МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Нижегородский государственный университет им. Н.И. Лобачевского

Механико–математический факультет

Кафедра математической физики

"Бакалаврская работа"

Компилятор для грамматики выражений

Исполнитель:
студент гр.644-2
Силаев С.Н.

Научный руководитель:
к.ф-м.н. доцент
Гаврилов В.С.

Нижний Новгород
2014

Оглавление

Список таблиц

Глава 1

Общие сведения

1.1. Регулярные языки

Алфавитом называется любое конечное множество некоторых символов. Как правило, алфавит обозначается какой–нибудь заглавной греческой буквой.

Из символов алфавита можно естественным образом составлять **строки**. Если строка составлена из символов алфавита Σ, её называют **строкой над алфавитом** Σ. Длину строки x над алфавитом Σ, т.е. количество символов в строке x, будем обозначать $|x|$.

Пустая строка, т.е. строка, вообще не содержащая символов, обычно обозначается буквой ε. Длина этой строки равна нулю.

Множество всех строк алфавита Σ обозначается через Σ^*. Понятно, что над любым алфавитом можно составить бесконечно много строк. Тогда, **язык** — это любое подмножество множества строк над используемым алфавитом.

Поскольку язык — это некоторое множество, то нужно как–то уметь его описывать. Одним из инструментов такого описания являются **регулярные выражения**.

Регулярное выражение рекурсивно строится из меньших регулярных выражений с использованием описанных ниже правил. Каждое регулярное выражение r описывает язык $L(r)$, который также рекурсивно определяется на основании языков, описываемых подвыражениями выражения r. Вот правила, которые определяют регулярные выражения над некоторым алфавитом Σ, и языки, описываемые этими регулярными выражениями.

Базис:

1) ε язвляется регулярным выражением, а $L(\varepsilon)$ представляет собой ε;

2) если a — символ из Σ, то a представляет собой регулярное выражение, причем $L(a) = a$.

Индукция: Пусть r и s — регулярные выражения, описывающие языки $L(r)$ и $L(s)$ соответственно. Тогда

1) $(r)|(s)$ — регулярное выражение, описывающее язык $L(r) \cup L(s)$;

2) $(r)(s)$ — регулярное выражение, описывающее язык $L(r)L(s)$;

3) $(r)^*$ — регулярное выражение, описывающее язык $(L(r))^*$;

4) (r) — регулярное выражение, описывающее язык $L(r)$. Иными словами, выражение можно заключить в скобки без изменения описываемого им языка.

Регулярные выражения часто содержат лишние пары скобок. Эти скобки можно опустить, если принять следующие соглашения:

1) унарный оператор $*$ левоассоциативен и имеет наивысший приоритет;

2) конкатенация имеет второй по величине приоритет и также левоассоциативна;

3) оператор $|$ левоассоциативен и имеет наименьший приоритет.

Язык, который может быть определён регулярным выражением, называется **регулярным языком**.

Для удобства записи определённым регулярным выражениям можно присваивать имена и использовать эти имена в последующих выражениях так, как если бы это были символы. Если Σ является алфавитом базовых символов, то **регулярное определение** представляет собой последовательность определений вида

1) $d_1 \to r_1$
2) $d_2 \to r_2$

   ...

n) $d_n \to r_n$

Здесь

1) каждое d_i — новый символ, не входящий в Σ и не совпадающий ни с каким другим d_i
2) каждое r_i — регулярное выражение над алфавитом $\Sigma \cup d_1, \dots, d_{i-1}$

Пример 1.1.1.

Идентификаторы языка Си представляют собой строки из букв (латинских), цифр и символов подчеркивания. Идентификатор должен начинаться с буквы или подчеркивания. Опишем идентификаторы языка Си с помощью регулярных определений:

буква $\to A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|$
$|n|o|p|q|r|s|t|u|v|w|x|y|z$

буква_или_подчеркивание $\to$ буква$|_$

цифра $\to 0|1|2|3|4|5|6|7|8|9$

идентификатор $\to$ буква_или_подчеркивание(буква_или_подчеркивание|цифра)$*$

1.2. Сведения о грамматиках и языках

Порождающей грамматикой (или просто грамматикой) называется четвёрка $G(\Sigma_T, \Sigma_N, P, S)$, где

1) Σ_T — конечное множество **терминальных (основных) символов** — основной алфавит;
2) Σ_N — конечное множество **нетерминальных (вспомногательны) символов** — вспомогательный алфавит, $\Sigma_T \cap \Sigma_T = \varnothing$;
3) P — конечное множество **правил вывода**, называемых также **продукциями**; каждое правило имеет вид $\alpha \to \beta$, где $\alpha \in \Sigma^+$, $\beta \in \Sigma^*$ $(\Sigma \equiv \Sigma_T \cup \Sigma_T)$;
4) S — целевой (начальный) символ грамматики, $S \in \Sigma_N$.

Множество $\Sigma = \Sigma_T \cup \Sigma_T$ называется **полным алфавитом** грамматики G.

Элементами множества Σ_T являются символы, из которых в конечном счете и состоят цепочки языка порождаемого данной грамматикой. Терминальный (от лат. terminus — предел, конец) и означает "конечный, концевой". Множество Σ_T есть не что иное, как алфавит языка, порождаемого грамматикой. Терминальные символы, или просто **терминалы**, обозначаются малыми латинскими буквами $(a, b, c, d, \dots)$.

Нетерминальные символы, или, иначе, **нетерминалы**, — это понятия грамматики, используемые при описании языка. Нетерминалы обозначаются заглавными латинскими буквами $(A, B, C, D, E, \dots)$.

Начальный нетерминал S — это понятие, соответствующее правильному предложению языка.

Пример 1.2.1.

Рассмотрим грамматику
$G_1 = (\{a, b\}, \{S\}, \{S \to aSb, S \to \varepsilon\}, S)$
Здесь $\Sigma_T = \{a, b\}, \Sigma_N = \{S\}, P = \{S \to aSb, S \to \varepsilon\}$, начальным нетерминалом является S.

Чаще всего в явном виде четверку элементов грамматики не выписывают, а выписывают лишь её правила, причем нередко первым выписывают правила, в левую часть которых входит начальный нетерминал.

Например, G_1 в соответствии с этим соглашением можно переписать в виде

$S \to aSb$

$S \to \varepsilon$

Если в грамматике есть несколько правил с одной левой частью,

$\alpha \to \beta_1, \alpha \to \beta_2, \ldots, \alpha \to \beta_n,$

то принято объединять эти правила, записывая в виде

$\alpha \to \beta_1 | \beta_2 | \ldots | \beta_n,$

Такую форму правил грамматики называют **формой Бэкуса–Наура**.
Например, G_1 в форме Бэкуса–Наура можно записать так

$S \to aSb | \varepsilon$

Грамматика может использоваться для порождения (вывода) цепочек — предложений языка. Процесс порождения начинается с начального нетерминала — S. Если среди правил есть правило, в левой части которого записана цепочка S, то начальный нетерминал может быть заменен правой частью любого из таких правил. Оба правила грамматики G_1 содержат в левой части S. Применим подстановку, заданную первым правилом, заменив S на aSb:

$S \Rightarrow_1 aSb$

К получившейся цепочке aSb снова, если удастся, можно применить одно из правил грамматики. Если в цепочке есть подцепочка, совпадающая с левой частью хотя бы одного из правил, то эту подцепочку можно заменить правой часть любого из таких правил.

В цепочке aSb есть подцепочка S, совпадающая с левой частью обоих правил, грамматики G_1. Мы вправе применить любое из этих правил. Используем правило (1):

$S \Rightarrow_1 aSb \Rightarrow_1 aaSbb$

Применив затем правило (2), получим такую последовательность подстановок:

$S \Rightarrow_1 aSb \Rightarrow_1 aaSbb \Rightarrow_2 aabb$

Ясно, что теперь процесс порождения завершен.

Нетрудно заметить, что с помощью грамматики G_1 можно породить любую цепочку языка $L_1 = \{a^n b^n; n \geq 0\}$. В то же время, грамматика G_1 не порождает ни одной цепочки терминальных символов, не принадлежащей языку L. Иными словами, грамматики G_1 порождает язык L_1:

$L(G_1) = L_1$

Сентециальной формой грамматики G называется любая цепочка, выводимая из начального нетерминала грамматики G. Иными словами, цепочка α — сентенциальная форма грамматики G, если $S \Rightarrow_G^* \alpha$.

Сентенцией (от sentence — предложение) грамматики G называется сентенциальная форма, состоящая только из терминальных символов.

Язык, порождённый грамматикой, — это множество всех сентенций данной грамматики.

Можно сказать также, что сентенции грамматики – это предложения порождаемого ею языка.

1.2.1. Классификация грамматик

Формальные грамматики классифицируются по структуре их правил. Если все без исключения правила грамматики имеют некоторую заданную структуру, то грамматику относят к определённому типу.

Тип 0. Грамматики с фразовой структурой

На структуру их правил не накладываетя никаких ограничений: для грамматики $G(\Sigma_T, \Sigma_N, P, S)$, $\Sigma = \Sigma_T \cup \Sigma_N$, правила имеют вид $\alpha \to \beta$, где $\alpha \in \Sigma^+$, $\beta \in \Sigma^*$. Это самый общий тип грамматик и в него попадают все без исключения формальные грамматики. Однако некоторые из грамматик можно отнести и к другим классификационным типам.

Грамматики, которые относятся только к типу 0, и которые нельзя отнести ни к какому другому типу, — самые сложные по структуре и практического применения не имеют.

Тип 1. Контекстно–зависимые и неукорачивающие грамматики

В этот тип входят два основных класса грамматик.

Контекстно–зависимые $G(\Sigma_T, \Sigma_N, P, S)$, $\Sigma = \Sigma_T \cup \Sigma_N$, имеют правила вида $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, где $\alpha_1, \alpha_2 \in \Sigma^*, A \in \Sigma_N, \beta \in \Sigma^*$.

Неукорачивающие грамматики G(Σ_T, Σ_N,P,S), $\Sigma = \Sigma_T \cup \Sigma_N$, имеют правила вида $\alpha \to \beta$, где $\alpha, \beta \in \Sigma^+, |\beta| \geq |\alpha|$.

Структура правил КЗ–грамматик такова, что при построении предложений заданного ими языка один и тот же нетерминальный символ может быть заменён на ту или иную цепочку символов в зависимости от того контекста, в котором встречается. Цепочки α_1 и α_2 в правилах грамматики обозначают контекст, и в общем случае любая из них (или даже обе) может быть пустой. Иначе говоря, значение символа может быть различным в различных контекстах.

По правилам неукорачивающих грамматик, любая цепочка символов может быть заменена на цепочку символов не меньшей длины.

Доказано, что эти два класса грамматик эквивалентны. Это значит, что для любого языка, заданного контекстно–зависимой грамматикой, можно построить задающую тот же язык неукорачивающую грамматику и наоборот.

При построении компиляторов грамматики типа 1 не применяются, так как языки программирования имеют более простую структуру и могут быть построены с помощью грамматик других типов.

Тип 2. Контекстно–свободные грамматики

Такие грамматики $G(\Sigma_T, \Sigma_N, P, S)$, $\Sigma = \Sigma_T \cup \Sigma_N$, имеют правила вида $A \to \beta$, где $A \in \Sigma_N$, $\beta \in \Sigma^+$.

Поскольку из правила видно, что в правой части всегда должен находиться как минимум один символ, такие грамматики также иногда называют неукорачивающими контекстно-свободными (НКС) грамматиками.

Существует также почти эквивалентный им класс грамматик — укорачивающие контекстно-свободные грамматики $G(\Sigma_T, \Sigma_N, P, S)$, $\Sigma = \Sigma_T \cup \Sigma_N$, имеют правила вида $A \to \beta$, где $A \in \Sigma_N$, $\beta \in \Sigma^*$. Различие лишь в возможности наличия пустой цепочки символов в правых частях правил.

Синтаксис большинства известных языков программирования основан именно на контекстно–свободных грамматиках.

Тип 3. Регулярные грамматики

К типу регулярных относятся два эквивалентных класса грамматик.

Леволинейные грамматики $G(\Sigma_T, \Sigma_N, P, S)$, $\Sigma = \Sigma_T \cup \Sigma_N$, могут иметь лишь правила двух видов: либо $A \to B\gamma$, либо $A \to \gamma$, где $A, B \in \Sigma_N$, $\gamma \in \Sigma_T^*$.

Праволинейные грамматики $G(\Sigma_T, \Sigma_N, P, S)$, $\Sigma = \Sigma_T \cup \Sigma_N$, могут иметь лишь правила двух видов: либо $A \to \gamma B$, либо $A \to \gamma$, где $A, B \in \Sigma_N$, $\gamma \in \Sigma_T^*$.

Эти два класса грамматик эквивалентны.

1.2.2. Классификация языков

Языки классифицируются в соответствии с типами грамматик, с помсощью которых заданы. Причем один и тот же язык в общем случае может быть задан сколь угодно большим количеством грамматик, которые могут относиться к разным классификационным типам, то для классификации самого языка выбирается грамматика с максимально возможным классификационным типом.

От классификационного типа ящыка зависит не только то, с помощью какой грамматики можно построить предложение этого языка, но и то, насколько сложно распознать эти предложения.

Сложность языка убывает с возрастанием номера классификационного типа языка, так что языки типа 0 — самые сложные, а языки типа 3 — самые простые.

Согласно классификации грамматик, существует четыре типа.

Тип 0. Языки с фразовой структурой

Это самые сложные языки, которые могут быть заданы только грамматикой типа 0. Для распознавания цепочек таких языков требуются вычислители, равномощные машине Тьюринга. Поэтому, если язык относится к типу 0, то для него невозможно построить компилятор, который гарантированно выполнил бы разбор предложений языка за ограниченное время на основе ограниченных вычислительных ресурсов. Все естественные языки общения между людьми, строго говоря, относятся к этому типу языков. Дело в

том, что структура и значение фразы естественного языка может зависеть не только от контекста данной фразы, но и от содержания всего текста. Одно и то же слово в естественном языке может не только иметь разный смысл в зависимости от контекста, но и играть различную роль в предложении. Именно это и является причиной сложности автоматизации перевода текста с естественного языка, а также отсутствуют компиляторы, которые бы воспринимали программы на основе таких языков.

Тип 1. Контестно–зависимые языки

Этот тип второй по сложности тип языков. В общем случае время на распознавание предложений этого языка экспоненционально зависит от длины исходной цепочки символов.

Языки и грамматики типа 1 применяются при анализе и переводе текстов на естественных языках. Распознаватели, построенные на основе грамматик класса 1, позволяют анализировать тексты с учетом контекстной зависимости в предложениях входного языка, но не учитывают содержание текста, в силу чего, требуют вмешательство человека.

На основе таких грамматик могут выполняться автоматизированный перевод с одного естественного языка на другой или сервисные функции проверки провописания.

В компиляторах эти языки не используются, так как синтаксис языков программирования имеет более простую структуру.

Тип 2. Контекстно–свободные языки

Эти языки лежат в основе синтаксических конструкций большинства современных языков. На основе КС–языков функционируют некоторые довольно сложные командные процессоры, допускающие управляющие команды цикла и условия.

В общем случае время распознавания предложений языка типа 2 полиномиально зависит от длины входной цепочки символов. В зависимости от класса языков, это либо кубическая, либо квадратисная зависимость.

Однако среди КС–языков имеется много классов языков, для которых эта зависимость линейна. Многие языки программирования можно отнести к одному из таких классов.

Тип 3. Регулярные языки

Регулярные языки — самые простые, и потому являются наиболее широко распространенным и широко используемым типом языков. Время на распознавание предложений регулярного языка зависит от длины исходной цепочки символов.

Регулярные языки лежат в основе простейших конструкций языков программирования. На основе таких языков строятся примитивные ассемблеры, некоторые командные процессоры и другие подобные объекты.

1.3. SLR-1 синтаксический анализ

На данный момент наиболее распространенный тип восходящих синтаксических анализаторов, основан на концепции LR(k)- анализатора.

1) L – Просмотр входной строки слева направо
2) R – построение правого вывода в обратном порядке
3) k – Количество предпросматриваемых символов входного потока, нужных для принятия решения.

SLR-1 анализатор работает на основе таблицы синтаксического анализа, построенной по следующему алгоритму:

1) Строим набор множеств LR(0)- пунктов для расширенной грамматики G'
2) По I_i строим состояние i. Действие синтаксического анализатора определяется по правилам:
   (a) Если $[A \to \alpha \text{ffl} a\beta]I_i$, а ϵ Σ_T и ПЕРЕХОД(I_i ,) = I_j , то ДЕЙСТВИЕ[i, a] = “перенос j”
   (b) Если $[A \to \alpha \text{ffl}]I_i A! = S, A\Sigma_N$, то в ячейку ДЕЙСТВИЕ[i, a] записываем “свертка по $[A \to \alpha]$” для всех $aFOLLOW(A)$
(c) Если $[S' - > S\text{ffl}]I_i$, то ДЕЙСТВИЕ[I,] = ıj.

3) Переходы для состояния i для всех нетерминалов A строим так: Если ПЕРЕХОД$(I_i, A) = I_j$, то ПЕРЕХОД[i, A] = j

Правило грамматики G с точкой в некоторой позиции правой части.
Набор множества LR(0)- пунктов называется каноническим набором.

Замыкание множества пунктов

Если I – множество пунктов грамматики G, то ЗАМЫКАНИЕ(I) представляет собой множество пунктов построенных из I согласно двум правилам:

1) Добавляем в ЗАМЫКАНИЕ(I) все пункты из I

2) Если $[A \to \alpha \text{ffl} a\beta](I)$, A и B – нетерминалы, и имеется правило [$\to \gamma$] , то в ЗАМЫКАНИЕ(I) добавляется $[to\text{ffl}\gamma]$, если такого пункта ещё нет. Это делается до тех пора, когда нельзя будет добавить новый пункт.

Функция ПЕРЕХОД

Функция ПЕРЕХОД(I, X) определяется, как ЗАМЫКАНИЕ множества всех пунктов вида $[A \to \alpha X \text{ffl} \beta]$, таких что $[A \to \alpha \text{ffl} X\beta]I$
Функция ПЕРЕХОД используется для определения переходов в LR(0) – автомате грамматики. Состояние автомата соответствует множеству пунктов, а ПЕРЕХОД(I, X) указывает переход из состояния I при входном символе X.

Алгоритм вычисления канонического набора множеств LR(0) - пунктов

1) Считаем C = ЗАМЫКАНИЕ([S' -> • S])

2) $C_m = C$

3) Для всех I є C, для каждого (X є "?"_?? U "?"_??): Если ПЕРЕХОД(I, X) не пусто и ПЕРЕХОД(I, X) не принадлежит C, то добавляем ПЕРЕХОД(I, X) в C

4) Повторяем пункт 3 пока $C_m ! = C$

Множество FIRST

Для произвольной строки символов ? грамматики, ? є $?^?, FIRST?$
Если x- терминал, то FIRST(x) = x

Если X- нетерминал, и есть правила X ->??_1,. . . , ??_?? для k ? 1, то поместим a в FIRST(X), если при i = 1,. . . ,k : a є FIRST(??_??) и ? є FIRST(??_1),. . . , FIRST(??_??)

Если есть правило X ->? , то добавляем ? к множеству FIRST(X)

Множество FOLLOW

Для нетерминала A, множеством FOLLOW(A) является множество терминалов a, которые могут располагаться непосредственно справа от A в некоторой сентенциальной форме.

1) Помещаем

$$FOLLOW(S)[A- >?B?], FIRST(?)?, FOLLOW(B)$$

2) Если имеется либо [A -> ? B], либо [A -> ? B ?], где ? є FIRST(?), то все эл-ты из множества FOLLOW(A) добавляем во множество FOLLOW(B).

Глава 2

Описания машины Муся64

2.1. Основы архитектуры

Машина Муся64 представляет из себя стековую машину, 64 разрядную, причем у каждого элемента стека имеется однобайтовый тэг, указывающий тип содержимого элемента стека.

1) Тип 0 – означает неизвестный тип.
2) Тип 1 – целые без знака.
3) Тип 2 – целые со знаком.
4) Остальные значения тэгов зарезервированы.

Использование в операции аргумента с зарезервированным тэгом вызывает аварийный останов машины.

У машины имеется регистр предыдущего результата. В регистр предыдущего результата заносится содержимое вершины стека, которое было до выполнения операции.

Разрядность машины, равная 64, означает, что 64-х разрядным является каждый элемент стека без сопутствующего тэга. Если в элементе содержится логическое значение, то значение ложь представляется числом 0, а истина 1.

Перейдём теперь к описанию системы команд, разбив команды на группы.

2.2. Описание команд

2.2.1. Арифметические команды

В эту группу входят команды выполнения арифметических операций (сложения, вычитания, умножения, деления, возведения в степень, смены знака). Данные команды обозначаются `op_ADD`, `op_SUB`, `op_MULT`, `op_DIV`, `op_POWER`, `op_CHANGE_SIGN` соответственно.

При арифметических операциях можно свободно смешивать знаковые и беззнаковые целые числа. Никакое другое смешивание операндов разных типов не допускается. Попытка такого смешивания приводит к аварийному останову машины.

Если оба операнда арифметических операций беззнаковые - выполняется беззнаковая операция. Результат этой операции будет того же типа, что и операнды. Если же типы операндов разные, то возможно два случая:

1) Знаковый операнд не отрицательный, тогда тэг знакового значения меняется на беззнаковый. Результат – беззнаковое целое.
2) Один из операндов отрицательный знаковые, а другой беззнаковый. В этом случае, внутри машины оба преобразуются в знаковые целые числа (128 бит) и после выполния арифметической операции на вершину стека помешаются 64 младших бита от результата.

2.2.2. Команды сравнения

В данную группу входят команды выполения операций сравнения.

Для операций "меньше "больше "меньше или равно "больше или равно"команды имею по 2 варианта(для знаковых и беззнаковых операндов).

Команды относящиеся к этой группе: `op_EQ`, `op_NEQ`, `op_SIGN_LT`, `op_SIGN_GT`, `op_SIGN_LEQ`, `op_SIGN_GEQ`,`op_UNSIGN_LT`, `op_UNSIGN_GT`, `op_UNSIGN_LEQ`, `op_UNSIGN_GEQ`.

Если оба операнда беззнаковые, то выбирается беззнаковая операция. Если же один из операндов знаковый, то возможно два случая:

1) Знаковый операнд не отрицательный, тогда тэг знакового значения меняется на беззнаковый. Выполняется беззнаковая операция.

2) Один из операндов отрицательный знаковые, а другой беззнаковый. В этом случае, внутри машины оба преобразуются в знаковые целые числа (128 бит) и выполняется знаковая операция.

2.2.3. Логические команды

‘ В эту группу входят команды выполнения логических операций (И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ, НЕ, НЕ–И, НЕ–ИЛИ;). Названия входящих в данную группу команд: `op_AND`, `op_OR`, `op_XOR`, `op_NOT`, `op_NAND`, `op_NOR`.

2.2.4. Прочие команды

Команда `op_LOAD_INT_CONST`. Данная команда загружает в стек целую беззнаковую константу.

Команда `op_LOAD_INT_SIGN_CONST`. Данная команда загружает в стек целую знаковую константу.

Команда `op_LOAD_FALSE`. Данная команда загружает в стек логическое значение "ложь".

Команда `op_LOAD_TRUE`. Данная команда загружает в стек логическое значение "истина".

Команда `op_LOG_TO_INT`. Данная команда переводит логическое значение в целое в соответствии с правилом машины.

Команда `op_CONDITION`. Данная команда соответствует тернарному оператору(условному).

Команда `op_STOP`. Данная команда приводит к остановке машины.

2.2.5. Кодировка команд

В следующей таблице описана кодировка команд Ять–машины (через imm32 обозначено непосредственно заданное тридцатидвухразрядное значение, коды команд записаны в шестнадцатиричном виде).

Команда	**Кодировка**
Логические команды	
`op_NOR`	00
`op_OR`	01
`op_XOR`	02
`op_NAND`	03
`op_AND`	04
`op_NOT`	05

Команда	**Кодировка**
Команды сравнения	
op_EQ	06
op_NEQ	07
op_SIGN_LT	08
op_SIGN_GT	09
op_SIGN_LEQ	0A
op_SIGN_GEQ	0B
op_UNSIGN_LT	0C
op_UNSIGN_GT	0D
op_UNSIGN_LEQ	0E
op_UNSIGN_GEQ	0F
Арифметические команды	
op_ADD	10
op_SUB	11
op_MULT	12
op_DIV	13
op_CHANGE_SIGN	14
op_POWER	15
Прочие команды	
op_LOAD_INT_CONST	16
op_LOAD_INT_SIGN_CONST	17
op_LOAD_TRUE	18
op_LOAD_FALSE	19
op_LOG_TO_INT	1A
op_CONDITION	1B
op_STOP	1C

Все прочие коды интерпретируются как команда „нет операции“.

Глава 3

Компиляция языка Киса

3.1. Грамматика языка и вид лексем

3.1.1. Грамматика языка

$S \rightarrow$ T **?** T **:** T | T
$T \rightarrow$ T **!||** E | T **||** E | T **^^** E | E
$E \rightarrow$ E **!&&** F | E **&&** F | F
$F \rightarrow$ **!** F | G
$G \rightarrow$ G = H | G **!=** H | G < H | G > H | G <= H | G >= H | H
$H \rightarrow$ H + K | H **-** K | K
$K \rightarrow$ K ***** L | K / L | L
$L \rightarrow$ M ****** L | M
$M \rightarrow$ + N | **-** N | N
$N \rightarrow$ **(** S **)** | **истина** | **ложь** | **целое**

3.1.2. Лексемы и их форматы

Текст программы состоит из лексем. В языке имеется три класса лексем:

1) ключевые слов;
2) числа;
3) знаки операций и разделители.

Никакая лексема не может разбиваться на части пробельными символами (т.е. пробелами, табуляциями и концами строк) или комментариями. Опишем каждый класс лексем.

Ключевые слова языка

Следующие слова не могут использоваться в качестве идентификаторов:
истина **ложь**

Числа

Числа — это беззнаковые целые. Знаковые константы — это беззнаковые константы, к которым применена унарная операция смены знака (операция „−“). Запись в РБНФ:

целое→*десятичное*|*шестнадцатиричное*
десятичное→*десятичная_ цифра*{*десятичная_ цифра*}
десятичная_ цифра→0|1|2|3|4|5|6|7|8|9
шестнадцатиричное→0(x|X)*шестнадцатиричная_ цифра*{*шестнадцатиричная_ цифра*}
шестнадцатиричная_ цифра→0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F|a|b|c|d|e|f

Разделители и знаки операций

Данные знаки операций и разделителей поддерживаются анализатором:

```
?       ||      &&      !=      <=      -       **
:       ^^      !       <       >=      *       (
!||     !&&     =       >       +       /       )
```

3.2. Структура компилятора

Данный компилятор состоит из следующих классов:

1) Scaner – в данном классе реализован Лексический анализатор. Читает входной буфер и определяет текущую лексему.

2) Parser – данный класс является Синтаксическим анализатором типа SLR-1. В этом классе происходит вызов метода определения лексемы класса Scaner, после чего происходит решение задачи синтаксического разбора и генерация промежуточного кода. Это делается до тех пор, пока не будет прочитана вся входная строка.

3) Parser_stack – данный класс используется в классе Parser, как стек для решения задачи синтаксического разбора.

4) Code_buffer – в данный класс происходит запись промежуточного кода, сгенерированного в процессе работы класс Parser. При успешном завершении работы компилятора содержимое этого контейнера записывается в бинарный файл.

5) error_count – данный класс отвечает за сбор количества ошибок. Он используется во всех классах. При обнаружении ошибки(лексической или синтаксической) выдается поясняющее сообщение и увеличивается счетчик ошибок.

3.3. Лексический анализатор

Далее будет подробно описан класс Scaner.

3.3.1. Список автоматов

Разобьём лексический анализатор на несколько конечных автоматов, каждый из которых распознаёт свой вид лексем.

В приводимой ниже таблице указаны имена этих конечных автоматов и выполняемые автоматами функции.

Таблица 3.3.1. Таблица имён конечных автоматов и выполняемых автоматами функций

Название автомата	**Пояснение**
a_START	Автомат, в который попадаем в начале работы сканера. Этот автомат инициализирует другие автоматы.
a_UNKNOWN	Этот автомат обрабатывает лексемы, не являющиеся ни целыми числами, ни строковыми константами, ни идентификаторами, ни ключевыми словами, ни разделителями.
a_NUMBER	Автомат, обрабатывающий целые числа.
a_KEYWORD	Автомат, обрабатывающий ключевые слова.
a_DELIMITER	Автомат, обрабатывающий разделители и знаки операций.

Запишем теперь номера этих автоматов:

```
enum Automata_name {a_START, a_UNKNOWN, a_DELIMITERS, a_KEYWORD,
a_NUMBER };
```

3.3.2. Категории символов

Разобьём все символы на следующие категории:

Таблица 3.3.2. Классификация однобайтовых символов.

Категория символов	**Входящие в категорию символы**
Пробельные символы	0x00..0x20
Десятичные цифры	'0'..'9'
Буквы Xx	'Xx'
Символы шестнадцатиричных цифр	'ABCFDabcfd'
Символы, с которых может начинаться ключевое слово	ил
Символы, состовляющие тело ключевых слов	ажность
Односимвольные разделители	`?:=+-/()`
Символы, с которых могут начинаться многосимвольные разделители	`!|&^<>*`
Прочее	Остальные символы

А теперь программа, генерирующая таблицу классификации символов:

```
#include <stdio.h>

int main(int argc, char** argv) {
    enum char_categories {  SPACES,     ZERO_DIGIT,      DIGIT_WITHOUT_ZERO   , LETTERS_X,
                            HEX_DIGIT , ONE_LETTER_DELIMITER,
                            MULTILETTER_DELIMITER_BEGIN,     KEYWORD_BEGIN, KEYWORD_BODY,
                            OTHER };

    int table[256];

    unsigned char* digits_chars = (unsigned char*)"0123456789";

    unsigned char* keyword_begin_chars = (unsigned char*)"ил";
    unsigned char* one_letter_delimiters_chars = (unsigned char*)"?:=+-/()";
    unsigned char* multiletter_delimiter_begin_chars = (unsigned char*)"!|&^<>*";
    unsigned char* hex_num_chars = (unsigned char*)"ABCDEFabcdef";
    unsigned char* letter_chars = (unsigned char*)"ажность";
    for(int i = 0; i<33; table[i++] = SPACES);
    for(int i = 33; i<256; table[i++] = OTHER);

    for( ; *digits_chars; table[*digits_chars++] = DIGIT_WITHOUT_ZERO);
    for( ; *hex_num_chars; table[*hex_num_chars++] = HEX_DIGIT);

    for( ; *keyword_begin_chars; table[*keyword_begin_chars++] = KEYWORD_BEGIN);
    for( ; *letter_chars; table[*letter_chars++] = KEYWORD_BODY);

    for( ; *one_letter_delimiters_chars; table[*one_letter_delimiters_chars++]
            = ONE_LETTER_DELIMITER);
    for( ; *multiletter_delimiter_begin_chars; table[*multiletter_delimiter_begin_chars++]
            = MULTILETTER_DELIMITER_BEGIN)

    table['x'] = LETTERS_X;
    table['X'] = LETTERS_X;
```

```
    table['0'] = ZERO_DIGIT;

    for(int i = 0; i<256; i+=16){
        printf("  %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d, %d\n",
                table[i]   ,  table[i+1] ,  table[i+2] ,  table[i+3] ,
                table[i+4] ,  table[i+5] ,  table[i+6] ,  table[i+7] ,
                table[i+8] ,  table[i+9] ,  table[i+10],  table[i+11],
                table[i+12],  table[i+13],  table[i+14],  table[i+15]);
    }
    return 0;
}
```

Откомпилировав эту программу, а затем запустив с перенаправлением вывода в файл, будем иметь следующее:

```
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 6, 9, 9, 9, 9, 6, 9, 5, 5, 6, 5, 9, 5, 9, 5,
  1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 5, 9, 6, 5, 6, 5,
  9, 4, 4, 4, 4, 4, 4, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 3, 9, 9, 9, 9, 9, 6, 9,
  9, 4, 4, 4, 4, 4, 4, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 3, 9, 9, 9, 6, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  8, 9, 9, 9, 9, 9, 8, 9, 7, 9, 9, 7, 9, 8, 8, 9,
  9, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 8, 9, 9, 9
```

3.3.3. Коды лексем

Опишем структуру используемых сканером глобальных переменных.

Прежде всего, каждая лексема имеет свой код, указывающий на разновидность лексемы.

В следующей таблице собраны коды лексем.

Код	**Пояснение**
0	нет лексемы (специальное значение `NONE`)
1	Условный оператор
2	Двоеточие
3	не-или
4	или
5	исключающее или
6	не-и
7	и
8	не
9	равно
10	не-равно
11	меньше
12	больше
13	меньше или равно
14	больше или равно
15	плюс

Код	Пояснение
16	минус
17	умножение
18	деление
19	возведение в степень
20	открывающаяся скобка
21	закрывающаяся скобка
22	ключевое слово истина
23	ключевое слово ложь
24	целое
25	неизвестно что
26	возможно, не-или
27	возможно, или
28	возможно, исключающее или
29	возможно, не-и
30	возможно, и

Приведём теперь код на Си++, описывающий нужные для сканера типы и переменные:

```
 /* Типы данных для сканера. */

#define MAX_NUMBER ULLONG_MAX #define DEC_SET ((1<<ZERO_DIGIT) +
(1<<DIGIT_WITHOUT_ZERO)) #define HEX_SET ((1<<ZERO_DIGIT) +
(1<<DIGIT_WITHOUT_ZERO) + (1<<HEX_DIGIT))

enum LEXEM_CODE {
    NONE            , CONDITION      , COLON     , NOR       ,      OR          ,
    XOR             , NAND           , AND       , NOT       ,      EQ          ,
    NEQ             , LT             , GT        ,  LEQ      ,      GEQ         ,
    PLUS            , MINUS          , MULT      ,  DIV      ,      DEGREE      ,
    OPENED_BRACKET  , CLOSED_BRACKET , kw_ISTINA ,  kw_LOZH  ,      INT         ,
    UNKNOWN         , MAYBE_NOR      , MAYBE_OR  , MAYBE_XOR ,      MAYBE_NAND  ,
    MAYBE_AND
};

struct returned_lexem{
    LEXEM_CODE probably_code;
    __int128 number;
};

inline int hex_digit_to_int(int digit) {
    int temp = digit - '0';
    return (temp<=9) ? temp : ((temp&0xDF) - 7);
}

static const char categories_table[] = {
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 6, 9, 9, 9, 9, 6, 9, 5, 5, 6, 5, 9, 5, 9, 5,
  1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 5, 9, 6, 5, 6, 5,
  9, 4, 4, 4, 4, 4, 4, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 3, 9, 9, 9, 9, 9, 6, 9,
  9, 4, 4, 4, 4, 4, 4, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 3, 9, 9, 9, 6, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
```

```
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
  8, 9, 9, 9, 9, 9, 8, 9, 7, 9, 9, 7, 9, 8, 8, 9,
  9, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 8, 9, 9, 9
};

 /* Класс сборщика ошибок */

 class Error_count {
    private:
        int number_of_errors;
    public:
        int get_number_of_errors();

        void increment_number_of_errors();

        void print_number_of_errors();

        Error_count();
}; int Error_count::get_number_of_errors(){
    return number_of_errors;
}

void Error_count::increment_number_of_errors(){
    ++number_of_errors;
    return;
}

void Error_count::print_number_of_errors(){
    printf("Всего ошибок: %d",number_of_errors);
    return;
}

Error_count::Error_count(){
    number_of_errors = 0;
    return;
}
```

3.3.4. Схема сканера

Работа сканера реализована через ссылки на функции. Далее частично описан класс сканер. Описание автоматов сканера будет приведено в следующих разделах. /* Класс сканер */

```
class Scaner {
    public:
        void       back();
        returned_lexem get_current_lexem();

        unsigned int get_line_number();

        Scaner(const unsigned char* t, Error_count* e);

    private:
        enum CATEGORY {
            SPACES,         ZERO_DIGIT,      DIGIT_WITHOUT_ZERO   ,
            LETTERS_X,      HEX_DIGIT ,      ONE_LETTER_DELIMITER,
```

```
            MULTILETTER_DELIMITER_BEGIN,     KEYWORD_BEGIN, KEYWORD_BODY,
            OTHER
        };

        enum Automaton_name {
            a_START, a_UNKNOWN, a_DELIMITERS, a_KEYWORD, a_NUMBER
        };

        enum a_NUMBER_states {
            DEC_INT, MB_HEX, HEX_START, HEX, OUT_OF_RANGE_DEC, OUT_OF_RANGE_HEX};

        Automaton_name Automaton;
        unsigned int   state;

        typedef bool (Scaner::*Automaton_proc)();
        static Automaton_proc procs[];

        bool start_proc();
        bool unknown_proc();
        bool delimiters_proc();
        bool keyword_proc();
        bool number_proc();

        void diagnostic_for_delimiters();

        unsigned char* text;
        unsigned char* pcurrent_char;
        unsigned char* lexem_begin;

        unsigned char  ch;
        CATEGORY       char_category;

        returned_lexem Lexem;

        unsigned int   current_line;

        Error_count*   en;
};

Scaner::Scaner(const unsigned char* t, Error_count* e){
    text = pcurrent_char = lexem_begin = const_cast<unsigned char*>(t);
    current_line = 1;
    en = e;
    Lexem.number = 0;
    return;
}

unsigned int Scaner::get_line_number(){
    return current_line;
}

void Scaner::back(){
    pcurrent_char = lexem_begin;
    return;
}

Scaner::Automaton_proc Scaner::procs[] = {
     &Scaner::start_proc     , &Scaner::unknown_proc,
     &Scaner::delimiters_proc, &Scaner::keyword_proc,
```

```
    &Scaner::number_proc
};
```

3.3.5. Начальный автомат сканера

Автомат a_START первым вызывается при каждом вызове лексического анализатора:

```
bool Scaner::start_proc(){
    bool t = true;
    Lexem.number = 0;
    switch(char_category){
        case SPACES:
            if(ch == '\n'){
                current_line++;
            }
            break;

        case OTHER: case KEYWORD_BODY:
            Automaton = a_UNKNOWN;
            Lexem.probably_code = UNKNOWN;
            printf("Нераспознаваемая лексема в строке %d.\n", current_line);
            en->increment_number_of_errors();
            break;

        case ONE_LETTER_DELIMITER:
            switch((char)ch){
                case '?':
                    Lexem.probably_code = CONDITION;
                    break;

                case ':':
                    Lexem.probably_code = COLON;
                    break;

                case '=':
                    Lexem.probably_code = EQ;
                    break;

                case '+':
                    Lexem.probably_code = PLUS;
                    break;

                case '-':
                    Lexem.probably_code = MINUS;
                    break;

                case '/':
                    Lexem.probably_code = DIV;
                    break;

                case '(':
                    Lexem.probably_code = OPENED_BRACKET;
                    break;

                case ')':
                    Lexem.probably_code = CLOSED_BRACKET;
                    break;
            }
            pcurrent_char++;
```

```
            t = false;
            break;

        case MULTILETTER_DELIMITER_BEGIN:
            Automaton = a_DELIMITERS;
            switch((char)ch){
                case '!':
                    state = 0;
                    break;

                case '|':
                    state = 6;
                    break;

                case '^':
                    state = 8;
                    break;

                case '&':
                    state = 10;
                    break;

                case '<':
                    state = 12;
                    break;

                case '>':
                    state = 14;
                    break;

                case '*':
                    state = 16;
                    break;
            }
            lexem_begin = pcurrent_char - 1;
            break;

        case KEYWORD_BEGIN:
            state = (((char)ch) == 'л')? 6: 0;
            Automaton = a_KEYWORD;
            lexem_begin = pcurrent_char - 1;
            break;

        case ZERO_DIGIT: case DIGIT_WITHOUT_ZERO:
            state = char_category & 1;
            Lexem.number = ch - '0';
            Lexem.probably_code = INT;
            Automaton = a_NUMBER;
            lexem_begin = pcurrent_char-1;
            break;

    }
    return t;
}
```

3.3.6. Автомат обработки несоответствий

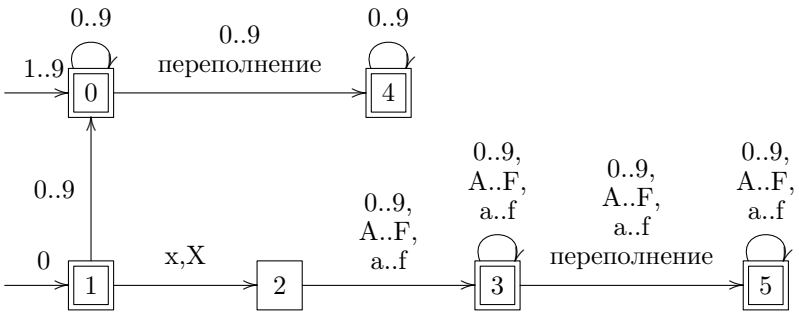
Автомат a_UNKNOWN обрабатывает неизвестные лексемы:

```
bool Scaner::unknown_proc(){
    return char_category == OTHER;
}
```

3.3.7. Автомат для обработки чисел.

3.3.8. Диаграмма переходов для чисел

Приведём диаграмму переходов конечного автомата, распознающего десятичные и шестнадцатиричные числа:



3.3.9. Автомат a_NUMBER

Добавим текст автомата a_NUMBER:

```
bool Scaner::number_proc(){
    bool t = true;
    switch(state){
        case DEC_INT:
            if( (1<< char_category) & DEC_SET )
            {
                Lexem.number = Lexem.number * 10 + (ch - '0');
                if(Lexem.number > MAX_NUMBER)
                {
                    printf("Слишком большое целое число в строке %d.\n", current_line);
                    en->increment_number_of_errors();
                    state = OUT_OF_RANGE_DEC;
                }
            }
            else
            {
                t = false;
            }
            break;

        case MB_HEX:
            if( (1<<char_category) & DEC_SET)
            {
                state = DEC_INT;
                Lexem.number = ch - '0';
            }
            else if( char_category == LETTERS_X)
            {
                state = HEX_START;
            }
            else
            {
```

```
                t = false;
            }
            break;

        case HEX_START:
            switch(char_category){
                case ZERO_DIGIT: case HEX_DIGIT: case DIGIT_WITHOUT_ZERO:
                    Lexem.number = Lexem.number * 16 + hex_digit_to_int(ch);
                    state = HEX;
                    break;

                default:
                    printf("В строке %d ожидается шестнадцатиричная цифра.\n", current_line);
                    en->increment_number_of_errors();
                    Lexem.number = 0;
                    t = false;
            }
            break;

        case HEX:
            if( Lexem.number > MAX_NUMBER )
            {
                printf("Слишком большое целое число в строке %d.\n", current_line);
                en->increment_number_of_errors();
                state = OUT_OF_RANGE_HEX;
            }
            else
            {
                if( (1<<char_category) & HEX_SET)
                {
                    Lexem.number = Lexem.number * 16 + hex_digit_to_int(ch);
                }
                else
                {
                    t = false;
                }
            }
            break;

        case OUT_OF_RANGE_DEC:
            if( !((1<<char_category) & DEC_SET) )
            {
                t = false;
            }
            break;

        case OUT_OF_RANGE_HEX:
            if( !((1<<char_category) & HEX_SET) )
            {
                t = false;
            }
            break;
    }
    return t;
}
```

Изменения, вносимые в последнюю инструкцию switch:

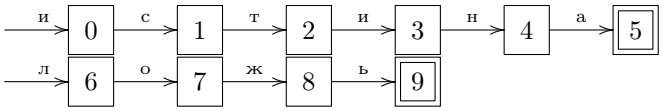
```
        case a_NUMBER:
            switch(state){
```

```
        case HEX_START:
            if( !((1<<char_category) & HEX_SET) )
            {
                printf("В строке %d ожидается шестнадцатиричная цифра.\n", current_line);
                    en->increment_number_of_errors();
                    Lexem.number = 0;
            }
            break;
    }
    break;
```

3.3.10. Автомат для обработки ключевых слов

Диаграмма переходов для ключевых слов

Прежде всего приведём диаграммы переходов для ключевых слов:



Текст автомата a_KEYWORD

Судя по диаграммам переходов для ключевых слов, из каждого состояния можно перейти в одно из не более чем четырёх состояний. Отсутствующие символы считаем нулями.

1й	2й	3й	4й

Рис. 3.3.1. Способ хранения символов, по которым происходит переход.

Таким образом, символы „ео“хранятся как

е	о	0x00	0x00

Рис. 3.3.2. Пример хранения символов, по которым происходит переход.

Далее, согласно диаграммам переходов, если из состояния с номером x по первому символу происходит переход в состояние с номером y, то по второму символу — в состояние с номером $y+1$, и так далее. Поэтому нам достаточно хранить номер состояния, в которое происходит переход по первому символу. Назовём это состояние первым состоянием. Ну и, наконец, нам нужно хранить код лексемы.

Итак, элемент таблиц переходов автоматов обработки ключевых слов и разделителей имеет вид:

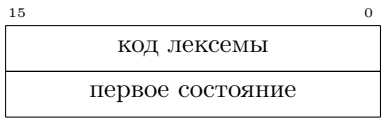


Рис. 3.3.3. Формат элемента таблиц переходов автоматов обработки ключевых слов и разделителей.

Описание на Си++ этой структуры и таблиц переходов:

```
typedef struct tag_ELEM{
```

```
    unsigned short int code;
    unsigned short int first_state;
    char               symbols[4];
} ELEM;

ELEM a_KEYWORD_jump_table[] = {
    {UNKNOWN  , 1, {'с',0,0,0}}, /*состояние 0 $ и...    */
    {UNKNOWN  , 2, {'т',0,0,0}}, /*состояние 1 $ ис...   */
    {UNKNOWN  , 3, {'и',0,0,0}}, /*состояние 2 $ ист...  */
    {UNKNOWN  , 4, {'н',0,0,0}}, /*состояние 3 $ исти... */
    {UNKNOWN  , 5, {'а',0,0,0}}, /*состояние 4 $ истин...*/
    {kw_ISTINA, 0, {0,0,0,0}  }, /*состояние 5 $ истина  */

    {UNKNOWN  , 7, {'о',0,0,0}}, /*состояние 6 $ л...    */
    {UNKNOWN  , 8, {'ж',0,0,0}}, /*состояние 7 $ ло...   */
    {UNKNOWN  , 9, {'ь',0,0,0}}, /*состояние 8 $ лож...  */
    {kw_LOZH  , 0, {0,0,0,0}  }  /*состояние 9 $ ложь    */
};
```

Работать автомат `a_KEYWORD_jump` будет так:

1) если текущий символ не является ни буквой, ни цифрой, то возвращаем лексему, в качестве кода лексемы выдавая значение `a_KEYWORD_jump_table[state].code`;
2) вычисляем `(ch * 0x01010101) ^ a_KEYWORD_jump_table[state].symbols`;
3) в полученном четырёхбайтовом целом числе вычисляем номер самого старшего нулевого байта в соответствии с листингом 6.4 на стр.102 книги [2] и обозначим этот номер через y;
4) если $y = 4$, то переходим в автомат `a_IDENT`;
5) если же $y = \overline{0,3}$, то выполняем оператор

```
state = a_KEYWORD_jump_table[state].first_state + y;
```

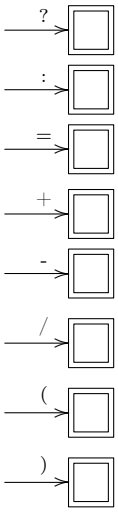
Добавим теперь к тексту сканера текст автомата `a_KEYWORD`:

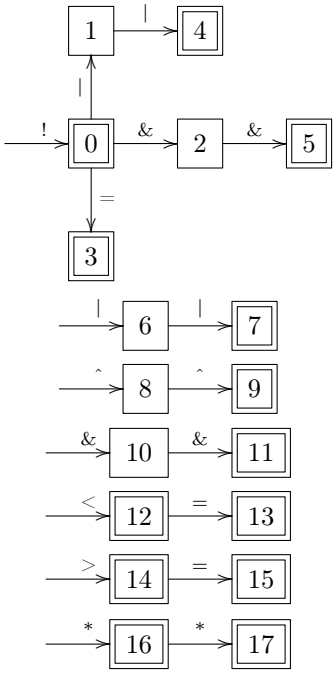
```
bool Scaner::keyword_proc(){
    bool t = false;
    Lexem.probably_code = static_cast<LEXEM_CODE>(a_KEYWORD_jump_table[state].code);
    if( (1<<char_category) & ( (1<<KEYWORD_BEGIN) | (1<<KEYWORD_BODY) ) ){
        int y = search_char(ch, a_KEYWORD_jump_table[state].symbols);
        if(y != 4){
            state = a_KEYWORD_jump_table[state].first_state + y;
            t = true;
        }
    }
    return t;
}

 Изменения, вносимые в последнюю инструкцию switch:
        case a_KEYWORD:
            Lexem.probably_code =
                static_cast<LEXEM_CODE>(a_KEYWORD_jump_table[state].code);
            break;
```

3.3.11. Автомат по обработке разделителей

Диаграмма переходов для разделителей

Приведём теперь диаграммы переходов для разделитилей:





Текст автомата a_DELIMITER

Отличие в работе этого автомата от автомата обработки ключевых слов в том, что допускающее состояние автомата не обязательно должно быть последним в последовательности.

```
ELEM a_DELIMITERS_jump_table[] = {
    {NOT         , 1 , {'|','&','=',0} },   /*состояние 0   $ !   */
    {MAYBE_NOR   , 4 , {'|',0,0,0} },       /*состояние 1   $ !|...   */
    {MAYBE_NAND  , 5 , {'&',0,0,0} },       /*состояние 2   $ !&...   */
    {NEQ         , 0 , {0,0,0,0} },         /*состояние 3   $ !=   */
    {NOR         , 0 , {0,0,0,0} },         /*состояние 4   $ !||   */
    {NAND        , 0 , {0,0,0,0} },         /*состояние 5   $ !&&   */
```

```
    {MAYBE_OR    , 7 , {'|',0,0,0} },       /*состояние 6   $ |...   */
    {OR          , 0 , {0,0,0,0} },         /*состояние 7   $ ||   */

    {MAYBE_XOR  , 9 , {'^',0,0,0} },        /*состояние 8   $ ^...   */
    {XOR        , 0 , {0,0,0,0} },          /*состояние 9   $ ^^   */

    {MAYBE_AND , 11 , {'&',0,0,0} },        /*состояние 10   $ &...   */
    {AND       , 0 ,  {0,0,0,0}   },        /*состояние 11   $ &&     */

    {LT        , 13 , {'=',0,0,0} },        /*состояние 12   $ <...   */
    {LEQ       , 0 ,  {0,0,0,0}   },        /*состояние 13   $ <=     */

    {GT        , 15 , {'=',0,0,0} },        /*состояние 14   $ >...   */
    {GEQ       , 0 ,  {0,0,0,0}   },        /*состояние 15   $ >=     */

    {MULT      , 17 , {'*',0,0,0} },        /*состояние 16   $ *...   */
    {DEGREE    , 0 ,  {0,0,0,0}   },        /*состояние 13   $ **     */
};

bool Scaner::delimiters_proc(){
    bool t = false;
    Lexem.probably_code = static_cast<LEXEM_CODE>(a_DELIMITERS_jump_table[state].code);
    int y = search_char(ch, a_DELIMITERS_jump_table[state].symbols);
    if(y != 4){
        state = a_DELIMITERS_jump_table[state].first_state + y;
        t = true;
    }
    return t;
}
```

Изменения, вносимые в последнюю инструкцию switch:

```
        case a_DELIMITERS:
            Lexem.probably_code = static_cast<LEXEM_CODE>(a_DELIMITERS_jump_table[state].code);
            diagnostic_for_delimiters();
            break;
```

Для диагностики ошибок в разделителях используется следующий код:

```
 static const char* op_as_string[] = {
    "", "", "", "!||", "||", "^^", "!&&", "&&"
};

void Scaner::diagnostic_for_delimiters(){
    if(Lexem.probably_code > UNKNOWN){
        Lexem.probably_code = static_cast<LEXEM_CODE>(Lexem.probably_code - MAYBE_NOR + NOR);
        printf("В строке %d ожидается операция %s\n", current_line,
               op_as_string[Lexem.probably_code]);
        en->increment_number_of_errors();
    }
    return;
}
```

3.3.12. Главный цикл сканера

```
returned_lexem Scaner::get_current_lexem(){
    Automaton = a_START;
    Lexem.probably_code = NONE;
    lexem_begin = pcurrent_char;
    state = 0;

    bool t = true;

    while(ch = *pcurrent_char++){
        char_category = static_cast<CATEGORY>(categories_table[ch]);
        t = (this->*procs[Automaton])();

        if(!t){
            pcurrent_char--;
            if(Automaton == a_DELIMITERS){
                diagnostic_for_delimiters();
            }
            return Lexem;
        }
    }
    pcurrent_char--;
    switch(Automaton){
        case a_KEYWORD:
            Lexem.probably_code = static_cast<LEXEM_CODE>(a_KEYWORD_jump_table[state].code);
            break;

        case a_DELIMITERS:
            Lexem.probably_code = static_cast<LEXEM_CODE>(a_DELIMITERS_jump_table[state].code);
            diagnostic_for_delimiters();
            break;

        case a_NUMBER:
            switch(state){
                case HEX_START:
                    if( !((1<<char_category) & HEX_SET) )
                    {

                            printf("В строке %d ожидается шестнадцатиричная цифра.\n", current_line)
                            en->increment_number_of_errors();
                            Lexem.number = 0;
                    }
                    break;
            }
            break;

        default:
            ;
    }
    return Lexem;
}
```

3.3.13. Пример работы сканера

Сканеру была подана следующая строка:

```
истина && (ложь ^^ истина !|| ложь !&& истина ) | (10 != 0x10)
```

Результатом работы стала последовательность лексем и диагностических сообщений:

```
kw_ISTINA
AND
OPENED_BRACKET
kw_LOZH
XOR
kw_ISTINA
NOR
kw_LOZH
NAND
kw_ISTINA
CLOSED_BRACKET
В строке 1 ожидается операция ||
||
OR
OPENED_BRACKET
INT
NEQ
INT
CLOSED_BRACKET
NONE
NONE
Всего ошибок: 1
```

3.4. Синтасический анализатор

3.4.1. Таблица синтаксического анализа

Прежде всего необходимо рассмотреть таблицу синтаксического анализа данного компилятора. Она составлена по правилам SLR-1 синтаксического анализа.

Таблица Действие

```
Таблица действий состоит из пар вида
(указатель на таблицу действий для состояния, размер таблицы для состояния)

Элементы таблицы действий для состояния выглядят так:
(код лексемы, действие)
Элементы таблицы действий упорядочены по возрастанию кодов лексем.
Кроме того, первой компонентой элемента таблицы действий для состояния
может быть специальное значение ANY, большее любого кода лексемы. Оно означает,
что если элемента таблицы с нужным кодом лексемы нет, то выполняется действие,
указанное второй компонентой пары с ANY в качестве кода лексемы.
Под действие отведём один байт (unsigned char). Этот байт представляет собой
битовую структуру со следующими компонентами:
   бит 15 (is_undefined) --- нужно ли при переносе или свёртке устанавливать признак
                             неопределённого значения;
   биты 14--13 (action_code) --- код действия (см. enum Action_names);
   биты 12--0 (argument) --- номер правила для свёртки при свёртке и номер
                             переносимого состояния при переносе.

#define ANY 200

#define ACTION(isdef,actcode,arg) (((isdef) << 15) | ((actcode) <<
                13) | (arg))

#define DEFINED_VALUE   0
```

```
#define UNDEFINED_VALUE 1

struct Entry_of_action_table_for_state {
    unsigned short lexem_code;
    unsigned short action_code;
};

#define ELEMS_IN_TABLE_FOR_STATE(t)
            (sizeof(t)/sizeof(Entry_of_action_table_for_state))

struct Entry_of_action_table {
    Entry_of_action_table_for_state* ptr_to_action_table_for_state;
    size_t                           number_of_actions;
};

static inline Action_info unpack_action_code(unsigned int a){
    Action_info ainfo;

    ainfo.is_undefined = a >> 15;
    ainfo.argument   = a & 0x1fff;
    ainfo.action     = static_cast<Action_names>((a >> 13) & 3);

    return ainfo;
}

static Entry_of_action_table_for_state actions_for_state_0[] = {
    {NOT,               ACTION(DEFINED_VALUE,   act_SHIFT,  11)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_SHIFT,  12)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_SHIFT,  13)},
    {OPENED_BRACKET,    ACTION(DEFINED_VALUE,   act_SHIFT,  14)},
    {kw_ISTINA,         ACTION(DEFINED_VALUE,   act_SHIFT,  15)},
    {kw_LOZH,           ACTION(DEFINED_VALUE,   act_SHIFT,  16)},
    {t_INT,             ACTION(DEFINED_VALUE,   act_SHIFT,  17)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_SHIFT,  11)}
};

static Entry_of_action_table_for_state actions_for_state_1[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_OK,     0 )}
};

static Entry_of_action_table_for_state actions_for_state_2[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, S_IS_T)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_SHIFT,  18)},
    {NOR,               ACTION(DEFINED_VALUE,   act_SHIFT,  19)},
    {OR,                ACTION(DEFINED_VALUE,   act_SHIFT,  20)},
    {XOR,               ACTION(DEFINED_VALUE,   act_SHIFT,  21)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, S_IS_T)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, S_IS_T)}
};

static Entry_of_action_table_for_state actions_for_state_3[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_E)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_E)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_E)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_E)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_E)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_E)},
    {NAND,              ACTION(DEFINED_VALUE,   act_SHIFT,  22)},
    {AND,               ACTION(DEFINED_VALUE,   act_SHIFT,  23)},
```

```
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_E)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, T_IS_E)}
};

static Entry_of_action_table_for_state actions_for_state_4[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_F)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_F)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_F)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_F)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_F)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_F)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_F)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_F)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_F)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, E_IS_F)}
};

static Entry_of_action_table_for_state actions_for_state_5[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_G)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_G)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_G)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_G)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_G)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_G)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_G)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_G)},
    {EQ,                ACTION(DEFINED_VALUE,   act_SHIFT,  24)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_SHIFT,  25)},
    {LT,                ACTION(DEFINED_VALUE,   act_SHIFT,  26)},
    {GT,                ACTION(DEFINED_VALUE,   act_SHIFT,  27)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_SHIFT,  28)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_SHIFT,  29)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_G)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, F_IS_G)}
};

static Entry_of_action_table_for_state actions_for_state_6[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_SHIFT,  30)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_SHIFT,  31)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_H)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, G_IS_H)}
};

static Entry_of_action_table_for_state actions_for_state_7[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
```

```
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {MULT,              ACTION(DEFINED_VALUE,   act_SHIFT,  32)},
    {DIV,               ACTION(DEFINED_VALUE,   act_SHIFT,  33)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_K)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, H_IS_K)}
};

static Entry_of_action_table_for_state actions_for_state_8[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_L)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, K_IS_L)}
};

static Entry_of_action_table_for_state actions_for_state_9[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
```

```
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {DEGREE,            ACTION(DEFINED_VALUE,   act_SHIFT,  34)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, L_IS_M)}
};

static Entry_of_action_table_for_state actions_for_state_10[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {DEGREE,            ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_N)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, M_IS_N)}
};

static Entry_of_action_table_for_state actions_for_state_12[] = {
    {OPENED_BRACKET,    ACTION(DEFINED_VALUE,   act_SHIFT,  14)},
    {kw_ISTINA,         ACTION(DEFINED_VALUE,   act_SHIFT,  15)},
    {kw_LOZH,           ACTION(DEFINED_VALUE,   act_SHIFT,  16)},
    {t_INT,             ACTION(DEFINED_VALUE,   act_SHIFT,  17)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_SHIFT,  14)}
};

static Entry_of_action_table_for_state actions_for_state_15[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
```

```
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {DEGREE,            ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_TRUE)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, N_IS_TRUE)}
};

static Entry_of_action_table_for_state actions_for_state_16[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {DEGREE,            ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_FALSE)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, N_IS_FALSE)}
};

static Entry_of_action_table_for_state actions_for_state_17[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {DEGREE,            ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_INT)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, N_IS_INT)}
};

static Entry_of_action_table_for_state actions_for_state_24[] = {
    {PLUS,              ACTION(DEFINED_VALUE,   act_SHIFT,  12)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_SHIFT,  13)},
    {OPENED_BRACKET,    ACTION(DEFINED_VALUE,   act_SHIFT,  14)},
    {kw_ISTINA,         ACTION(DEFINED_VALUE,   act_SHIFT,  15)},
```

```
    {kw_LOZH,           ACTION(DEFINED_VALUE,   act_SHIFT,  16)},
    {t_INT,             ACTION(DEFINED_VALUE,   act_SHIFT,  17)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_SHIFT,  12)}
};

static Entry_of_action_table_for_state actions_for_state_35[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_NOT_F)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_NOT_F)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_NOT_F)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_NOT_F)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_NOT_F)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_NOT_F)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_NOT_F)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_NOT_F)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, F_IS_NOT_F)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, F_IS_NOT_F)}
};

static Entry_of_action_table_for_state actions_for_state_36[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {DEGREE,            ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_PLUS_N)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, M_IS_PLUS_N)}
};

static Entry_of_action_table_for_state actions_for_state_37[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
```

```
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {DEGREE,            ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, M_IS_MINUS_N)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, M_IS_MINUS_N)}
};

static Entry_of_action_table_for_state actions_for_state_38[] = {
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_SHIFT,  56)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_SHIFT,  56)}
};

static Entry_of_action_table_for_state actions_for_state_39[] = {
    {COLON,             ACTION(DEFINED_VALUE,   act_SHIFT,  57)},
    {NOR,               ACTION(DEFINED_VALUE,   act_SHIFT,  19)},
    {OR,                ACTION(DEFINED_VALUE,   act_SHIFT,  20)},
    {XOR,               ACTION(DEFINED_VALUE,   act_SHIFT,  21)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_SHIFT,  57)}
};

static Entry_of_action_table_for_state actions_for_state_40[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_NOR_E)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_NOR_E)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_NOR_E)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_NOR_E)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_NOR_E)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_NOR_E)},
    {NAND,              ACTION(DEFINED_VALUE,   act_SHIFT,  22)},
    {AND,               ACTION(DEFINED_VALUE,   act_SHIFT,  23)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_NOR_E)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, T_IS_T_NOR_E)}
};

static Entry_of_action_table_for_state actions_for_state_41[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_OR_E)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_OR_E)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_OR_E)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_OR_E)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_OR_E)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_OR_E)},
    {NAND,              ACTION(DEFINED_VALUE,   act_SHIFT,  22)},
    {AND,               ACTION(DEFINED_VALUE,   act_SHIFT,  23)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_OR_E)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, T_IS_T_OR_E)}
};

static Entry_of_action_table_for_state actions_for_state_42[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_XOR_E)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_XOR_E)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_XOR_E)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_XOR_E)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_XOR_E)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_XOR_E)},
    {NAND,              ACTION(DEFINED_VALUE,   act_SHIFT,  22)},
    {AND,               ACTION(DEFINED_VALUE,   act_SHIFT,  23)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, T_IS_T_XOR_E)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, T_IS_T_XOR_E)}
};
```

```
static Entry_of_action_table_for_state actions_for_state_43[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_NAND_F)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_NAND_F)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_NAND_F)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_NAND_F)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_NAND_F)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_NAND_F)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_NAND_F)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_NAND_F)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_NAND_F)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, E_IS_E_NAND_F)}
};

static Entry_of_action_table_for_state actions_for_state_44[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_AND_F)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_AND_F)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_AND_F)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_AND_F)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_AND_F)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_AND_F)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_AND_F)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_AND_F)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, E_IS_E_AND_F)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, E_IS_E_AND_F)}
};

static Entry_of_action_table_for_state actions_for_state_45[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_SHIFT,  30)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_SHIFT,  31)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_EQ_H)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, G_IS_G_EQ_H)}
};

static Entry_of_action_table_for_state actions_for_state_46[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
```

```
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_SHIFT,  30)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_SHIFT,  31)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_NEQ_H)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, G_IS_G_NEQ_H)}
};

static Entry_of_action_table_for_state actions_for_state_47[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_SHIFT,  30)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_SHIFT,  31)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LT_H)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, G_IS_G_LT_H)}
};

static Entry_of_action_table_for_state actions_for_state_48[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_SHIFT,  30)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_SHIFT,  31)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GT_H)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, G_IS_G_GT_H)}
};

static Entry_of_action_table_for_state actions_for_state_49[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
```

```
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_SHIFT,  30)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_SHIFT,  31)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_LEQ_H)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, G_IS_G_LEQ_H)}
};

static Entry_of_action_table_for_state actions_for_state_50[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_SHIFT,  30)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_SHIFT,  31)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, G_IS_G_GEQ_H)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, G_IS_G_GEQ_H)}
};

static Entry_of_action_table_for_state actions_for_state_51[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {MULT,              ACTION(DEFINED_VALUE,   act_SHIFT,  32)},
    {DIV,               ACTION(DEFINED_VALUE,   act_SHIFT,  33)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_PLUS_K)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, H_IS_H_PLUS_K)}
};

static Entry_of_action_table_for_state actions_for_state_52[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
```

```
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {MULT,              ACTION(DEFINED_VALUE,   act_SHIFT,  32)},
    {DIV,               ACTION(DEFINED_VALUE,   act_SHIFT,  33)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, H_IS_H_MINUS_K)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, H_IS_H_MINUS_K)}
};

static Entry_of_action_table_for_state actions_for_state_53[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_MULT_L)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, K_IS_K_MULT_L)}
};

static Entry_of_action_table_for_state actions_for_state_54[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
```

```
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, K_IS_K_DIV_L)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, K_IS_K_DIV_L)}
};

static Entry_of_action_table_for_state actions_for_state_55[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, L_IS_M_DEGREE_L)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, L_IS_M_DEGREE_L)}
};

static Entry_of_action_table_for_state actions_for_state_56[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {CONDITION,         ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {COLON,             ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {NOR,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {OR,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {XOR,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {NAND,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {AND,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {EQ,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {NEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {LT,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {GT,                ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {LEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {GEQ,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {PLUS,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {MINUS,             ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {MULT,              ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {DIV,               ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {DEGREE,            ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, N_IS_OB_S_CB)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, N_IS_OB_S_CB)}
};

static Entry_of_action_table_for_state actions_for_state_58[] = {
    {NONE,              ACTION(DEFINED_VALUE,   act_REDUCE, S_IS_T_COND_T_COLON_T)},
    {NOR,               ACTION(DEFINED_VALUE,   act_SHIFT,  19)},
    {OR,                ACTION(DEFINED_VALUE,   act_SHIFT,  20)},
```

```
    {XOR,               ACTION(DEFINED_VALUE,   act_SHIFT,  21)},
    {CLOSED_BRACKET,    ACTION(DEFINED_VALUE,   act_REDUCE, S_IS_T_COND_T_COLON_T)},
    {ANY,               ACTION(UNDEFINED_VALUE, act_REDUCE, S_IS_T_COND_T_COLON_T)}
};


static const Entry_of_action_table action_table[] = {
    {actions_for_state_0,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 0
    {actions_for_state_1,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_1 )},    // состояние 1
    {actions_for_state_2,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_2 )},    // состояние 2
    {actions_for_state_3,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_3 )},    // состояние 3
    {actions_for_state_4,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_4 )},    // состояние 4
    {actions_for_state_5,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_5 )},    // состояние 5
    {actions_for_state_6,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_6 )},    // состояние 6
    {actions_for_state_7,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_7 )},    // состояние 7
    {actions_for_state_8,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 8
    {actions_for_state_9,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 9

    {actions_for_state_10,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 10
    {actions_for_state_0,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 11
    {actions_for_state_12,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_12 )},   // состояние 12
    {actions_for_state_12,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_12 )},   // состояние 13
    {actions_for_state_0,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 14
    {actions_for_state_15,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_15 )},   // состояние 15
    {actions_for_state_16,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_16 )},   // состояние 16
    {actions_for_state_17,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_17 )},   // состояние 17
    {actions_for_state_0,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 18
    {actions_for_state_0,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 19
    {actions_for_state_0,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 20

    {actions_for_state_0,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 21
    {actions_for_state_0,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 22
    {actions_for_state_0,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0 )},    // состояние 23
    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 24
    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 25
    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 26
    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 27
    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 28
    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 29
    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 30

    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 31
    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 32
    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 33
    {actions_for_state_24,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_24 )},   // состояние 34
    {actions_for_state_35,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_35 )},   // состояние 35
    {actions_for_state_36,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_36)},    // состояние 36
    {actions_for_state_37,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_37)},    // состояние 37
    {actions_for_state_38,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_38)},    // состояние 38
    {actions_for_state_39,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_39)},    // состояние 39
    {actions_for_state_40,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_40)},    // состояние 40

    {actions_for_state_41,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_41 )},   // состояние 41
    {actions_for_state_42,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_42 )},   // состояние 42
    {actions_for_state_43,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_42 )},   // состояние 43
    {actions_for_state_44,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_44 )},   // состояние 44
    {actions_for_state_45,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_45 )},   // состояние 45
    {actions_for_state_46,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_46)},    // состояние 46
    {actions_for_state_47,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_47)},    // состояние 47
```

```
    {actions_for_state_48,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_48)},    // состояние 48
    {actions_for_state_49,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_49)},    // состояние 49
    {actions_for_state_50,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_50)},    // состояние 50

    {actions_for_state_51,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_51 )},   // состояние 51
    {actions_for_state_52,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_52 )},   // состояние 52
    {actions_for_state_53,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_53 )},   // состояние 53
    {actions_for_state_54,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_54 )},   // состояние 54
    {actions_for_state_55,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_55 )},   // состояние 55
    {actions_for_state_56,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_56)},    // состояние 56
    {actions_for_state_0,   ELEMS_IN_TABLE_FOR_STATE(actions_for_state_0)},     // состояние 57
    {actions_for_state_58,  ELEMS_IN_TABLE_FOR_STATE(actions_for_state_58)}     // состояние 58
};
```

В строке состояния поиск правила производится с помощью бинарного поиска.

```
static Action_info search_in_action_table_for_state(
Entry_of_action_table t, LEXEM_CODE lc) {
    Entry_of_action_table_for_state* q = t.ptr_to_action_table_for_state;
    int left = 0;
    int right = t.number_of_actions ;
    int result = t.number_of_actions ;

    while( left <= right )
    {
        int mid = (left + right) / 2;

        if( lc == q[mid].lexem_code)
        {
            result = mid;
            break;
        }

        if( lc > q[mid].lexem_code)
        {
            left = mid + 1;
        }
        else if( lc < q[mid].lexem_code)
        {
            right = mid - 1;
        }
    }
    return unpack_action_code( q[result].action_code);
}

static inline Action_info get_action(int number_of_state, LEXEM_CODE
lc){
    return search_in_action_table_for_state(action_table[number_of_state], lc);
}
```

Таблица Переход

```
Таблица ПЕРЕХОД будет состоять из элементов вида
(указатель на массив записей для нетерминала,количество записей)

Массив записей для нетерминала будет состоять из элементов вида
(текущее состояние, следующее состояние)
При этом в качестве первой компоненты элемента может быть специальное значение ANY,
большее любого номера состояния.
```

```
struct Entry_for_GOTO_table_for_nonterminal {
    unsigned char current_state;
    unsigned char next_state;
};

struct Entry_of_GOTO_table {
    Entry_for_GOTO_table_for_nonterminal* ptr_to_table_for_nonterminal;
    size_t                                number_of_jumps;
};

#define ELEMS_IN_TABLE_FOR_NONTERMINAL(t) \
    (sizeof(t)/sizeof(Entry_for_GOTO_table_for_nonterminal) - 1)

static Entry_for_GOTO_table_for_nonterminal GOTO_for_S[] = {
    {0, 1}, {ANY, 38}
};

static Entry_for_GOTO_table_for_nonterminal GOTO_for_T[] = {
    {18, 39}, {57, 58}, {ANY, 2}
};

static Entry_for_GOTO_table_for_nonterminal GOTO_for_E[] = {
    {19, 40}, {20, 41}, {21, 42}, {ANY, 3}
};

static Entry_for_GOTO_table_for_nonterminal GOTO_for_F[] = {
    {11, 35}, {22, 43}, {23, 44}, {ANY, 4}
};

static Entry_for_GOTO_table_for_nonterminal GOTO_for_G[] = {
    {ANY, 5}
};

static Entry_for_GOTO_table_for_nonterminal GOTO_for_H[] = {
    {24, 45}, {25, 46}, {26, 47}, {27, 48}, {28, 49}, {29, 50}, {ANY, 6}
};

static Entry_for_GOTO_table_for_nonterminal GOTO_for_K[] = {
    {30, 51}, {31, 52}, {ANY, 7}
};

static Entry_for_GOTO_table_for_nonterminal GOTO_for_L[] = {
    {32, 53}, {33, 54}, {34, 55}, {ANY, 8}
};

static Entry_for_GOTO_table_for_nonterminal GOTO_for_M[] = {
    {ANY, 9}
};

static Entry_for_GOTO_table_for_nonterminal GOTO_for_N[] = {
    {12, 36}, {13, 37}, {ANY, 10}
};

static const Entry_of_GOTO_table GOTO_table[] = {
    {GOTO_for_S, ELEMS_IN_TABLE_FOR_NONTERMINAL(GOTO_for_S)},
    {GOTO_for_T, ELEMS_IN_TABLE_FOR_NONTERMINAL(GOTO_for_T)},
    {GOTO_for_E, ELEMS_IN_TABLE_FOR_NONTERMINAL(GOTO_for_E)},
    {GOTO_for_F, ELEMS_IN_TABLE_FOR_NONTERMINAL(GOTO_for_F)},
```

```
    {GOTO_for_G, ELEMS_IN_TABLE_FOR_NONTERMINAL(GOTO_for_G)},
    {GOTO_for_H, ELEMS_IN_TABLE_FOR_NONTERMINAL(GOTO_for_H)},
    {GOTO_for_K, ELEMS_IN_TABLE_FOR_NONTERMINAL(GOTO_for_K)},
    {GOTO_for_L, ELEMS_IN_TABLE_FOR_NONTERMINAL(GOTO_for_L)},
    {GOTO_for_M, ELEMS_IN_TABLE_FOR_NONTERMINAL(GOTO_for_M)},
    {GOTO_for_N, ELEMS_IN_TABLE_FOR_NONTERMINAL(GOTO_for_N)}
};
```

Для поиска в этой таблице реализован последовательный поиск.

```
static int search_in_goto_table_for_nonterminal(Entry_of_GOTO_table
t, unsigned char s){
    int i = 0;
    Entry_for_GOTO_table_for_nonterminal* q = t.ptr_to_table_for_nonterminal;
    while(s > q[i].current_state){
        i++;
    }
    return (q[i].current_state == s) ? q[i].next_state : q[t.number_of_jumps].next_state;
}

static inline int get_next_state(Non_terminal nt, unsigned char
current_state){
    return search_in_goto_table_for_nonterminal(GOTO_table[nt], current_state);
}
```

3.4.2. Описание стека

Данный компилятор использует стек реализованный следуюшим образом.

```
Объявление класса:
#ifndef PARSER_STACK_H #define PARSER_STACK_H

#include <vector> using namespace std;

struct Parser_stack_elem {
    short state;
    short is_undefined;
    int   type;
};

class Parser_stack {
    public:
        Parser_stack() : top_index(0){
            s = vector<Parser_stack_elem>();
        };
        Parser_stack_elem top();
        void pop();
        void push(const Parser_stack_elem& elem);
        void get_elems(Parser_stack_elem result[], size_t number_of_elems);
        void multi_pop(size_t number_of_elems);
        bool empty() const;
        size_t size() const;
        void print_stack() const;
    private:
        vector<Parser_stack_elem> s;
        size_t top_index;
};
```

```
#endif
```

Определение класса:

```
bool Parser_stack :: empty() const{
    return top_index == 0;
}

size_t Parser_stack :: size() const{
    return top_index;
}

Parser_stack_elem Parser_stack :: top(){
    return s[top_index-1];
}

void Parser_stack :: pop(){
    s.pop_back();
    if(top_index != 0){top_index--;};
    return;
}

void Parser_stack :: multi_pop(size_t number_of_elems){
    s.erase(s.end() - number_of_elems,  s.end());
    if(top_index >= number_of_elems){
        top_index -= number_of_elems;
    }
    else{
        top_index = 0;
    }
    return;
}

void Parser_stack :: get_elems(Parser_stack_elem result[], size_t
number_of_elems){
    if(top_index >= number_of_elems){
        size_t temp = top_index - number_of_elems;
        for(size_t i = temp; i < top_index; i++ ){
            result[i - temp] = s[i];
        }
    }
    return;
}

void Parser_stack :: push(const Parser_stack_elem& elem){
    s.push_back(elem);
    top_index++;
    return;
}

void Parser_stack::print_stack() const{
    printf("%s","States and args: ");
    for(size_t i = 0; i < top_index; i++){
        printf("{state=%d, type=%d}",s[i].state, s[i].type);
    }
    printf("%s","\n");
    return;
}
```

Данный класс был написан, поскольку для работы SLR-1 анализатора необходима возможность снимать несколько элементов с вершины стека.

3.4.3. Класс синтаксического анализатора

Текст класса Parser

Объявление класса Parser:

```
#include "parser_stack.h"
#include "scaner.h"
#include "error_count.h"
#include "code_buffer.h"

#ifndef PARSER_H
#define PARSER_H
enum Action_names {act_ERROR, act_REDUCE, act_SHIFT, act_OK};

struct Action_info {
    short        is_undefined;
    short        argument;
    Action_names action;
};

#define MAXIMAL_LENGTH_OF_RULE 5

#define LOGICAL_TYPE        1
#define UNSIGN_INT_TYPE     2

class Parser {
    public:
        void parse();

        void diagnostic_for_reduce(unsigned state);
        void diagnostic_for_shift(unsigned state);

        Parser(const unsigned char* t, Error_count* e, Code_buffer* cg);
        ~Parser();
    private:
        Parser_stack*       stack;
        Scaner*             scan;
        Error_count*        ec;
        Code_buffer*        gen;
        Action_info         ai;
        returned_lexem      lc;

        Parser_stack_elem  body[MAXIMAL_LENGTH_OF_RULE];

        void shift(short shifted_state, short is_undef);
        void reduce(short rule_for_reduce, short is_undef);
};
#endif
```

Описание класса Parser:

```
#include <stdio.h> #include <stdlib.h> #include "parser.h"

Parser :: ~Parser(){
    delete stack;
```

```
    delete scan;
    return;
}

Parser :: Parser(const unsigned char* t, Error_count* e,
Code_buffer* cg){
    scan  = new Scaner(t,e);
    stack = new Parser_stack();
    gen   = cg;
    ec    = e;
    return;
}
```

Основной метод класса Parser

Метод parse() является главным циклом компилятора.

```
void Parser::parse(){
    Parser_stack_elem  initial_elem;
    initial_elem.state        = 0;
    initial_elem.is_undefined = 0;
    initial_elem.type         = 0;

    stack->push(initial_elem);

    for( ; ; ){
        lc = scan->get_current_lexem();
        printf("lc = %s\n", lexem_names_as_strings[lc.probably_code]);
        ai = get_action((stack->top()).state, lc.probably_code);
        switch(ai.action){
            case act_REDUCE:
                reduce(ai.argument, ai.is_undefined);
                scan->back();
                break;

            case act_SHIFT:
                shift(ai.argument, ai.is_undefined);
                break;

            case act_OK:
                Intermediate_command command;
                command.op  = op_STOP;
                gen->append(&command);
                gen->writeToFile();
                return;

            default:
                ;
        }
    }
    return;
}
```

Здесь происходит вызов лексического сканера, поиск в таблице синтаксического анализа и выбор последующего действия.

При выборе действия "свертка"вызывается метод сдвигающий лексический сканер на предыдущую лексему.

При успешном завершении компиляции здесь в конец вектора с промежуточным кодом записывается команда op_STOP и вызывается запись промежуточного кода в файл.

Метод Переход

В методе shift на вершину стека помещается новый элемент, с номером состояния куда будет производится переход. В этом методе при обнаружении признака перехода по ANY вызывается диагностика.

```
void Parser::shift(short shifted_state, short is_undef){
    if(is_undef){
        diagnostic_for_shift(shifted_state);
    }
    Parser_stack_elem    pushed_elem;
    pushed_elem.state        = shifted_state;
    pushed_elem.is_undefined = is_undef;
    pushed_elem.type         = 0;
    stack->push(pushed_elem);
    return;
}
void Parser::diagnostic_for_shift( unsigned state) {
    switch(state)
    {
    case 11:
        {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
               scan->get_line_number(), "!, +, -, (, истина, ложь, целое");
            break;
        }
    case 12:
        {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
               scan->get_line_number(), "+, -, (, истина, ложь, целое");
            break;
        }
    case 14:
        {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
               scan->get_line_number(), "(, истина, ложь, целое");
            break;
        }
    case 56:
        {   printf("В строке %d ожидается: ).\n",
               scan->get_line_number());
            break;
        }
    case 57:
        {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
               scan->get_line_number(), ":, !||, ||, ^^");
            break;
        }
    }
    ec->increment_number_of_errors();
    return;
}
```

Метод Свертка

Данный метод получает на вход порядковый номер правила, по которому производится свертка и признак свертки по неопределенной лексеме. Если свертка производится по неопределенной лексеме, вызывается диагностика. Далее с вершины стека удаляется столько элементов сколько записано в ячейке массива rules для правила, по которому производится свертка. Затем на вершину стека добавляется новый элемент, содержащий новое текущее состояние стека.

```
enum Non_terminal {nt_S, nt_T, nt_E, nt_F, nt_G, nt_H, nt_K, nt_L,
nt_M, nt_N};
```

```
enum Rule_names {
    S_IS_T_COND_T_COLON_T,  S_IS_T,
    T_IS_T_NOR_E,           T_IS_T_OR_E,        T_IS_T_XOR_E,   T_IS_E,
    E_IS_E_NAND_F,          E_IS_E_AND_F,       E_IS_F,
    F_IS_NOT_F,             F_IS_G,
    G_IS_G_EQ_H,            G_IS_G_NEQ_H,       G_IS_G_LT_H,
    G_IS_G_GT_H,            G_IS_G_LEQ_H,       G_IS_G_GEQ_H,
    G_IS_H,
    H_IS_H_PLUS_K,          H_IS_H_MINUS_K,     H_IS_K,
    K_IS_K_MULT_L,          K_IS_K_DIV_L,       K_IS_L,
    L_IS_M_DEGREE_L,        L_IS_M,
    M_IS_PLUS_N,            M_IS_MINUS_N,       M_IS_N,
    N_IS_OB_S_CB,           N_IS_TRUE,          N_IS_FALSE,     N_IS_INT
};

struct Rule_info {
    char non_terminal;
    char length;
};

Rule_info rules[] = {
    {nt_S, 5}, {nt_S, 1},
    {nt_T, 3}, {nt_T, 3}, {nt_T, 3}, {nt_T, 1},
    {nt_E, 3}, {nt_E, 3}, {nt_E, 1},
    {nt_F, 2}, {nt_F, 1},
    {nt_G, 3}, {nt_G, 3}, {nt_G, 3}, {nt_G, 3}, {nt_G, 3}, {nt_G, 3}, {nt_G, 1},
    {nt_H, 3}, {nt_H, 3}, {nt_H, 1},
    {nt_K, 3}, {nt_K, 3}, {nt_K, 1},
    {nt_L, 3}, {nt_L, 1},
    {nt_M, 2}, {nt_M, 2}, {nt_M, 1},
    {nt_N, 3}, {nt_N, 1}, {nt_N, 1}, {nt_N, 1}
};

void Parser::reduce(short rule_for_reduce, short is_undef) {
    size_t               length_of_rule;
    Parser_stack_elem    pushed_elem;
    Rule_info            ri;
    Intermediate_command command;

    if(is_undef){
        diagnostic_for_reduce((stack->top()).state);
    }

    ri = rules[rule_for_reduce];
    length_of_rule = ri.length;
    stack->get_elems(body, length_of_rule);
    stack->multi_pop(length_of_rule);

    pushed_elem.is_undefined = is_undef;
    pushed_elem.state        =
        get_next_state(static_cast<Non_terminal>(ri.non_terminal), (stack->top()).state);

    switch(rule_for_reduce)
    {
        case S_IS_T: case T_IS_E: case E_IS_F: case F_IS_G: case G_IS_H:
        case H_IS_K: case K_IS_L: case L_IS_M: case M_IS_N: case M_IS_PLUS_N:
        case N_IS_OB_S_CB:
            pushed_elem.type = body[0].type;
```

```
            stack->push(pushed_elem);
            return;
        case S_IS_T_COND_T_COLON_T:
            command.op      = op_CONDITION;
            break;
        case T_IS_T_NOR_E: case T_IS_T_OR_E: case T_IS_T_XOR_E:
            command.op      = op_NOR + static_cast<Instruction_name>(rule_for_reduce
                    - (int)T_IS_T_NOR_E);
            break;
        case E_IS_E_NAND_F: case E_IS_E_AND_F:
            command.op      = op_NAND + static_cast<Instruction_name>(rule_for_reduce
                    - (int)E_IS_E_NAND_F);
            break;
        case F_IS_NOT_F:
            command.op      = op_NOT;
            break;
        case G_IS_G_EQ_H: case G_IS_G_NEQ_H: case G_IS_G_LT_H:
        case G_IS_G_GT_H: case G_IS_G_LEQ_H: case G_IS_G_GEQ_H:
            command.op      = op_EQ + static_cast<Instruction_name>(rule_for_reduce
                    - (int)G_IS_G_EQ_H);
            break;
        case H_IS_H_PLUS_K: case H_IS_H_MINUS_K:
            command.op      = op_ADD + static_cast<Instruction_name>(rule_for_reduce
                    - (int)H_IS_H_PLUS_K);
            break;
        case K_IS_K_MULT_L: case K_IS_K_DIV_L:
            command.op      = op_MULT + static_cast<Instruction_name>(rule_for_reduce
                    - (int)K_IS_K_MULT_L);
            break;
        case L_IS_M_DEGREE_L:
            command.op      = op_POWER;
            break;
        case M_IS_MINUS_N:
            command.op      = op_CHANGE_SIGN;
            break;
        case N_IS_TRUE:
            pushed_elem.type        = LOGICAL_TYPE;
            command.op              = op_LOAD_TRUE;
            command.number_const    = 1;
            break;
        case N_IS_FALSE:
            pushed_elem.type        = LOGICAL_TYPE;
            command.op              = op_LOAD_FALSE;
            command.number_const    = 0;
            break;
        case N_IS_INT:
            pushed_elem.type        = UNSIGN_INT_TYPE;
            command.op              = op_LOAD_INT_CONST;
            command.number_const    = lc.number;
            break;
        default:
            return;
    }
    gen->append(&command);
    stack->push(pushed_elem);
    return;
}

void Parser::diagnostic_for_reduce( unsigned state) {
```

```
    if(state == 2)
    {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
           scan->get_line_number(), "?, !||, ||, ^^");
    }
    else if( (state > 2 && state < 5) ||
             (state > 34 && state < 38) ||
             (state > 38 && state < 42) ||
             (state > 42 && state < 45))
    {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
           scan->get_line_number(), "?, :, !||, ||, ^^, !&&, &&");
    }
    else if( state == 5)
    {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
           scan->get_line_number(), "?, :, !||, ||, ^^, !&&, &&, =,
                !=, <, >, <=, >=");
    }
    else if( state == 6 || (state > 44 && state < 51))
    {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
           scan->get_line_number(), "?, :, !||, ||, ^^, !&&, &&, =,
                !=, <, >, <=, >=, +, -");
    }
    else if( (state > 6 && state < 9) || (state > 50 && state < 56))
    {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
           scan->get_line_number(), "?, :, !||, ||, ^^, !&&, &&, =,
                !=, <, >, <=, >=, +, -, *, /");
    }
    else if( (state > 8 && state < 11) || (state > 14 && state < 18) || (state == 56))
    {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
           scan->get_line_number(), "?, :, !||, ||, ^^, !&&, &&, =,
                !=, <, >, <=, >=, +, -, *, /, **");
    }
    else if(state == 58)
    {   printf("В строке %d ожидается что-либо из следующего: %s.\n",
           scan->get_line_number(), "!||, ||, ^^");
    }
    ec->increment_number_of_errors();
    return;
}
```

3.4.4. Генерация промежуточного кода

За хранение и записать в файл промежуточного кода отвечает класс Code_buffer. Он содержит в себе STL::vector с элементами типа Intermediate_command.

Далее описание данного класса:

```
#ifndef CODE_BUFFER_H #define CODE_BUFFER_H

#include <vector> using namespace std;

Имена команд поддерживаемых компилятором.

enum Instruction_name {
    op_NOR,         op_OR,                  op_XOR,
    op_NAND,        op_AND,                 op_NOT,
    op_EQ,          op_NEQ,                 op_SIGN_LT,
    op_SIGN_GT,     op_SIGN_LEQ,            op_SIGN_GEQ,
    op_UNSIGN_LT,   op_UNSIGN_GT,           op_UNSIGN_LEQ,
```

```
    op_UNSIGN_GEQ,  op_ADD,                 op_SUB,
    op_MULT,        op_DIV,                 op_CHANGE_SIGN,
    op_POWER,       op_LOAD_INT_CONST,      op_LOAD_INT_SIGN_CONST,
    op_LOAD_TRUE,   op_LOAD_FALSE,          op_LOG_TO_INT,
    op_CONDITION,   op_STOP
};

struct Intermediate_command {
    Instruction_name op;
    __int64 number_const;
};

class Code_buffer {
    public:
        Code_buffer() : current_operation_index(0)
        {
            code = vector<Intermediate_command>();
        }

        unsigned append(Intermediate_command* c);
        void writeToFile();

    private:

        unsigned current_temp_var;

        unsigned current_operation_index;
        vector<Intermediate_command> code;
}; #endif

#include <stdio.h>
#include "code_buffer.h"
#include <stdlib.h>

unsigned Code_buffer::append(Intermediate_command* c) {
    code.push_back(*c);
    current_operation_index++;

    printf("Current intermediate code command: {op=%s, type=%d, %d}\n",
           operation_names_as_strings[c->op]);

    return current_operation_index;
}

void Code_buffer::writeToFile() {
    FILE* pFile;
    pFile = fopen("file.mbin", "wb");

    int s = current_operation_index * sizeof(Intermediate_command);
    char *p = new char[s];
    char *q = p;

    for(int i = 0; i < code.size(); i++)
    {
        *q = code[i].op;
        printf("code command: {op=%s, %d}\n",
           operation_names_as_strings[code[i].op], code[i].number_const);
        q++;
        if(code[i].op == op_LOAD_INT_CONST)
```

```
        {
            *((__int64*)q) = code[i].number_const;
            q += sizeof(__int64);
        }
    }

    fwrite(&p, sizeof(char), s, pFile);

    fclose(pFile);
}
```

Основная генерация промежуточного кода идет из блока switch(rule_for_reduce) метода Parser::reduce. По всем правилам кроме правил длиной 1 в буффер промежуточного кода добавляется новый элемент. Этот элемент содержит номер операции, которую нужно произвести и, для команд загрузки, целое число.

Литература

[1] *А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман* Компиляторы: принципы, технологии и инструментарий. 2-е изд. Пер. с англ. М.: Вильямс. 2008.

[2] *Г. С. Уоррен.* Алгоритмические трюки для программистов. Пер. с англ. М.: Вильямс, 2003.

[3] *М. В. Мозговой.* Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход. СПб.: Наука и Техника. 2006.

[4] *Д. Кнут* Искусство программирования. Т.2. Получисленные алгоритмы. 3-е изд. Пер. с англ. М.: Вильямс. 2004.