\langle S_2, s\rangle \Rightarrow^* \gamma$, где γ - либо конечная, либо тупиковая конфигурация
- последовательность вывода начинающаяся в $\langle S_1, s\rangle$ бесконечна тогда и только тогда, когда бесконечна последовательность вывода начинающаяся в $\langle S_2, s\rangle$.

Заметим, что в первом случае длины последовательностей вывода могут различаться.

Семантическая функция $\mathcal{S}_{sos}$.

$\mathcal{S}_{sos}$ **: Stm** $\rightarrow$ (**State** $\rightarrow$ **State**).

То есть, для любого оператора S

$$\mathcal{S}_{sos}\,[S]s = \begin{cases} s' \text{ если } \langle S, s\rangle \Rightarrow^* s' \\ \underline{\text{undef}} \text{ в противном случае} \end{cases}$$

2.3 Эквивалентность операционных семантик.

Теорема 2. 26 Для любого оператора S языка WHILE имеем $\mathcal{S}_{ns}$ **[**S**] =** $\mathcal{S}_{sos}$ **[**S**].**

Это утверждение выражает два свойства:

- Если выполнение S начавшееся в каком-либо состоянии завершится в одной семантике, то оно завершится и в другой и, конечные состояния будут равны.
- Если выполнение S начавшееся в каком-либо состоянии зациклится в одной семантике, то оно зациклится и в другой.

2.4 Расширения языка WHILE.

Для иллюстрации силы и слабости двух подходов к операционной семантике рассмотрим различные расширения языка WHILE.

Оператор abort.

Оператор abort останавливает выполнение всей программы. Это означает, что поведение abort отличается от поведения while true do skip, так же как и от skip, так как оператор следующий за abort никогда не будет выполнен, тогда как оператор следующий за skip всегда будет выполнен.

Новый синтаксис:

$S ::= x := a \mid \text{skip} \mid S_1;S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \mid \text{abort}$

В структурной операционной семантике конфигурация $\langle \text{abort}, s \rangle$ - тупиковая, таким образом, мы добавим аксиому $\langle \text{abort}, s \rangle$.

В натуральной семантике ясно, что skip и abort не эквивалентны. Но while true do skip и abort являются семантически эквивалентными!

Причина в том, что в натуральной семантике мы имеем дело с выполнениями, которые завершаются. То есть, если мы не можем построить дерево вывода для $\langle S,s \rangle \rightarrow$ s', то мы не можем говорить почему: из-за того ли, что мы пришли к тупиковой конфигурации, или из-за зацикливания.

Упражнение 2. 32 Добавить в WHILE оператор assert b before S. Если b = true, то выполнить S.

Недетерминированность.

$S ::= S_1 \text{ or } S_2$

Идея состоит в том, что мы недетерминированно выбираем на выполнение либо S_1, либо S_2.

Таким образом, мы ожидаем, что выполнение оператора

$x := 1$ or $(x := 2; x := x + 2)$

будет иметь результирующее состояние в котором x имеет значение либо 1, либо 4.

В описание натуральной семантики мы мы добавим два правила:

$$[\text{or}^1_{\text{ns}}] \qquad \frac{\langle S_1, s\rangle \to s'}{\langle S_1 \text{ or } S_2, s\rangle \to s'}$$

$$[\text{or}^2_{\text{ns}}] \qquad \frac{\langle S_2, s\rangle \to s'}{\langle S_1 \text{ or } S_2, s\rangle \to s'}$$

Соответственно, для конфигурации $\langle x := 1 \text{ or } (x := 2; x := x + 2), s\rangle$ имеем одно дерево вывода с корнем

$\langle x := 1 \text{ or } (x := 2; x := x + 2), s\rangle \to s[x\mathbf{:}\ 1]$

и второе дерево вывода с корнем

$\langle x := 1 \text{ or } (x := 2; x := x + 2), s\rangle \to s[x\mathbf{:}\ 4]$

Важно отметить что, если мы заменим в примере $x := 1$ на while true do skip, то будем иметь только одно дерево вывода с корнем

$\langle \text{while true do skip or } (x := 2; x := x + 2), s\rangle \to s[x\mathbf{:}\ 4]$

В описание структурной операционной семантики добавим две аксиомы:

$[\text{or}^1_{\text{sos}}] \qquad \langle S_1 \text{ or } S_2, s\rangle \Rightarrow \langle S_1, s\rangle$

$[\text{or}^2_{\text{sos}}] \qquad \langle S_1 \text{ or } S_2, s\rangle \Rightarrow \langle S_2, s\rangle$

Для нашего примера мы имеем две последовательности вывода:

$\langle x := 1 \text{ or } (x := 2; x := x + 2), s\rangle \Rightarrow^* s[x\mathbf{:}\ 1]$

и

$\langle x := 1 \text{ or } (x := 2; x := x + 2), s\rangle \Rightarrow^* s[x\mathbf{:}\ 4]$

Если мы заменим в примере $x := 1$ на while true do skip, то будем иметь тоже две последовательности вывода.

Одну бесконечную:

$\langle$while true do skip or (x := 2; x := x + 2), $s\rangle \Rightarrow$

$\langle$while true do skip, $s\rangle \Rightarrow$

$\langle$while true do skip, $s\rangle \Rightarrow$

…

Другую конечную:

$\langle$while true do skip or (x := 2; x := x + 2), $s\rangle \Rightarrow^* s[x\mathbf{:}\ 4]$

Сравнивая два подхода к операционной семантике, мы видим, что структурная операционная семантика может выбрать «не верную» ветку or-оператора. В то время, как натуральная семантика всегда выбирает «верную» ветку.

Упражнение 2.34 Добавьте в язык WHILE оператор random(x), где x любое натуральное число.

Параллелизм.

$S ::= S_1$ par S_2

Идея заключается в том, что выполнение расслаивается.

x := 1 par (x := 2; x := x + 2)

может иметь три различных результата для x – 4, 1 и 3.

1) x := 1	2) x := 2; x := x + 2	4
1) x := 2; x := x + 2	2) x := 1	1
1) x := 2	2) x := 1 3) x := x + 2	3

В описание структурной операционной семантики добавим следующие правила:

$$[\text{par}^1_{\text{sos}}] \quad \frac{\langle S_1, s\rangle \Rightarrow \langle S'_1, s'\rangle}{\langle S_1 \text{ par } S_2, s\rangle \Rightarrow \langle S'_1 \text{ par } S_2, s'\rangle}$$

$$[\text{par}^2_{\text{sos}}] \quad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1 \text{ par } S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

$$[\text{par}^3_{\text{sos}}] \quad \frac{\langle S_2, s\rangle \Rightarrow \langle S'_2, s'\rangle}{}$$

$$\frac{}{\langle S_1 \text{ par } S_2, s\rangle \Rightarrow \langle S_1 \text{ par } S'_2, s'\rangle}$$

$$[\text{par}^4_{\text{sos}}] \qquad \frac{\langle S_2, s\rangle \Rightarrow s'}{\langle S_1 \text{ par } S_2, s\rangle \Rightarrow \langle S_1, s'\rangle}$$

В описание натуральной семантики мы могли бы добавить два правила:

$$\frac{\langle S_1, s\rangle \rightarrow s',\ \langle S_2, s'\rangle \rightarrow s''}{\langle S_1 \text{ par } S_2, s\rangle \rightarrow s''}$$

$$\frac{\langle S_1, s\rangle \rightarrow s',\ \langle S_1, s'\rangle \rightarrow s''}{\langle S_1 \text{ par } S_2, s\rangle \rightarrow s''}$$

Однако это работать не будет, так как правила выражают тот факт, что или S_1 выполняется перед S_2, или наоборот. То есть, мы теряем возможность расслаивать выполнение двух операторов.

Упражнение 2.35 Добавьте вместе с par-оператором оператор protect *S* end. В котором *S* должен выполняться как единый оператор.

Например,

x :=1 par protect ($x := z$; $x := x + 2$) end

имеет только два возможных результата – 1 и 4.

2.5 Блоки и процедуры.

Расширим язык WHILE блоками содержащими объявления переменных и процедурами.

До этого мы должны рассмотреть пару важных концепций:

- окружение переменной и процедуры, и
- расположение и память.

Мы будем рассматривать натуральную семантику для описания динамической и статической области видимости и не рекурсивных и рекурсивных процедур.

Блоки и простые объявления.

Добавим я язык WHILE сначала блоки содержащие объявления локальных переменных. Новый язык BLOCK имеет следующий синтаксис:

S ::= x := a | skip | S_1;S_2 | if b then S_1 else S_2 | while b do S |

| begin D_v S end

где D_v метапеременная из синтаксической категории $\mathrm{Dec_v}$ объявления переменных.

D_v ::= var x := a; D_v | ε

где ε пустое объявление. Идея заключается в том, что переменные объявленные внутри блока begin D_v S end являются локальными для него. Таким образом, в операторе

begin var y := 1;
(x := 1;
begin var x := 2; y := x + 1 end;
x := y + x)
end

x в y := x + 1 является локальной переменной объявленной var x := 2, в то время как x в x := y + x является глобальной переменной которая используется также в операторе x := 1. В обоих случаях переменная y является глобальной (то есть объявленной во внешнем блоке). Таким образом, оператор y := x + 1 присваивает у значение 3, а не 2, и оператор x := y + x присваивает x значение 4, а не 5.

Прежде чем переходить к деталям описания семантики, определим множество DV(D_v) переменных объявленных в D_v:

DV(var x := a; D_v) = $\{x\} \cup \mathrm{DV}(D_v)$

DV(ε) = $\varnothing$

Наконец, мы можем описать натуральную семантику. Идея будет состоять в том, что у нас будет по своей системе переходов для каждой синтаксической категории **Stm** и $\mathbf{Dec_v}$. Для операторов система переходов будет такой же как в таблице 2.1 с добавлением правила из таблицы 2.3.

Глава 3. Реализация с доказательством корректности.

Формальная спецификация семантики будет полезной при реализации языка программирования. В частности становится возможным рассуждение о корректности реализации. Мы проиллюстрируем это на примере перевода языка WHILE в структурную форму ассемблерного кода абстрактной машины, и докажем, что этот перевод корректен. Идея заключается в том, что сначала мыопределяем смысл (значение) инструкций абстрактной машины как операционную семантику, затем определяем функции перевода, которые отображают выражения и операторы языка WHILE в последовательность таких инструкций. Корректность реализации будет состоять в том, что если мы:

- переводим программу в код, и
- выполняем код на абстрактной машине,

то мы получим результат, не отличающийся от результата специфицируемого функциями и предыдущей главы.

3.1 Абстрактная машина.

При описании абстрактной машины удобно сначала определить ее конфигурации, а затем уже описывать инструкции и их семантику. Такт работы абстрактной машины можно рассматривать как переход от одной конфигурации к другой. Здесь конфигурация – это мгновенное состояние абстрактной машины.

Абстрактная машина будет иметь конфигурации вида $\langle c, e, s \rangle$, где

- c – последовательность еще не выполненных инструкций (кода),
- e – стек для вычислений,
- s – память.

Мы используем стек для вычисления арифметических и булевых выражений. Формально, это список значений, поэтому если **Stack** = (**Z** ∪ **T**)*, то $e \in$ **Stack**.

Чтобы упростить дальнейшие объяснения мы примем, что память подобна состоянию, то есть $s \in$ **State** и используется для хранения значений переменных.

Синтаксис инструкций абстрактной машины (AM) задается следующей грамматикой

inst ::= PUSH-*n* | ADD | MULT | SUB
| TRUE | FALSE | EQ | LE | AND | NEG
| FETCH-*x* | STORE-*x*
| NOOP | BRANGH(*c*, *c*) | LOOP(*c*, *c*)

c ::= ε | *inst***:***c*

где ε - пустая цепочка.

Мы будем использовать **Code** для обозначения синтаксической категории последовательностей инструкций, таким образом, *c* метапеременная из **Code**. Имеем

$\langle c, e, s\rangle \in$ **Code** × **Stack** × **State**

Будем называть конфигурацию терминальной (или конечной), если ее код пустой - $\langle \varepsilon, e, s\rangle$.

Семантику инструкций абстрактной машины зададим операционной семантикой. Также как в предыдущей главе она будет описываться системой переходов. Отношение перехода заданное на множестве конфигураций описывает выполнение инструкций:

$\langle c, e, s\rangle \blacktriangleright \langle c', e', s'\rangle$

то есть, один шаг выполнения преобразует конфигурацию $\langle c, e, s\rangle$ в $\langle c', e', s'\rangle$. Отношение определяется аксиомами таблицы 3.1, в которой мы используем обозначение '**:**' и для объединения двух последовательностей инструкций, и

для отделения элемента от последовательности. Стек для вычислений представляется как последовательность элементов. Вершина стека находится справа и, мы будем писать ε в качестве пустой последовательности.
В добавление к обычным арифметическим и булевым операциям мы имеем шесть инструкций, которые модифицируют стек вычислений: Инструкция PUSH-*n* вталкивает константное значение n в стек, TRUE и FALSE вталкивают константы **tt** и **ff**, соответственно, в стек. Инструкция FETCH-*x* вталкивает в стек значение связанное с переменной *x*. STORE-*x* выталкивает элемент из вершины стека и обновляет память так, что вытолкнутое значение связывается с переменной *x*. Две инструкции изменяют поток управления. Инструкция BRANCH(c_1, c_2) используется для реализации условия. Если в вершине стека находится значение **tt**, то оно выталкивается из стека и c_1 должно быть выполнено следующим. Иначе, если в вершине стека находится значение **ff**, то оно выталкивается из стека и c_2 должно быть выполнено следующим. Циклическая конструкция реализуется инструкцией LOOP(c1, c2). Семантика этой инструкции определяется переписыванием ее в комбинацию других инструкций включая инструкцию BRANCH и ее саму. Семантика из таблицы 3.1 является не чем иным как структурной операционной семантикой абстрактной машины AM.

⟨PUSH-*n*:*c*, *e*, *s*⟩	►
⟨ADD:*c*, z_1:z_2:*e*, *s*⟩	►
⟨MULT:*c*, z_1:z_2:*e*, *s*⟩	►
⟨SUB:*c*, z_1:z_2:*e*, *s*⟩	►
⟨TRUE:*c*, *e*, *s*⟩	►
⟨FALSE:*c*, *e*, *s*⟩	►
⟨EQ:*c*, z_1:z_2:*e*, *s*⟩	►
⟨LE:*c*, z_1:z_2:*e*, *s*⟩	►
⟨AND:*c*, t_1:t_2:*e*, *s*⟩	►

⟨NEG:c, t:e, s⟩	►	⟨c, **ff**:e, s⟩	
		⟨c, **tt**:e, s⟩	
⟨FETCH-x:c, e, s⟩	►	⟨c, (s x):e, s⟩	
⟨STORE-x:c, z:e, s⟩	►	⟨c, e, s[x : z]⟩	
⟨NOOP:c, e, s⟩	►	⟨c, e, s⟩	
⟨BRANCH(c_1, c_2):c, t:e, s⟩	►	⟨c_1:c, e, s⟩	если t = **tt**
		⟨c_2:c, e, s⟩	если t = **ff**
⟨LOOP(c_1, c_2):c, e, s⟩	►		