

ЯЗЫК ПРОГРАММИРОВАНИЯ АЛГОЛ 68***В. В. Броль, А. А. Красилов, А. Н. Маслов*****ВВЕДЕНИЕ**

Важным этапом 20-летнего развития языков программирования является публикация АЛГОЛа 68. Алгоритмический язык АЛГОЛ 68 разработан Рабочей группой 2.1. по АЛГОЛу Международной федерации по обработке информации (ИФИП) «для распространения алгоритмов, для эффективного их выполнения на самых различных вычислительных машинах и как средство для изучения алгоритмов».

АЛГОЛ 68 позволяет кратко и естественно записывать алгоритмы, относящиеся к различным областям. Компиляторы с АЛГОЛа 68 создают достаточно эффективные рабочие программы [2, 68, 81].

Строгое определение семантики, явное выделение всех зависимостей от конкретной ЭВМ и стандарт на машинное представление позволяют переносить программы на АЛГОЛе 68 между разнотипными ЭВМ (см. § 14).

АЛГОЛ 68 свел воедино подавляющее большинство идей современного программирования. Использование контроля видов (§ 2, 18) значительно упрощает отладку программ на АЛГОЛе 68 и повышает их надежность.

Разработка АЛГОЛа 68 и проблемы его реализации стимулировали проведение ряда теоретических исследований (§14, 18, 19). Многие из возникших проблем потребовали привлечения мощного математического аппарата и в значительной степени определили развитие информатики как самостоятельной отрасли науки. Показательно, что известная монография [6] использует АЛГОЛ 68 как базовый язык.

Имеющееся мнение о трудности изучения АЛГОЛа 68 в известной мере справедливо по отношению к его формальному описанию, но не к учебно-методическим материалам [30, 155, 162, 188]. Опыт преподавания АЛГОЛа 68 в Московском, Ленинградском и Новосибирском университетах, в Московском

физико-техническом институте и Московском институте электронного машиностроения показывает, что в его изучении нет трудностей.

АЛГОЛ 68 преподается во многих университетах Англии, ФРГ, Голландии и других стран Западной Европы, а также в некоторых университетах США и Канады.

В данном обзоре содержится краткое описание АЛГОЛа 68 (согласно сообщению [240], называемому в дальнейшем «RR»; сообщение о первоначальном варианте АЛГОЛа 68 [239] называется «R»), его русского варианта ([34], называемого «ПС») и его машинного представления ([118], называемого «SHR»), а также сопоставление материалов, предшествовавших появлению указанных документов, и статей, содержащих критику на них, и дальнейшие исследования. Рассматриваются также методы реализации АЛГОЛа 68 и вопросы, находящиеся в стадии обсуждения:

- разработка стандарта на машинное представление программ на национальных вариантах АЛГОЛа 68,
- разработка метода переносимости программ на АЛГОЛе 68,
- описание подязыков и надязыков АЛГОЛа 68,
- перспективы использования АЛГОЛа 68.

Относительно статуса документов RR, SHR и ПС см. § 10.

Часть 1

ОСНОВНЫЕ ПОНЯТИЯ АЛГОЛА 68

§ 1. ПРЕДВАРИТЕЛЬНЫЕ ОПРЕДЕЛЕНИЯ

Существенная часть определений АЛГОЛа 68 связана с понятием дерева. Формально говоря, дерево представляет собой множество вершин и множество пар вершин, называемых стрелками, такое, что: 1) транзитивное замыкание стрелок является отношением (обозначается $<$) частичного порядка; 2) в этом отношении порядка имеется минимальный элемент (называемый корнем); 3) если $b < a$ и $c < a$, то либо $b < c$, либо $b = c$, либо $c < b$.

У рассматриваемых нами деревьев может быть бесконечное число вершин, но из каждой вершины будет исходить только конечное число стрелок.

Вершины, в которые ведут стрелки из некоторой вершины x , называются наследниками x . Вершина x называется предшествующей для своих наследников. На множестве наследников вершины может быть задано отношение линейного порядка «быть левее»; в таком случае вершина называется фиксирован-

ной. Если отношение «быть левее» не задано, то вершина называется коммутативной.

Дерево называется размеченным элементами множества V , если задано отображение вершин дерева в множество V .

Последовательность вершин a_1, \dots, a_n называется ветвью, если каждая вершина a_i предшествует вершине a_{i+1} .

Чтобы объяснить вычисления, задаваемые программой на АЛГОЛе 68, введем некоторую идеализированную вычислительную машину (сокращенно ИВМ). Относительно каждого конструкта АЛГОЛа 68 будет объяснено, какие действия этот конструкт предписывает ИВМ. Эквивалентные действия может совершить и другая машина. ИВМ принципиально отличается от реальных ЭВМ лишь потенциальной бесконечностью памяти.

Обычно ЭВМ (модель фон Неймана) оперирует с некоторыми классами значений, которые она может преобразовывать или хранить в памяти. Память представляет собой последовательность ячеек, в которых могут храниться числа и, в частности, номера других ячеек памяти. Машина может по номерам ячеек читать хранящиеся в них значения и совершать с ними действия, а также переписывать содержимое одной ячейки в другую. Однако, при одновременном счете нескольких задач можно, обратившись по случайному адресу, испортить информацию в чужой программе. Чтобы этого не случилось, в АЛГОЛ 68 введено понятие «имя вида». Можно затребовать у машины (точнее, у операционной системы машины) выделение памяти под значение вида, и тогда машина сообщит имя вида, т. е. местоположение памяти (в удобном для машины месте). В АЛГОЛе 68 нет средств изменить имя. В какой-то момент можно отказаться от памяти. При этом необходимо контролировать, чтобы имя также пропало (иначе по этому имени можно будет использовать память, отданную другой задаче). Для такого контроля вводится понятие «область действия».

Память ИВМ устроена в виде дерева. Мы ее рассмотрим ниже, а сейчас укажем только как устроена одна ячейка, чтобы затем описать, какие имеются виды значений в языке.

Ячейка состоит из вида ячейки, области действия ячейки и хранимого в ячейке значения. Вид ячейки определяет, что можно делать со значением в этой ячейке и к какому множеству принадлежит это значение (например, к множеству целых чисел, по абсолютной величине меньших некоторого). Вид ячейки неизменен. Значение, хранимое в ячейке, является элементом этого множества, определяемого видом, и может изменяться. Область действия ячейки определяет продолжительность ее существования. Имя не должно существовать дольше, чем ячейка, которую оно именуется.

Процесс вычисления представляет собой некоторое дерево. Область действия — это номер вершины дерева вычислений. В каждый момент исполнения программы имеется некоторая

текущая область действия — номер активной вершины. В начальный момент область действия нулевая. При вызове процедуры создается новая активная вершина в дереве вычислений. По окончании вызова уничтожаются активная вершина и, как следствие, все ячейки с текущей областью действия. Программа может потребовать создания новых ячеек с текущей областью действия, с нулевой областью действия или с областью действия некоторого данного имени (если создается подымя).

ИВМ оперирует со значениями. Каждое значение имеет некоторый вид. Значение может быть записано в ячейку, если известно имя ячейки и вид значения соответствует виду ячейки.

§ 2. ВИДЫ ЗНАЧЕНИЙ

Значения некоторых видов могут быть изображены в программе. Эти виды указаны в таблице 1.

Таблица 1


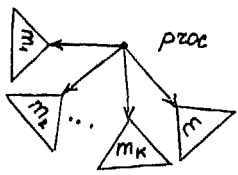
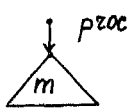

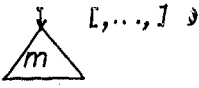

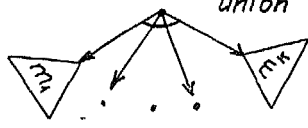
Виды, значения которых могут быть изображены

№№	Обозначение вида	Название значения этого вида	Примеры изображения значений этого вида
1.	<code>int</code>	целое	765
2.	<code>long int</code>	длинное целое	<code>long 765</code>
3.	<code>short int</code>	короткое целое	<code>short 765</code>
4.	<code>real</code>	вещественное	<code>1.23e-41.23 4e+8 2e3</code>
5.	<code>long real</code>	длинное вещественное	<code>long 1.23 long 1.23e-6</code>
6.	<code>short real</code>	короткое вещественное	<code>short 1.23e+4 short 2e3</code>
7.	<code>bits</code>	битовое	<code>2r1011 4r320 8r765 16r8b</code>
8.	<code>long bits</code>	длинное битовое	<code>long 2r1011 long 16r15abf</code>
9.	<code>short bits</code>	короткое битовое	<code>short 16red10 short 8r76</code>
10.	<code>bool</code>	логическое	<code>true</code> (истина) <code>false</code> (ложь)
11.	<code>char</code>	литерное	<code>"a" "1" "+" "a" "s"</code>
12.	<code>string</code>	строка	<code>"" "a" "ab1&" "1""a1"</code>
13.	<code>void</code>	пустое значение	<code>empty</code> (пустое)

Следует отметить, что каждому изображению соответствует только один вид и что в программе никакие два изображения не могут стоять рядом (ничем не разделенные).

Символы `long` и `short` увеличивают или уменьшают количество разрядов, отводимых под значение. Имеются также виды `long long int`, `short short real` и т. д. и изображения для них. С некоторого момента добавление еще одного символа `short` или еще одного символа `long` не изменяет количества разрядов, отводимых под значение, но виды с разным числом приставок считаются различными.

Значение вида `bits` (битовое) это массив значений вида `bool`, упакованный в одну ячейку.

№№ пр-л	Описатель вида и название вида	Дерево вида
1.	ref m имя вида m	
2.	proc $(m_1, \dots, m_k)m$ процедура с параметрами видов m_1, \dots, m_k , вырабатывающая значение вида m (возможно, что $m = \text{void}$).	
2а.	proc m процедура без параметров вырабатывает значение вида m	
3.	struct $(m_1 a_1, \dots, m_k a_k)$ структура с полями видов m_i и с указателями полей a_i	
4.	$[\dots] m$ массив из элементов вида m	
5.	flex $[\dots] m$ подвижный массив из элементов вида m	
6.	union (m_1, \dots, m_k) объединение представителей (возможно, что $m_i = \text{void}$)	 <p data-bbox="532 1364 957 1444">Особенностью этого дерева, в отличие от предыдущих, является перестановочность стрелок в корне дерева.</p>

Результат процедуры, не вырабатывающей значения, обозначается видом **void**.

Имеются также слоговые виды **bytes**, **long bytes** и **short bytes** (т. е. массивы литер, упакованные в одну ячейку) и виды **compl**, **long compl** и **short compl** (комплексное число, состоящее из двух вещественных). Использование символов **long** и **short** как выше.

Рассмотренные виды называются стандартными. Кроме того можно образовывать описатели более сложных видов, используя указанные в табл. 2 правила 1-6. Каждому описателю вида соответствует (растущее вниз) размеченное дерево. Для стандартных описателей пока можно считать, что дерево состоит из одной вершины, помеченной этим описателем.

Описатели видов могут строиться путем применения правил 1-6 в любом порядке, исключая некоторые естественные ограничения.

Сделаем последний шаг в определении описателя вида. Описатели, которые строятся правилами 1-6, называются виртуальными. Виртуальными описателями описываются виды ячеек и виды областей памяти. Имеются два других типа описателей. Если в построении описателя не применялось правило 5, за которым не следовало бы правило 1, то описатель называется формальным. Таким образом **ref flex[]m** может считаться формальным, а **struct(..., flex[]reala, ...)** не может. Если при применении правила 4 или 5, после которого не будет примениться правило 1, в квадратных скобках для каждой размерности указать граничные пары, (граничная пара — это пара разделенных двоеточием конструкторов, вырабатывающих целые, например, $x+1 : (inty; y : =10)$), то такой описатель будет называться фактическим. Граничную пару $l : m$ можно заменять на m (т. е. при отсутствии первой границы считается, что она равна 1).

Фактический описатель описывает не только вид, но и фактический размер памяти, отведенной под значение. Фактический или виртуальный описатель можно формализовать, устраняя границы и мешающие символы **flex**. Фактический описатель путем устранения границ можно виртуализовать.

Фактический описатель используется для задания заявки на создание области памяти. Формальный описатель используется для указания видов значений, которые могут получиться в результате исполнения конструктора. Формальные описатели, не начинающиеся на **union**, описывают виды собственно значений.

Виртуальный описатель играет вспомогательную роль: он входит в фактические и в формальные описатели под символом **ref** (чтобы характеризовать именуемую область памяти).

Замечание. В любом описателе подописатель, стоящий после **ref**, виртуальный, а подописатель, являющийся представителем объединения, параметром или результатом процедуры, всегда

формальный. Поле структуры и элемент массива сохраняет тип объемлющего описателя.

Вид может быть задан посредством описателя или посредством индикатора. Индикатор — это выделенное (шрифтом, подчеркиванием или другим способом) слово. Индикатор вида должен быть описан посредством описания вида:

`mode<индикатор> = <фактический описатель>`.

Каждый индикатор вида задает некоторый вид, сам является описателем вида и может участвовать в образовании новых описателей. Например, если `i` — это индикатор вида с описателем `x`, то `ref i` — описатель, эквивалентный описателю `ref x`. Если индикатор стоит в позиции, где требуется формальный или виртуальный описатель, то он формализуется или виртуализуется соответственно (синтаксис устроен таким образом, что про каждое вхождение описателя можно сказать, какого оно типа).

Однако индикатор вида это не просто сокращенная запись для описателя. Индикатор вида может участвовать в собственном определении, например:

`mode библиотека = struct(string текст, ref библиотека продолжение)`.

Без использования описаний видов этот вполне осмысленный вид (текст и ссылка на продолжение библиотеки) был бы неопределим. В последний текст «библиотеки» можно включить ссылку на начало (на всю библиотеку), либо же поставить «псевдоимя».

Конечно, описание

`mode i=i`

некорректно. Определим класс корректных описаний вида. Для определения корректности удобно использовать деревья, соответствующие описателям.

Дерево описателя вида, построенное с использованием правил 1-6, будем называть сокращенным деревом. Сокращенное дерево имеет конечное число вершин. Вершины, из которых не исходит стрелок, помечены стандартными описателями или индикаторами, а остальные вершины помечены символами `ref`, `struct`, `proc`, `flex`, `[]` или `union`.

Будем подставлять в сокращенное дерево описателя вместо каждого индикатора описатель этого индикатора. Возможно, что описатель индикатора каждый раз будет оказываться индикатором. Ясно, что такое описание некорректно.

Пусть указанная некорректность не имеет места. Тогда каждая подстановка описателя (не являющегося индикатором) вместо индикатора уменьшает число индикаторов на данном расстоянии от корня. Таким образом каждому описателю вида соответствует дерево, быть может бесконечное (как, например, в случае вида `библиотека`), у которого вершины помечены не индикаторами. Помеченное дерево будем называть деревом вида.

Деревья, которые получались в процессе построения дерева вида из сокращенного дерева, будем называть промежуточными деревьями.

Вхождение индикатора в промежуточное дерево описателя этого же индикатора назовем прикрытым, если на ветви дерева от корня до данного вхождения этого индикатора имеется либо символ **proc**, из которого исходит более одной стрелки, либо в любом порядке пара символов, один из которых **proc** или **ref**, а другой **struct**. Описание вида некорректно, если его индикатор не прикрыт в некотором промежуточном дереве его описателя. В дальнейшем будет введено еще одно ограничение на описатели, связанное с представителями объединения.

Некорректные описатели запрещены, чтобы не было значений, требующих бесконечной памяти, и не допустить двусмысленности в приведениях (см. § 3).

Вид — это класс описателей вида. Два описателя вида эквивалентны (принадлежат одному классу), если их деревья совпадают с точностью до перестановки коммутативных стрелок. Это определение не конструктивно. Алгоритм, устанавливающий эквивалентность или неэквивалентность двух описателей, обсуждается в § 18.

§ 3. ПРИВЕДЕНИЯ

Если некоторый конструктор (см. § 15) вырабатывает значение, вид которого не подходит для дальнейших вычислений с этим значением, то ИВМ может его преобразовать в значение приемлемого вида (в терминологии АЛГОЛа 68 — сделать приведение).

Имеется шесть элементарных приведений.

1. **Распроцедуривание.** ИВМ, получив процедуру (без параметров) вида **proc m**, вызывает ее и получает в результате вызова значение вида **m**.

2. **Разыменование.** ИВМ, получив имя вида **ref m**, считывает значение (вида **m**, причем вид **m** формализуется), на которое указывает это имя.

3. **Объединение.** Вид **m** может быть приведен к виду с описателем **union (m₁, ..., m_k)**, если некоторый вид **m_i** эквивалентен **m**. Возможны два случая. Если вид **m** начинается с **union**, то никаких существенных действий ИВМ не делает (такой случай назовем дообъединением), в противном случае отводится память, подходящая для хранения данного значения вида **m**. В эту память записывается данное значение, а ИВМ вместо записанного значения получает (скрытое от программиста) имя памяти вида **ref m**.

4. **Опустошение.** Любой вид **m** может быть заменен видом **void**. Это означает, что ИВМ уничтожает полученное значение.

5. **Обобщения.** Вид **real** приводится к виду **comp1**, вид **int** к виду **real** или к виду **comp1**, вид **bits** к виду **[]bool** и вид **bytes**

к виду [] *char*. При обобщении *real*→*compl* вещественного *x* создается комплексное (*x*, 0). При обобщении *int*→*real* создается вещественное, равное данному целому. Обобщение *int*→*compl* это *int*→*real*→*compl*. При обобщении битовых и слоговых значений происходит распаковка (элементы, запакованные в одну ячейку, рассылаются в специально создаваемый массив). Аналогичные обобщения выполняются для длинных и коротких модификаций.

6. Векторизация. Вид *m* приводится к виду $[\phi 1 : 1 \phi] m$, вид *ref m* к виду *ref* $[\phi 1 : 1 \phi] m$, вид *ref flex [z]m* к виду *ref* $[\phi 1 : 1 \phi, z] m$, вид $[z] m$ к виду $[\phi 1 : 1 \phi, z] m$ и вид *ref [z]m* к виду *ref* $[\phi 1 : 1 \phi, z] m$. В квадратных скобках написаны комментарии, чтобы указать на пределы изменения массива по новой координате (во всех случаях от 1 до 1).

В результате векторизации ИВМ получает новый массив или новое имя массива. Однако если в памяти имеется копия старого имени, то это старое имя именуется массив, состоящий из тех же элементов, что и имя, полученное в результате векторизации. Поэтому в случае векторизации: *ref flex [z]m*→*ref* $[\phi 1 : 1 \phi, z] m$ происходит отбрасывание *flex*, а полученное имя является временным. Это имя должно уничтожиться раньше, чем имя, из которого оно получилось.

Обозначим буквами *p* — распроецирование, *r* — разыменовывание, *u* — объединение, *m* — векторизацию, *w* — обобщение и *v* — опустошение.

Каждый конструкт языка находится в некоторой позиции, которая определяется объемлющим конструктом. Имеются пять разновидностей позиций: *k1* (мягкая), *k2* (слабая), *k3* (раскрытая), *k4* (крепкая), *k5* (сильная). Позиция определяет, какие приведения можно делать над выработанным значением.

Последовательность приведений можно рассматривать как слово из букв *p*, *r*, *m*, *u*, *w*, *v*. Для задания слов, которые обозначают допустимые последовательности приведений, используем операции Клини [41]: U, · и *. Через ϵ обозначим пустое слово.

Пусть *proc 0* это начало описателя без параметров (*proc 0* — условное обозначение).

В первых трех позициях вид, к которому надо приводить исходный вид *m*, неизвестен, но он определяется по виду *m*.

Позиция *k1*. Последовательность приведений — любое слово из $\{p\}^*$, которое приводит *m* к виду *m*₁, не начинающемуся с *proc 0*.

Позиция *k2*. Последовательность приведений — любое слово из $(\{p\} \cup \{r\})^*$, которое приводит *m* к виду *m*₁, не начинающемуся с *ref proc 0* или *ref ref* или *proc 0*.

Позиция *k3*. Последовательность приведений — любое слово из $(\{p\} \cup \{r\})^*$, которое приводит *m* к виду *m*₁, не начинающемуся с *ref* или *proc 0*.

В последующих позициях по объемлющему конструкту необходимо определить вид m , к которому нужно привести исходный вид m . Синтаксис АЛГОЛа 68 устроен так, что это всегда можно сделать.

Позиция k4. Последовательность приведений — любое слово из $(\{p\} \cup \{r\})^* \cdot (\{u\} \cup \{e\})$.

Позиция k5. Последовательность приведений — любое слово из $(\{p\} \cup \{r\})^* \cdot (\{u\} \cup \{w\} \cup \{e\}) \cdot \{m\}^* \cup \tilde{v}$. Здесь $\tilde{v} = \{\tilde{v}\}$ или $(\{p\} \cup \{r\})^* \cdot p \cdot \{v\}$ в зависимости от того, какой исполнялся конструкт. Основы (так называются приводимые конструкты) делятся на раскрываемые и прямые. Для прямых $v = u$, а для раскрываемых $\tilde{v} = (\{p\} \cup \{r\})^* \cdot p \cdot v$, причем слово из $(\{p\} \cup \{r\})^* \cdot p$ приводит m к виду, не начинающемуся с **proc 0**. Последнее различие в методе опустошения нарушает гармонию схемы приведений, но необходимо для того, чтобы иметь возможность выполнить часть возможных распроедуриваний, а затем сделать опустошение. К раскрываемым относятся (см. табл. 3) идентификатор, выборка, вырезка, вызов, формула и текст процедуры (который может распроедуриваться). Все остальные основы относятся к прямым.

Отметим, что если бы описания вида

mode i1 = ref i1 или mode i2 = proc i2

были бы корректными, то приведение вида $i1$ к $i1$ и вида $i2$ к $i2$ в позициях $k4$ или $k5$ было бы двусмысленным, так как можно было бы сделать любое число разыменований или распроедуриваний соответственно.

В [44] доказано, что имеющаяся в АЛГОЛе 68 система приведений не имеет двусмысленностей.

§ 4. ЭЛЕМЕНТАРНЫЕ ДЕЙСТВИЯ ИВМ

ИВМ имеет дело с деревом программы, деревом вычислений и ячейками памяти.

Программу можно представить как дерево. Выпишем конструкты блока или процедуры в линию, но вместо вложенных блоков или текстов процедур нарисуем направленные вверх стрелки, над каждой из которых расположена линия конструкта вытесненного блока или процедуры, в которых также произведено вытеснение блоков и процедур. Получится дерево программы.

Смысл разделения программы по вершинам дерева состоит в том, что если нужно найти описание некоторого идентификатора (или индикатора), входящего в часть программы при некоторой вершине дерева, то оно ищется в этой части программы, затем, если его там нет, поиск продолжается в части программы, соответствующей предшествующей вершине и т. д.

В процессе вычисления по дереву программы строится де-

рево вычислений. В каждый момент исполняется некоторая процедура (блок можно считать частным случаем распроедуривания текста процедуры). В этот момент активная вершина в дереве вычислений соответствует исполняемой процедуре, а вершины на ветви от корня дерева вычислений к активной вершине соответствуют входам в объемлющие процедуры на ветви от корня дерева программы до исполняемой процедуры. При каждой вершине дерева вычислений хранится список пар: идентификатор или индикатор (описание в процедуре, соответствующей данной вершине) и значение, которым он обладает (точнее скрытое от программиста имя значения, так как значение может быть сложным). Этот список пар называется участком данного вызова процедур.

ИВМ имеет счетчик вызовов процедур, значение которого называется текущей областью действия. Перед каждым новым вызовом его значение увеличивается на единицу. Каждая вершина в дереве вычислений помечена целым числом — содержимым счетчика в момент создания этой вершины. Это целое число называется областью действия вершины. Активная вершина имеет наибольший номер среди вершин, имеющих в данный момент в дереве вычислений. В дальнейшем, когда мы будем говорить, что значение имеет некоторую область действия, это будет означать, что данное значение уничтожится вместе с вершиной в дереве вычислений с такой областью действия.

Некоторые действия в программе исполняются последовательно, т. е. первое должно завершиться раньше, чем начнется второе. Некоторая совокупность действий может исполняться совместно, т. е. действия могут исполняться в любом порядке, причем возможно, что некоторые из них исполняются одновременно. (Из-за совместности исполнения предложение $x := 1$; $x + (x := 2)$ может вырабатывать значение 3 или 4, так как операнды в формуле исполняются совместно. Программист должен заботиться, чтобы подобные конфликты не возникали).

Кроме того, некоторая совокупность действий может исполняться параллельно [16].

Дерево вычислений строится ИВМ по дереву программы, начиная с одной стандартной вершины (§ 8).

Теперь определим, какие бывают значения. Действия над значениями стандартных видов весьма просты. Они задаются операциями стандартного вступления (см. § 8). Над числами определены все арифметические операции. Над логическими значениями определен некоторый базис функций алгебры логики. Литеры можно сравнивать (каждой литере соответствует целое — код литеры) и из них можно строить строки. Литеры обычно используются для ввода-вывода информации. Изображение **empty** (вида **void**) возможно написать только в том случае, если над ним делается объединение. Область действия всех

стандартных значений — это корень дерева вычислений, или, как говорят, все они глобальны.

Далее мы будем иногда отождествлять значение с некоторым конструктом, вырабатывающим это значение.

Значение вида `ref m` это: а) либо `nil` — имя с глобальной областью действия, которое ничего не именуется (псевдоимя), б) либо именуется некоторое значение вида `m`. Область действия имени определяет, при выходе из какой процедуры имя и именуемая им память уничтожатся. Область действия имени — это область действия той ячейки (тех ячеек с одинаковой областью действия), на которую (на которые) это имя указывает. Возможно создание имени с текущей областью действия (локальным генератором `loc m` или описанием переменной `m x`;) или с глобальной областью действия (глобальным генератором `heap m` или описанием переменной `heap m x`;) или с областью действия, равной данному имени (последнее возможно при выборке, вырезке или векторизации). По имени можно извлечь именуемое значение (произвести разыменование). Путем присваивания можно сделать имя именующим другое значение с областью действия не младше этого имени, т. е. записать в ячейку (ячейки), на которую указывает имя, новое значение (пример: `real x:=1; to 10 do print(x:=x+1)od; x:=1;`). Два имени `x` и `y` одного и того же вида можно сравнить на равенство (`x:=y` или, эквивалентно, `x is y`), или на неравенство (`x/=y` или, эквивалентно, `x isnt y`). Виды конструктов `x` и `y` уравниваются (см. § 18).

Значение `x` вида `struct(m1a1, ..., mkak)` состоит из k значений видов `m1, ..., mk`, расположенных в указанном порядке. Выборка `ai of x` выбирает (копирует и выдает в качестве результата) i -тое из этих значений, а из имени `xx` вида `ref struct(m1a1, ..., mkak)` выборка `ai of xx` получает имя i -того поля (область действия подымени такая же, как у имени). Область действия структуры — это самая младшая из областей действия ее полей.

Значение `x` вида `[, ...,]m` — это гиперкуб в целочисленном n -мерном пространстве (n — размерность массива), с каждым элементом которого (целочисленной точкой внутри куба) связана некоторая ячейка (или ячейки), содержащая значение вида `m`. Область действия массива — это самая младшая из областей действия его элементов.

Имя фиксированного массива, имеющее вид `ref[, ...,]m`, указывает на некоторый такой куб (точнее на ячейки, которые связаны с его элементами), а имя подвижного массива, имеющее вид `ref flex[, ...,]m`, указывает на все потенциально бесконечное целочисленное пространство, в котором заполнен некоторый куб. Имя фиксированного массива указывает все время на один и тот же куб, а имя подвижного в разные моменты может быть связано с разными кубами. По набору целочисленных

индексов i_1, \dots, i_n вырезка $x[i_1, \dots, i_n]$ извлекает элемент массива x , а из имени массива x такая вырезка получает имя: (с той же областью действия) элемента именуемого массива. По набору индексаторов (одни из которых не индекс) вырезка из массива выделяет подмассив, а по имени массива создает имя (с той же областью действия) подмассива — у этого имени будет свой куб в другом экземпляре целочисленного пространства (может быть другой размерности), но с элементами нового куба будут связаны ячейки, которые связаны с соответствующими элементами старого куба. Индексатор или пуст, или является индексом, или имеет вид $a : b$ at c , где a, b и $at c$ могут отсутствовать независимо друг от друга; a и b вырабатывают (в системе координат массива) нижнюю и верхнюю границы подмассива. В системе координат подмассива эта граничная пара равна $(c, b - a + c)$. Если $at c$ отсутствует, то подразумевается $at 1$, но в случае пустого индексатора сдвига оси координат не происходит.

Выборка a_i of x , где x вида $[\dots] \text{struct}(m_1 a_1, \dots, m_n a_n)$, строит массив вида $[\dots] m_i$. Этот массив состоит из такого же куба как и первоначальный массив, но элементу нового куба соответствует a_i of y , где y — значение, соответствовавшее такому же элементу старого куба. Аналогично из имени xx вида $\text{ref}[\dots] \text{struct}(m_1 a_1, \dots, m_n a_n)$ или вида $\text{ref flex}[\dots] \text{struct}(m_1 a_1, \dots, m_n a_n)$ выборка a_i of xx строит имя (с такой же областью действия как у xx) на массив полей. Вид нового имени — $\text{ref } m_i$. Каждый раз (по усмотрению ИВМ) или строится новое имя, или находится некоторое уже существующее имя массива полей с заданным указателем.

Из имени подвижного при любой выборке или вырезке получается временное имя.

Значение q вида $\text{proc}(m_1, \dots, m_n)$ — это процедура, которая состоит из ссылки на начало некоторого конструкта в программе и номера вершины дерева вычислений. Этот же номер является областью действия процедуры. Процедура может быть вызвана. При вызове процедуры должны быть сообщены k значений a_1, \dots, a_k видом m_1, \dots, m_k . Конструкт p на который указывает процедура, должен начинаться с описания метки (напр., $l : p$) или с плана $(m_1 x_1, \dots, m_k x_k) m : p$ или $m : p$. Перед началом вызова $q(a_1, \dots, a_k)$ (или распроцедурирования q при $k=0$) к вершине дерева вычислений, указанной в процедуре, подвешивается новая вершина, которая снабжается участком: список пар (x_i, a_i) . Список пар пуст, если процедура без параметров. Новая вершина становится активной. Счетчик вызовов увеличивается на единицу и новая вершина помечается новым содержанием счетчика. Начинает исполняться конструкция p . После завершения вызова эта новая вершина уничтожается и ИВМ получает некоторое значение вида m в качестве результата вызова.

Вызов, как и многие другие действия ИВМ, может быть прекращен переходом. Переход осуществляется по метке, которая состоит (как и процедура) из ссылки на начало некоторого конструкта в дереве программы и ссылки на вершину дерева вычислений. При переходе уничтожаются все вершины в дереве вычислений с большей областью действия, чем у вершины из метки. Вершина из метки становится активной и начинает исполняться конструкт, на который стоит ссылка в метке. В частности, если исполняется совместно или параллельно несколько действий, то переход может все их прекратить. Метка не является значением. Поэтому нет имен, именующих метки, и метка не может быть передана как параметр при вызове. Но имеется косвенная возможность сделать такую передачу.

Если переход стоит в позиции, требующей вид, начинающийся на `proc`, то он «запроцедуривается», т. е. вместо осуществления перехода вырабатывается процедура, распроцедуривание которой вызовет этот переход. Пример:

```
([ ] proc void x=(работа, работа, работа, работа, работа,
отдых, развлечение);
int день недели; read (день недели);
x [день недели];
работа : работать exit.
развлечение : отдых : skip).
```

Остальные действия ИВМ будут рассмотрены после описания синтаксиса.

§ 5. СИНТАКСИС ПРОГРАММЫ

Программа состоит из изображений, идентификаторов (слов), индикаторов (выделенных слов), специальных значков и служебных слов (под которые зарезервированы некоторые индикаторы).

Скобки и некоторые служебные слова, выполняющие роль скобок, разбивают программу на дерево блоков. Кроме того каждый блок может быть представлен в виде дерева действий.

Конструкты языка можно разбить на описания и на основы. Синтаксис описаний сравнительно прост.

Примеры описаний:

```
mode compl = struct (real re, im)  $\phi$  описание вида  $\phi$ ;
prio + = 6  $\phi$  описание приоритета  $\phi$ ;
op + = (compl x, y) compl: (re of x + re of y) i (im of x +
im of y)  $\phi$  описание операции  $\phi$ ;
real x, y: = 1  $\phi$  описание переменных  $\phi$ ;
ref int x = z, y = loc int  $\phi$  описание тождеств  $\phi$ ;
proc p = (compl x, y) compl: (re of x + re of y) i
(im of x + im of y)  $\phi$  описание процедуры  $\phi$ ;
l:  $\phi$  описание метки  $\phi$ .
```

Между символами ϕ и ϕ заключены комментарии, которые не учитываются ИВМ, а пишутся для удобства изучения программы человеком.

Синтаксически основы представляют линейную запись деревьев. При исполнении основы сначала исполняются вычисления, соответствующие наследникам корня дерева (в любом порядке, если не говорится обратное), а затем над полученными результатами осуществляется действие, указанное в корне дерева.

Для того чтобы по линейной записи программы построение деревьев было однозначным, основы разбиты на четыре класса (каждый включает в себя предыдущий): первичные, вторичные, третичные и класс всех основ, а важный класс основ — формулы (они относятся к третичным) разбит на группы по приоритетам.

Утверждение об однозначности построения дерева по линейной записи программы с использованием классов и приоритетов не является доказанным фактом (современная техника теории грамматик не позволяет доказывать такие утверждения), но оно заслуживает доверия в качестве эмпирического факта.

Программа на АЛГОЛе 68 строится с помощью синтаксических правил, приведенных в таблице 3, с учетом следующих соотношений между видами конструкторов (позиции конструкторов указаны в табл. 3):

1) Пропуск, переход и псевдоимя допускают любой вид, называемый объемлющим конструктором.

2) В ядре описатель задает вид закрытого предложения.

3) В вызове параметры приводимы к видам, задаваемым процедурой (всегда одного и того же вида), вырабатываемой первичным.

4) Операнды в формуле приводимы к видам, указанным в описании операции.

5) Первичное вырезки (вторичное выборки) приводится к виду, из которого можно делать вырезку (выборку).

6) Операнды отношения одноименности вырабатывают имена одного и того же вида. Один из операндов стоит в позиции $k1$, чтобы не допустить лишнее разыменование.

7) В присваивании левая часть приводится к имени некоторого вида, а правая часть приводится к этому виду. Результат присваивания (до приведений) — имя, выработанное левой частью.

8) Текст процедуры вырабатывает процедуру вида, определяемого планом процедуры.

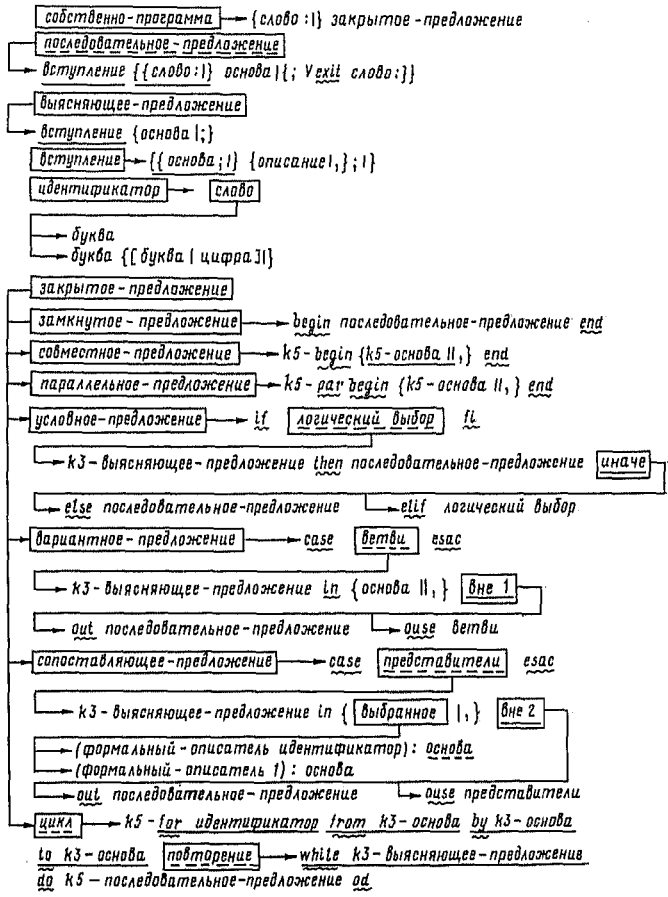
9) Описание переменной $\text{in } x := a$ эквивалентно описанию тождества $\text{ref } \text{in } x = \text{loc } \text{in } := a$.

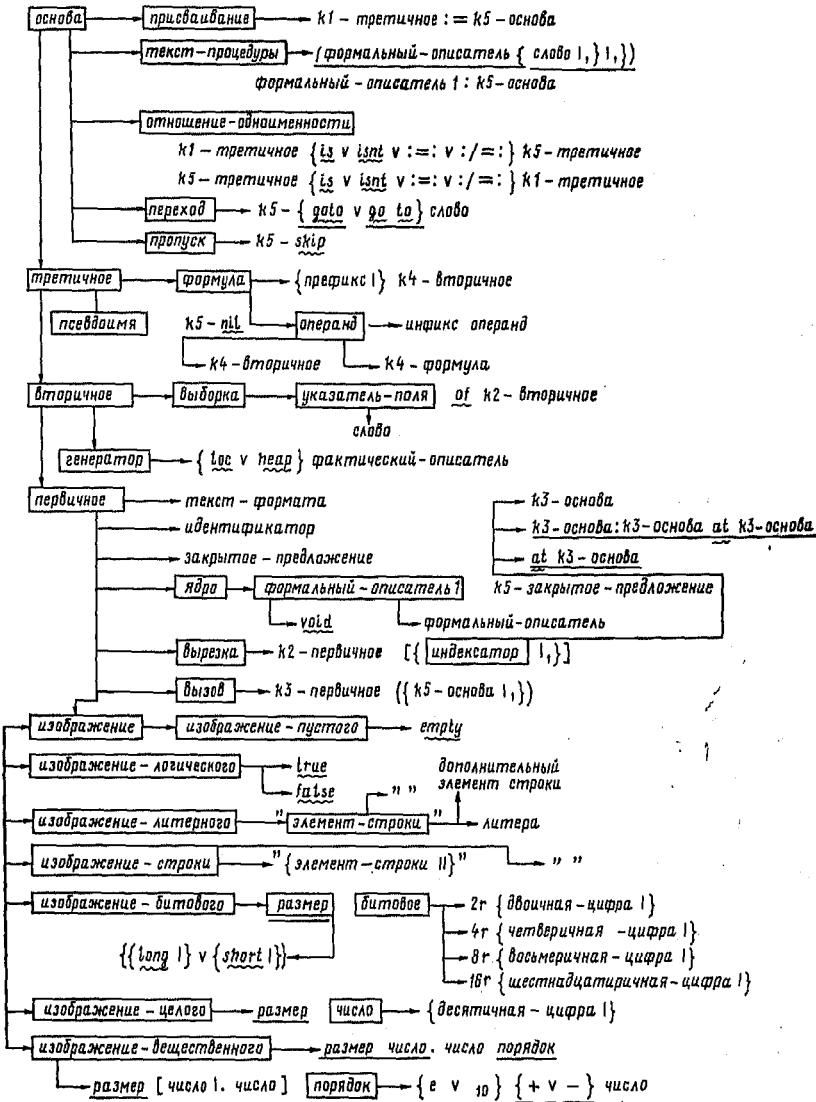
10) Если закрытое предложение имеет несколько окончаний, то виды этих окончаний уравниваются (как например, в ветвях выбирающих предложений или в последовательном предложении

Синтаксическая схема

Обозначения

- A** → **B**
- S** обозначает, что A должно быть заменено на B или C
- {A V B}** обозначает A или B
- {A || B}** обозначает одно из: ABA, ABAVA, ABAVAVA, ...
- {A | B}** обозначает A или {A || B}
- [A] B** обозначает A, B или AB
- k3** - сильная позиция (*strong*)
- k4** - крепкая позиция (*firm*)
- k3** - раскрытая позиция (*weak*)
- k2** - слабая позиция (*weak*)
- k1** - мягкая позиция (*soft*)
- k5** - обозначает, что стоящая за ним конструкция может появляться только в сильной позиции
- abc** обозначает, что abc может быть опущено
- abc** обозначает, что abc образует новый блок
- abc** обозначает выделенное слово abc





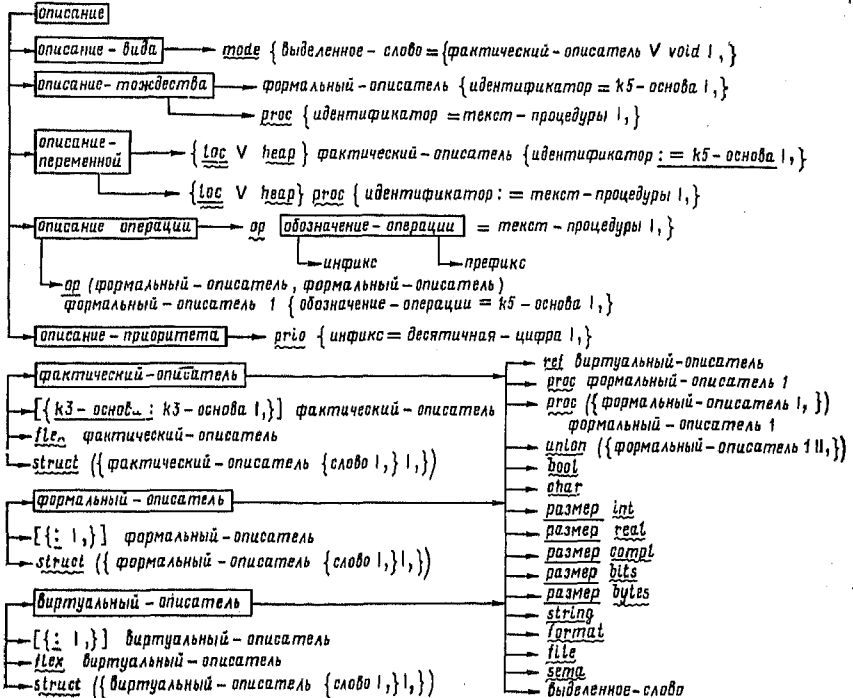
Знаки операций

Знак 1 — один из V, A, &, ≠, ≤, ≥, ÷, %, □, Γ, ⊥, ∇, ~, †, ‡, +, - .
 Знак 2 — один из <, >, =, /, *, x .

инфикс → {знак 1 ∨ знак 2} **знак 2** { := ∨ = : }
 → выделенное слово

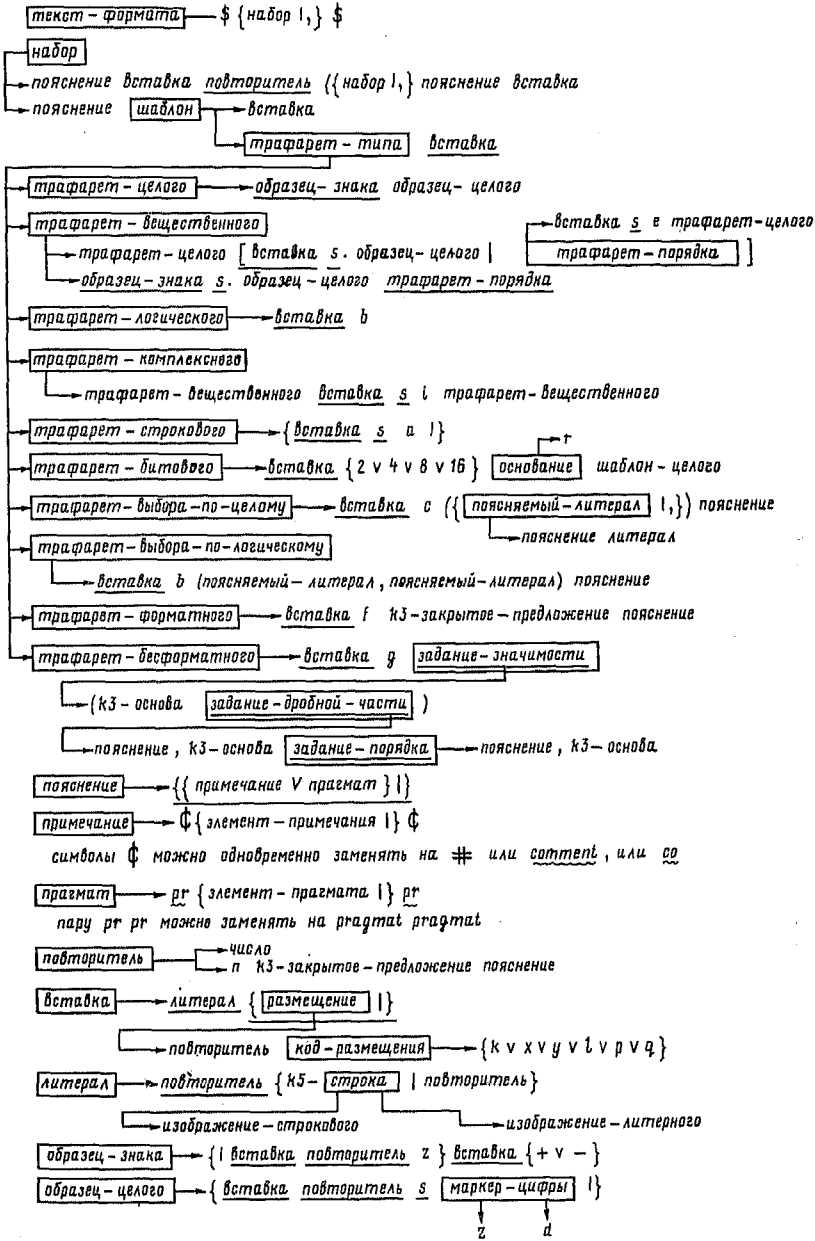
префикс → знак 1 **знак 2** { := ∨ = : }
 → выделенное слово

Знаки V, A, &, ≠, ≤, ≥, ÷, %, □, Γ, ⊥, ∇, ~, †, ‡, x не предусмотрены стандартом ЗНК. Вместо ≠, ≤, ≥, x, † можно использовать /, <, >, =, *, ** соответственно, а остальные заменяются выделенными словами.



Примечание

- после *flex* может стоять только описатель массива ;
- в описателях и вырезках квадратных скобки можно заменять на круглые ;
- в закрытых - предложениях *begin, end* можно заменять на (,) ;
- в условных предложениях *if, then, else, elif, fi* можно заменять на (, |, |, | :) ;
- в варианном и сопоставляющем - предложениях *case, in, out, case, case* можно заменять на (, |, |, | :) ;
- пояснения (кроме текстов - формата) могут находиться всюду, кроме как внутри изображений и слов.



закрытого предложения, когда уравниваются виды основ, стоящих перед *exit*, и последняя основа). При уравнивании один из видов (главный, см. § 18) приводится в позиции, навязываемой объемлющим конструктором, а остальные приводятся в позиции *k5* к тому же виду.

11) В совместном предложении, вырабатывающем одномерный массив, виды основ должны сильно приводиться к виду элемента этого массива; в вырабатывающем многомерный массив — значения вида массив на единицу меньшей размерности (вид элемента массива сохраняется); в вырабатывающем структуру — значения видов полей структуры.

Замечание. Синтаксис АЛГОЛа 68 таков, что два идентификатора не могут стоять рядом, поэтому внутри идентификатора возможны пробелы и возможны переносы идентификатора на новую строчку. Таким образом допустимы идентификаторы: новая страница или *bits width*. Но два индикатора могут стоять рядом (например, в случае применения двух унарных операций подряд: *abs sign x*). Поэтому в индикаторе не может быть пробелов и переносов.

§ 6. СТРУКТУРА ПРЕДЛОЖЕНИЙ

Из основ и описаний строятся последовательные предложения и выясняющие предложения. Из них с помощью ограничителей строятся закрытые предложения, которые в свою очередь являются основами. Собственно программа представляет собой закрытое предложение, которому могут предшествовать метки.

Последовательное предложение имеет вид:

<вступление> <последовательность кортежей>.

Вступление состоит из разделенных точками с запятой совместных описаний и основ. Каждое совместное описание состоит из списка разделенных запятыми описаний. Основы во вступлении стоят в позиции *k5* и вырабатывают *void*. Кортеж может начинаться с описания меток. На практике, по-видимому, не будет употребляться более одной метки подряд. Далее кортеж строится из последовательности основ, разделенных точками с запятыми. Элементы последовательности кортежей разделены одним из двух символов: *exit* (выход) или точка с запятой. Все основы в последовательности кортежей, за которыми следует точка с запятой, стоят в позиции *k5* и вырабатывают *void*. Результат последовательного предложения вырабатывает одна из основ, за которой следует *exit*, или последняя основа. Виды этих основ уравниваются.

Перед началом исполнения последовательного предложения к активной вершине дерева вычислений подвешивается новая вершина, которая становится активной. В ее участок заносятся все идентификаторы и индикаторы из вступления предложения,

причем индикатору приоритета соответствует приоритет, индикатору вида — ссылка на его фактический описатель и на активную вершину, идентификатору метки — соответствующая метка (все они в дальнейшем не изменяются). Остальные индикаторы и идентификаторы не доступны вплоть до исполнения соответствующего описания.

Затем последовательно исполняются описания и первый из кортежей. Переход может прекратить исполнение описания или начатого кортежа (и всех начатых в нем действий) и инициировать исполнение другого кортежа. Область действия результата (после приведения) должна быть старше области действия вершины, созданной перед началом исполнения последовательного предложения.

Частным случаем последовательного предложения является выясняющее предложение, в котором не могут входить описания меток.

Теперь можно объяснить, как исполняются закрытые предложения. Они все не приводимы, и все они, кроме цикла, совместного и параллельного предложения, могут стоять в любой позиции.

1) Исполнение замкнутого предложения состоит из исполнения входящего в него последовательного предложения. Его назначение — сделать последовательное предложение основой, а также управлять порядком действий в формулах.

2) Условное предложение исполняется как в АЛГОЛе 60. Если после **else** стоит только одно условное предложение, то **else if** можно заменить на **elif**, а последний символ **fi** ликвидировать. Виды ветвей **then** и **else** (или **elif**) уравниваются.

3) В вариантном предложении выясняющее предложение вырабатывает целое, по которому производится выбор основы. Основы нумеруются слева направо начиная с единицы. Если целое не попадает в интервал $[1 : \langle \text{число основ} \rangle]$, то исполняется часть **out**. Если после **out** стоит только одно вариантное предложение, то **out case** можно заменить на **ouse**, а последний символ **esac** ликвидировать. Виды основ после **in** и предложения после **out (ouse)** уравниваются.

4) В сопоставляющем предложении выясняющее предложение вырабатывает значение некоторого объединенного вида **m**. В списке, следующем после **in**, перед каждой основой стоит спецификация, описатель которой допустим видом **m** (т. е. он эквивалентен представителю или объединению некоторых представителей из **m**). Если вид значения **a**, вырабатываемого выясняющим предложением, допустим описателем **a** из некоторой спецификации, то исполняется эта спецификация, а затем стоящая за ней основа. В противном случае исполняется часть **out**. Если в спецификацию входит идентификатор **x**, то к активному участку добавляется пара (x, a) , где **x** имеет вид **a**; эта пара уничтожается после исполнения основы. В противном случае

никаких действий при выполнении спецификации не предпринимается. Если после **out** стоит одно только сопоставляющее предложение, то **out case** можно заменить на **ouse**, а последний символ **esac** ликвидировать. Виды основ после **in** и предложения после **out (ouse)** уравниваются.

Каждая основа в программе вырабатывает значение определенного (зависящего от этой основы, позиции и объемлющего конструкта) вида. Возможно, это значение одного из видов представителей объединения (если основа вырабатывает значение объединенного вида). Такое значение можно только записывать, считывать и использовать как результат в выясняющем предложении сопоставляющего предложения. При записи в память оно записывается по имени объединенного вида. При передаче в качестве фактического параметра вызова процедуры, соответствующий формальный параметр текста процедуры должен быть объединенного вида. Таким образом значение как бы законсервировано. Никаких действий над ним делать нельзя (кроме дообъединения), и только если значение является результатом выясняющего предложения в сопоставляющем предложении, оно расконсервируется.

Одной из ошибок при работе с объединенными видами будет попытка использовать недопустимо «похожие» представители, такие как **int** и **proc int** (см. §18).

Три предыдущих предложения называются выбирающими. Вершины дерева вычислений, созданные при исполнении любого выбирающего предложения, после получения результата уничтожаются. Область действия результата выбирающего предложения должна быть старше уничтожаемых после его исполнения вершин. (Заметим, что из-за неаккуратности в определении области действия в АЛГОЛе 68, программа **if ref real x; true then real y; x := y fi** ошибочна).

5) Совместное предложение вырабатывает либо значение вида **void**, либо структуру, поля которой вырабатываются основами в порядке их следования, либо одномерный массив, элементы которого вырабатываются основами в порядке их следования, либо многомерный массив, тогда это массив x такой, что результат вырезки $x[i, \dots]$ равен значению, вырабатываемому i -той основой. Во всех случаях основы исполняются совместно в позиции $k5$.

Отметим, что совместное предложение не может состоять из одной основы. Действительно, допустив совместные предложения с одной основой, в программе

```
(mode a = struct (ref a a);
```

```
  a нобуо, йонеда;
```

```
  a of йонеда : = nil;
```

```
  нобуо : = (йонеда);
```

```
  a of нобуо : = : йонеда)
```

результат отношения одноименности зависит от того, как трак-

товать конструктор (Йонедэ): то ли как замкнутое предложение, основа которого разыменовывается и результатом оказывается структура *s*, единственным полем которой является *nil*, то ли как совместное предложение с одной основой, вырабатывающее структуру, единственное поле которой именует *s*. Имя, именуемое что-либо, не может быть *nil*. Таким образом в первом случае отношение одноименности вырабатывает *false*, а во втором случае *true*.

Нобуо Йонедэ — логик, заметивший, что в совместном предложении не может стоять одна основа.

6) Параллельное предложение содержит список основ (процессов), исполнение которых происходит параллельно и управляется целочисленными семафорами. Семафор это значение вида *sema* (стандартный вид), которое задает количество процессов, стоящих в очереди. Управляются семафоры операциями *up* и *down* из стандартного вступления (§ 8) согласно [16].

7) Цикл синтаксически выглядит так: **for** счетчик **from** начало **by** шаг **to** конец **while** условие **do** тело **od**. Каждая из частей — **for** счетчик, **from** начало, **by** шаг, **to** конец, **while** условие — может отсутствовать независимо от других. Тогда подразумевается **for** <идентификатор, не используемый в программе>, **from** 1, **by** 1, **to** ∞ и **while true** соответственно. Перед началом цикла пара (счетчик, начало), где «счетчик» вида *int*, помещается в участок активной вершины, а по окончании цикла уничтожается. В цикле, если шаг ≥ 0 и счетчик \leq конец или шаг ≤ 0 и счетчик \geq конец, то исполнение продолжается. Исполняется выясняющее предложение. Если вырабатывается *true*, то исполнение продолжается. Исполняется предложение после **do**. К значению «счетчика» прибавляется «шаг». Вершины дерева вычислений, созданные выясняющим предложением, стоящим после **while**, и последовательным, стоящим после **do**, устраняются. Цикл повторяется.

В выясняющем и последовательном предложениях цикла может использоваться идентификатор, стоящий после **for**. Но он не может изменяться программистом, так как его вид *int*, а не *ref int*.

§ 7. ПЕРЕРЫВАНИЯ

В некоторых случаях исполнение программы согласно RR является неопределенным. Тогда в реализации может быть предусмотрено некоторое действие, например прерывание.

Неопределенными являются следующие ситуации:

1) Если область действия результата старше области действия в момент начала исполнения:

- последовательного предложения;
- выбирающего предложения;

- формулы;
 - вызова;
 - распроедуривания.
- 2) Если при присваивании область действия источника стар-
ше области действия получателя.
- 3) Если псевдоимя встречается:
- в левой части присваивания;
 - во вторичном выборки;
 - в первичном вырезки;
 - при разыменовании.
- 4) Если в совместном предложении, вырабатывающем мно-
гомерный массив, результаты составляющих основ имеют раз-
ные граничные пары.
- 5) Если при присваивании фиксированному имени источник
и получатель имеют разные граничные пары.
- 6) Если в вырезке индексы или индексомеры выходят за гра-
ницы массива.
- 7) Если идентификатор или индикатор к моменту его исполь-
зования не занесен в соответствующий участок.
- 8) Если неопределен результат стандартной операции (на-
пример, возникло деление на ноль).
- Возможны также неопределенные ситуации в обмене
(см. § 9).

Часть 2

ОКРУЖЕНИЕ ПРОГРАММЫ

§ 8. СТАНДАРТНОЕ ВСТУПЛЕНИЕ

Синтаксис АЛГОЛа 68 не определяет ни одного вида, ни одной процедуры и операции, а только лишь задает механизм их построения из видов, процедур и операций окружения. Пакет программ на АЛГОЛе 68 исполняется в следующем окружении:

```
( <стандартное вступление>;
  <библиотечное вступление>;
  <системное вступление>;
  par begin <системная задача 1>, ..., <системная задача n>
    (<собственное вступление>; (<метки> <программа 1>);
    <собственное заключение>),
    .
    .
    .
    (<собственное вступление>; (<метки> <программа m>);
    <собственное заключение>) end)
```

Таким образом перед исполнением пакета исполняются стандартное, библиотечное и системное вступление, а перед исполнением отдельной программы исполняется собственное вступ-

ление этой программы. Завершается исполнение каждой отдельной программы исполнением собственного заключения.

Стандартное вступление содержит описание наиболее употребительных видов, операций и процедур. Конечно, обозначения стандартного вступления могут быть переопределены в программе.

С помощью библиотечного вступления можно расширить стандартное вступление дополнительными примитивами. Системное вступление служит для определения взаимодействия с операционной системой. Параллельно с программами пакета исполняются управляющие задачи операционной системы, которые взаимодействуют с помощью семафоров, а также иницируются в случае прерываний или завершения программ пакета.

Таблица 4

Унарные операции

(в фигурных скобках даны представления в русском варианте АЛГОЛа 68)

Обозначение	Вид операнда <i>a</i>	Вид результата	Пояснения
not {не}	bool bits	bool bits	отрицание каждого элемента <i>a</i>
+, —	L int L real L compl	L int L real L compl	
bin {бин}	L int	L bits	элементы двоичного представления <i>a</i>
conj {сопрж}	L compl	L compl	сопряженное для <i>a</i>
arg {арг}	L compl	L real	аргумент <i>a</i>
abs {абс}	L int L real L compl	L int L real L real	модуль <i>a</i>
odd {нчт}	bool	int	if <i>a</i> then 1 else 0 fi
sign {знак}	char L bits	int L int	внутренний код литеры обратная для bin
round {окр}	L int	bool	true, если <i>a</i> нечетно
entier {целч}	L int	int	{ +1 для <i>a</i> > 0, 0 для <i>a</i> = 0,
repr {пред}	L real	int	-1 для <i>a</i> < 0
leng {удл}	L real	L int	ближайшее целое к <i>a</i>
shorten {укр}	L real	L int	целая часть <i>a</i>
	L int	char	литера с кодом <i>a</i>
	m	long m	m = L int, L real, L bits,
	long m	m	L compl, L bytes (изменение разрядности <i>a</i>)
level {ур}	int	sema	установка семафора
up {вверх}	sema	int	счетчик семафора
down {вниз}	sema	void	поднять семафор
re {вч}	sema	void	опустить семафор
im {мч}	L compl	L real	вещественная часть <i>a</i>
	L compl	L real	мнимая часть <i>a</i>

Описания стандартного вступления включают: стандартные виды **void**, **bool**, **int**, **real**, **char**, **compl**, **bits**, **bytes** и **string**, а также потенциально бесконечное число модификаций длин видов **int**, **real**, **compl**, **bits** и **bytes**; стандартные обозначения операций и функций; запросы к обстановке и описания обмена.

Все стандартные операции определены над стандартными видами значений. Исключение составляют бинарные операции взятия нижних и верхних границ массивов по n -й размерности n **lwb** x и n **upb** x соответственно, где n вида **int**, а x любого вида, специфицированного как массив или как объединение массивов, а также унарные операции **lwb** $x=1$ **lwb** x и **upb** $x=1$ **upb** x .

В идентификации операций участвуют виды операндов, поэтому одно обозначение часто используется для многих операций. В таблицах 4-6 приведены все операции стандартного вступления с указанием видов значений операндов и результата.

Однотипные операции над значениями различных модификаций длин имеют одинаковые обозначения. Значок L в таблицах 4 и 6 обозначает последовательность любого числа символов **long** или **short**, но одинаковую в пределах одной операции. В идентификаторах стандартных функций и запросов к обстановке L обозначает последовательность любого числа приставок **long** или **short**. Аналогично в русском варианте **Д** (**Д**) обозначает последовательность символов **дли** или **кор** (**дли** или **кор**).

Таблица 5

Стандартные приоритеты
(в фигурных скобках даны представления в русском варианте: АЛГОЛа 68)

Приоритет	Обозначения
9	+ *, i {им}
8	** , up {вверх}, down {вниз}, shl {лев}, shr {прав}, lwb {нигр}, upb {верг}
7	*, /, %, %*, mod {мод}, over {цед}, elem {элемент}
6	-, +
5	<, <=, >, >=, lt {мш}, le {нб}, ge {им}, gt {бш}
4	=, /=, eq {пр}, ne {нр}
3	and {и}
2	or {или}
1	-: =, +: =, *: =, /: =, %: =, %*: =, + =:, minusab {минпр}, plusab {плюспр}, timesab {умпр}, divab {делпр}, modab {модпр}, plusto {прип}, overab {цедпр}

Стандартные математические функции введены описанием процедур вида **proc** (**L real**) **L real** для идентификаторов **L sqrt**, **L exp**, **L ln**, **L cos**, **L arccos**, **L sin**, **L arcsin**, **L tan**, **L arctan** и **L next random**.

Бинарные операции
(в скобках приведены обозначения, обладающие такой же операцией)

Обозначение	Виды операндов		Вид результа- та	Пояснения
	<i>a</i>	<i>b</i>		
i (+ *) ** (up)	L int, L real L int	L int, L real int	L compl L int	$a+ib$ a^b
shl (up) shr (down) % (over)	L real L compl L bits L bits L int	int int int int L int	L real L compl L bits L bits L int	сдвиг (shl влево, shr вправо) целочисленное де- ление
mod (% *)	L int	L int	L int	остаток по моду- лю <i>b</i>
/	L int	L int	L int	арифметические операции
+ , - , * + , - , * , /	L int L real L real L int	L int L real L int L real	L int L real L real L real	
*	L int, L real L compl L compl int	L compl L int, L real L compl	L compl L compl L compl	<i>a</i> раз повторить <i>b</i> <i>b</i> раз повторить <i>a</i> <i>a</i> -тый бит <i>b</i>
elem	string, char int int	string, char int L bits L bytes	string string bool char	<i>a</i> -тая литера <i>b</i> конкатенация строк
+	string, char	string, char	string	конкатенация строк
< (lt), > (gt), <= (le), >= (ge)	L int, L real	L int, L real	bool	сравнение чисел
< (lt), > (gt), <= (le), >= (ge), <= (le), >= (ge), <= (le), >= (ge), = (eg), /= (ne)	char string string char L bytes L bits L bits bool	char string string string L bytes L bits L bits bool	bool bool bool bool bool bool bool bool bool	сравнение кодов литер лексикогра- фически
and	L int, L real L bytes, L compl, L bits char string bool L bits	L int, L real L bytes, L compl, L bits char string bool L bits	bool bool bool bool bool bool bool bool	$a \wedge b$ поэлементно
or	L bits	L bits	L bits	$a \vee b$ поэлементно
-: = (minusab), +: = (p usab), *: = (time sab)	ref L int ref L real ref L compl	L int L real, L int L real,	ref L int ref L real ref L compl	$a: = a - b$ $a: = a + b$ $a: = a * b$

Обозначение	Виды операндов		Вид результата	Пояснения
	а	в		
%:= (divab)	ref L int	L compl, L int	ref L int	$a := a \% b$
%*:= (modab)	ref L int	L int	ref L int	$a := a \text{ mod } b$
/:= (divab)	ref L real	L real, L int	ref L real	$a := a / b$
	ref L compl	L compl, L real, L int	ref L compl	
+= (plusab)	ref string	string, char	ref string	$b := a + b$
+ -= (plusto)	string, char	ref string	ref string	$b := a + b$
*:= (timesab)	ref string	int	ref string	$a := a * b$

{в русском варианте Д корень, Д эксп, Д логрф, Д кос, Д арк-кос, Д син, Д арксин, Д танг, Д арктанг и Д след ПСЧ}. Аналогично введены константа $L \pi$ {Д пи в русском варианте}, а также константы запросов к обстановке: *int lengths, int shorths, L max int, real lengths, real shorths, L max real, L small real, bits lengths, bits shorths, L bits width, bytes lengths, bytes shorths, L bytes width, max abs char, null character, flip, flop, errorchar, blank*

{соответственно в русском варианте: число длин цел, число кор цел, Д макс цел, число длин вещ, число кор вещ, Д макс вещ, Д точность вещ, число длин бит, число кор бит, Д размер бит, число длин слог, число кор слог, Д размер слог, макс лит, заполнитель, да, нет, литера ошибки, пробел} для отражения особенностей реализации в части числа значащих приставок **long** и **short**, разрядности представлений и внутренних представлений некоторых важных литер. К запросам к обстановке также относятся операции **abs** и **repr**, отражающие связь литер с их целочисленными представлениями.

Для упаковки логических массивов и строк предусмотрены процедуры $L \text{ bits pack}$ {Д бит пак} с планом ($[] \text{ bool } a$) $L \text{ bits}$ и $L \text{ bytes pack}$ {Д слог пак} с планом ($\text{string } a$) $L \text{ bytes}$ соответственно.

Для выработки псевдослучайных чисел в собственном вступлении программы предусматривается переменная

$L \text{ int } L \text{ last random} := \text{round} (L \text{ max int} / L 2)$

и процедура

$\text{proc } L \text{ random} = L \text{ real} : L \text{ next random} (L \text{ last random}).$

{В русском варианте Д пред псч и Д псч соответственно}

§ 9. ОБМЕН

Предполагается, что ИВМ кроме ячеек памяти имеет некоторое количество книг. Доступ к книгам сложнее, чем к памяти,

но объем информации, которую можно хранить в книгах, существенно больше, чем память.

Книга содержит: ссылку на находящийся на каком-либо внешнем носителе текст книги, идентификационную строку, размер текста и информацию о возможности внесения в книгу новой информации. Последнее введено с той целью, чтобы разрешить в каждый момент писать в книгу только одному пользователю.

Информация в тексте содержится всегда в литерной форме. Форма представления значений в тексте зависит от способа обмена этих значений. Существует три способа обмена: бесформатный, форматный и двоичный. При бесформатном и форматном обмене информация в тексте доступна для человеческого восприятия, а при двоичном обмене способ представления информации скрыт. Текст книги содержит некоторое переменное число страниц, каждая страница — переменное число строчек, каждая строчка — переменное число литер. Позиция внутри текста определяется тройкой (p, l, c) , где p — номер страницы, l — номер строчки и c — номер литеры.

Каждая группа однотипных внешних устройств моделируется каналом. Канал характеризуется свойствами, такими как возможность ввода/вывода, возможность двоичного обмена, возможность возврата на начало книги, возможность установки на любую позицию внутри книги, возможность изменять размер книги, возможность переобозначить книгу, способ кодировки литер, пропускная способность канала и максимальный размер книги. Все эти свойства, кроме последних двух, зависят от книги, обмениваемой через данный канал (например, ввод через канал разрешен, но запрещено чтение из некоторой книги, обмениваемой через данный канал).

В реализации каждому каналу ставится в соответствии значение вида `channel` {канал}. Число каналов зависит от реализации, однако язык требует, чтобы в каждой реализации было по крайней мере три канала. Это допускающий ввод `stand in channel` {станд канал ввода}, допускающий вывод `stand out channel` {станд канал вывода} и допускающий ввод, вывод и двоичный обмен `stand back channel` {стенд канал обмена}.

Связь между книгой и программой осуществляется через файл. Файл — это значение вида `file` {файл}, которое содержит: ссылки на книгу и текст этой книги; информацию о канале, через который производится обмен; кодировку информации в книге (пользователь может вызвать процедуру `make conv` задать кодировку, отличную от кодировки, принятой для канала); состояние файла (т. е. в данный момент книга читается или пишется); текущая позиция книги; в случае форматного обмена, информацию о формате; стоп строку, любая литера которой служит ограничителем ввода при чтении значений вида `string`

(стоп строка задается вызовом *make term*); процедуры обработки некоторых событий в процессе обмена.

В процессе обмена предусматриваются реакции на некоторые ситуации. К таким ситуациям относятся: выход, при вводе, за конец информации в книге (выход за логический конец); выход, при выводе, за пределы книги (выход за физический конец); выход за конец строчки; выход за конец страницы; конец формата (в случае форматного обмена); ошибки значения, если, при вводе, строка не может быть преобразована в значение требуемого вида, или, при форматном выводе, значение не может быть выведено под контролем шаблона; ошибки литеры, если литера не может быть перекодирована или при вводе поступает неожиданная литера. При открытии файла задаются стандартные реакции обработки событий. В дальнейшем они могут быть изменены посредством процедур реакции (см. таблицу 8).

Перед началом обмена книга должна быть открыта, после окончания — закрыта (см. табл. 8). Процедура *open* осуществляет открытие книги, а процедура *close* закрывает книгу. В отличие от *close* процедура *lock* делает книгу недоступной для открытия (например, выдается указание оператору снять магнитную ленту), но книга может быть открыта после некоторого системного действия. Процедура *scratch* уничтожает книгу.

Новая пустая книга заданного размера и с заданной идентификацией может быть создана и открыта вызовом процедуры *establish*. Процедура *create* создает книгу с максимальным допустимым размером для заданного канала и с пустой идентификацией.

Язык позволяет работать с некоторой областью памяти как с книгой. Процедура *associate* связывает с заданным файлом заданный трехмерный массив литер.

В собственном вступлении каждой программы через стандартные каналы *stand in channel*, *stand out channel* и *stand back channel* на файлах *stand in*, *stand out* и *stand back* открыто по книге. В программе обмен с этими книгами производится без предварительного открытия. В собственном заключении эти книги закрываются.

Обмен с книгой производится посредством процедур обмена. Этим процедурам кроме имени файла, на котором данная книга открыта, как параметр, подается список данных обмена. При выводе в списке могут находиться значения, описатели видов которых после раскрытия не содержат символов **void**, **ref**, **proc**, **flex** и **union**. При вводе в списке могут находиться значения, виды которых после слабого приведения являются либо именами выводимых видов, либо **ref string**. В процессе обмена значения из списка данных обмениваются по порядку.

Массивы обмениваются поэлементно, в лексикографическом

порядке возрастания набора индексов. Поля структур обмениваются в порядке следования их в описателе.

При бесформатном обмене значения, в зависимости от вида, представляются некоторым стандартным образом в виде строк литер. При выводе представляем целых и вещественных значений можно управлять, написав в списке данных вызов одной из процедур перевода (см. табл. 8). Первый параметр этих процедур — переводимое число. Параметр *width* задает разрядность десятичного представления. Если *width* > 0, то всегда выдается знак; если *width* < 0, то выдается только знак минус, а плюс заменяется пробелом; если *width* = 0 (кроме *float*, где всегда *width* ≠ 0), то выдается строка наименьшей длины, в которую может быть переведено значение согласно другим параметрам. Параметр *after* задает разрядность дробной части. Параметр *exp* задает разрядность порядка и интерпретируется как *width* в *whole*.

Размещением значений при бесформатном обмене можно управлять, поместив в список данных процедуры вида *proc (ref file) void*, как правило это будут процедуры размещения (см. таблицу 8), выполняющие переход на новую строчку и т. п.

При форматном обмене управление размещением информации в книге осуществляется форматами. Формат — это значение вида *format*. Он может помещаться в списке данных для процедур форматного обмена среди обмениваемых значений. Формат, встреченный в процессе форматного обмена, становится текущим форматом файла.

Форматы задаются текстами формата (синтаксис см. в табл. 3). Основной единицей текста формата является шаблон. В процессе форматного обмена обмениваемые значения простых видов сопоставляются с шаблонами, выбираемыми по порядку из формата. Шаблон определяет способ перевода значения в литерную форму или из литерной формы. Шаблон управляет размещением значений в книгах, позволяет производить произвольные текстовые вставки. Например, в результате вывода под контролем шаблона *"x=" + d" "3d.3d" "ez+d* значение 376.9246 будет напечатано с новой строчки:

x = +3 769.246 e -1

Шаблоны будем называть по типу содержащихся в них трафаретов.

Тексты из шаблона при выводе выдаются, а при вводе запрашиваются столько раз, чему равно значение повторителя (отсутствие повторителя всюду обозначает повторитель = 1).

Размещение из шаблона управляет расположением информации в книге (*R* обозначает повторитель):

Rx вызывает *R* раз *space*;
Ry вызывает *R* раз *back space*;
Rp вызывает *R* раз *new page*;
Rl вызывает *R* раз *new line*;

Rk передвигает позицию внутри текущей строки на литеру с номером *R*;

Rq эквивалентно *R " . "*.

Маркеры из шаблонов целого, вещественного, логического, комплексного, битового и строкового определяют редактирующие действия (см. табл. 7). Если перед маркером стоит символ *s*, то: при вводе запрашиваемые литеры не считаются из книги, а ввод продолжается как будто они были считаны, при выводе соответствующие литеры не записываются в книгу.

Таблица 7

Маркер	Действие при вводе	Действие при выводе
<i>Rd</i>	Запрашивает <i>R</i> цифр.	Выдает <i>R</i> цифр.
+	Если знак не был введен, то запрашивает «+» или «-».	Если знак не был выдан то выдает «+» или «-».
-	Если знак не был введен, то запрашивает «-» или «+».	Если знак не выдан, то выдает «-» или «+».
<i>Rz</i>	Пусть имеется <i>n</i> начальных пробелов а) в образце знака «+» («-»): Если знак не был введен, то пропускается не больше $\min(n, R)$ первых пробелов, затем запрашивается «+» или «-» («-» или «+»), затем запрашивается $R=n$ цифр, иначе действие как у маркера « <i>d</i> », б) в образце целого: $\min(R, n)$ первых пробелов заменяются нулями, затем запрашивается $R-n$ цифр.	Пусть имеется <i>n</i> начальных нулей а) в образце знака «+» («-»): Если знак не был выдан, то первые $\min(R, n)$ нулей заменяются пробелами, затем выдается знак и $R-n$ цифр, иначе действие как у маркера « <i>d</i> ». б) в образце целого: $\min(R, n)$ первых пробелов заменяются нулями, затем выдается $R-n$ цифр.
.	Запрашивает десятичную точку.	Выдает десятичную точку.
<i>e</i>	Запрашивает « <i>e</i> » (затем порядок).	Выдает « <i>e</i> » (затем порядок).
<i>i</i>	Запрашивает « <i>i</i> ».	Выдает « <i>i</i> ».
<i>2r</i>	Показывает, что вводимое битовое представлено по основанию, стоящему перед <i>r</i> .	Показывает, что выводимое битовое следует представить по основанию, стоящему перед <i>r</i> .
<i>4r</i>		
<i>8r</i>		
<i>16r</i>		
<i>Ra</i>	Запрашивает <i>R</i> литер.	Выдает <i>R</i> литер.
<i>b</i>	Запрашивает литеру <i>flip</i> или <i>flop</i> .	Выдает литеру <i>flip</i> или <i>flop</i> .

Шаблоны выбора позволяют кодировать целые или логические значения текстами (например: *b* («истина», «ложь»)). Шаблоны бесформатного позволяют в процессе бесформатного обмена обменивать отдельные значения процедурами бесформатного обмена. Назовем задания значимости, дробной части и порядка из шаблона бесформатного спецификациями. Если спецификации отсутствуют, то значение обменивается процедурой *put* или *get*. Если присутствуют одна (две, три) спецификации, то значение при выводе преобразуется в строку процедурой *whole (fixed, float)* со спецификациями, взятыми в качестве параметров, и полученная строка выдается процедурой *put*; при

Процедуры обмена

13*	№	Идентификатор процедуры	План	Комментарий
	1.	<i>estab possible</i> {можно завести}	(channel <i>chan</i>) bool	Запросы к обстановке для канала выдает true, если на канале <i>chan</i> можно завести еще один файл
	2.	<i>stand conv</i> {станд код}	(channel <i>chan</i>) proc (ref file) conv	
	3.	<i>get possible</i> {возм ввод}	(ref file <i>f</i>) bool	Запросы к обстановке для файла выдает true, если файл <i>f</i> может использоваться для ввода
	4.	<i>put possible</i> {возм вывод}	(ref file <i>f</i>) bool	
	5.	<i>bin possible</i> {возм двоичный}	(ref file <i>f</i>) bool	выдает true, если файл <i>f</i> может использоваться для вывода
	6.	<i>compressible</i> {сжимаем}	(ref file <i>f</i>) bool	
	7.	<i>reset possible</i> {возм возврат}	(ref file <i>f</i>) bool	выдает true, если книга, открытая на файле <i>f</i> , сжимаема
	8.	<i>set possible</i> {возм установка}	(ref file <i>f</i>) bool	
	9.	<i>reidf possible</i> {возм переобозначение}	(ref file <i>f</i>) bool	выдает true, если на файле <i>f</i> возможен возврат
	10.	<i>chan</i> {канал}	(ref file <i>f</i>) channel	
	11.	<i>on logical file end</i> {при конце лог файла}	(ref file <i>f</i> , proc (ref file) bool <i>p</i>) void	Процедуры реакции
	12.	<i>on physical file end</i> {при конце физ файла}	(ref file <i>f</i> , proc (ref file) boo <i>l p</i>) void	
	13.	<i>on page end</i> {при конце страницы}	(ref file <i>f</i> , proc (ref file) bool <i>p</i>) void	

№	Идентификатор процедуры	План	Комментарий
14.	<i>on line end</i> {при конце строки}	(ref file <i>f</i> , proc (ref file) bool <i>p</i>) void	
15.	<i>on format end</i> {при конце формата}	(ref file <i>f</i> , proc (ref file) bool <i>p</i>) void	
16.	<i>on value error</i> {при ошибке значения}	(ref file <i>f</i> , proc (ref file) bool <i>p</i>) void	
17.	<i>on char error</i> {при ошибке литеры}	(ref file <i>f</i> , proc (ref file, ref char) bool <i>p</i>) void	
18.	<i>char number</i> {номер литеры}	(ref file <i>f</i>) int	Запросы к позиции выдает номер текущей литеры внутри текущей строки и страницы
19.	<i>line number</i> {номер строки}	(ref file <i>f</i>) int	выдает номер текущей строки внутри текущей страницы
20.	<i>page number</i> {номер страницы}	(ref file <i>f</i>) int	выдает номер текущей страницы
21.	<i>open</i> {открыть}	(ref file <i>file</i> , string <i>idf</i> , channel <i>chan</i>) int	Процедуры открытия и закрытия файлов открывает книгу идентификацией <i>idf</i> на файле <i>file</i> через канал <i>chan</i> ; выдает целое, показывающее причину неоткрытия (0, если книга открыта)
22.	<i>establish</i> {завести}	(ref file <i>file</i> , string <i>idf</i> , channel <i>chan</i> , int <i>p</i> , <i>l</i> , <i>e</i>) int	заводит пустую книгу с идентификацией <i>idf</i> , с <i>p</i> страницами по <i>l</i> строчек в каждой из <i>s</i> литер в каждой; открывает эту книгу на файле <i>file</i> через канал <i>chan</i> , результат—как в <i>open</i>
23.	<i>create</i> {создать}	(ref file <i>file</i> , channel <i>chan</i>) int	заводит пустую книгу с пустой идентификацией и максимальным размером, допустимыми для канала <i>chan</i> ; открывает эту книгу на файле <i>file</i> через канал <i>chan</i> , результат—как в <i>open</i>
24.	<i>associate</i> {соединить}	(ref file <i>file</i> , ref [] [] char <i>sss</i>) void	соединяет массив <i>sss</i> с файлом <i>file</i>
25.	<i>close</i> {закрыть}	(ref file <i>file</i>) void	закрывает книгу, открытую на файле <i>file</i> ; книга остается доступной для немедленного открытия

26.	<i>lock</i> {заблокировать}	(ref file <i>file</i>) void	закрывает книгу, открытую на файле <i>file</i> ; книга может быть вновь открыта после некоторого системного действия
27.	<i>scratch</i> {удалить}	(ref file <i>file</i>) void	закрывает и уничтожает книгу, открытую на файле <i>file</i>
28.	<i>space</i> {вперед}	(ref file <i>f</i>) void	Процедуры размещения продвигает текущую позицию файла <i>f</i> на одну литеру вперед
29.	<i>back space</i> {назад}	(ref file <i>f</i>) void	продвигает текущую позицию файла <i>f</i> на одну литеру назад в пределах одной строки
30.	<i>newline</i> {нов строка}	(ref file <i>f</i>) void	продвигает текущую позицию на начало следующей строки
31.	<i>new page</i> {нов страница}	(ref file <i>f</i>) void	продвигает текущую позицию на начало следующей страницы
32.	<i>set</i> {установить}	(ref file <i>f</i> , int <i>p</i> , <i>l</i> , <i>c</i>) void	устанавливает текущую позицию в значение (<i>p</i> , <i>l</i> , <i>c</i>)
33.	<i>reset</i> {возврат}	(ref file <i>f</i>) void	устанавливает текущую позицию в значение (1, 1, 1)
34.	<i>set char number</i> {уст номер литеры}	(ref file <i>f</i> , int <i>c</i>) void	устанавливает текущую позицию на литеру <i>c</i> внутри текущих строки и страницы
35.	<i>whole</i> {целое}	(number <i>v</i> , int <i>width</i>) string	Процедуры перевода вид <i>number</i> (не доступный пользователю) это объединение значений видов <i>int</i> и <i>real</i> с любым числом приставок <i>long</i> и <i>short</i>
36.	<i>fixed</i> {фикс}	(number <i>v</i> , int <i>width</i> , <i>after</i>) string	
37.	<i>float</i> {плав}	(number <i>v</i> , int <i>width</i> , <i>after</i> , <i>exp</i>) string	
38.	<i>put</i> {вывод}	(ref file <i>f</i> , [] union (outtype, proc (ref file) void) <i>x</i>) void	Процедуры бесформатного обмена вид <i>outtype</i> (не доступный для пользователя)—это объединение выводимых видов (см. §9)
39.	<i>get</i> {ввод}	(ref file <i>f</i> , [] union (intype, proc (ref file) void) <i>x</i>) void	вид <i>intype</i> (не доступный для пользователя) —это объединение вводимых видов (см. § 9)

№	Идентификатор процедуры	План	Комментарий
			Процедуры форматного обмена
40.	<i>putf</i> {Ф вывод}	(ref file <i>f</i> , [] union (outtype, format) <i>x</i>) void	
41.	<i>getf</i> {Ф ввод}	(ref file <i>f</i> , [] union (intype, format) <i>x</i>) void	
			Процедуры двоичного обмена
42.	<i>put bin</i> {дв вывод}	(ref file <i>f</i> , [] outtype <i>x</i>) void	
43.	<i>get bin</i> {дв ввод}	(ref file <i>f</i> , [] intype <i>x</i>) void	
44.	<i>print, write</i> {печ, зап}	([] union (outtype, proc (ref file)	Процедуры обмена на стандартных файлах
		void) (<i>x</i>) void	бесформатный вывод на файле
45.	<i>read</i> {чит}	([] union (intype, proc (ref file)	<i>stand out</i>
		void) (<i>x</i>) void	бесформатный ввод на файле
46.	<i>printf, writef</i> {Ф печ, Ф зап}	([] union (intype, format) <i>x</i>) void	<i>stand in</i>
47.	<i>readf</i> {Ф чит}	([] union (outtype, format) <i>x</i>) void	форматный вывод на файле
		([] union (intype, format) <i>x</i>) void	<i>stand out</i>
48.	<i>write bin</i> {дв зап}	([] union (intype, format) <i>x</i>) void	форматный ввод на файле
49.	<i>read bin</i> {дв чит}	([] outtype <i>x</i>) void	<i>stand in</i>
50.	<i>char in string</i> {литера в строке}	([] intype <i>x</i>) void	двоичный вывод на файле
51.	<i>make conv</i> {задать код}	(char <i>c</i> , ref int <i>i</i> , string <i>s</i>) bool	<i>stand back</i>
52.	<i>make term</i> {задать стопстроку}	(ref file <i>f</i> , proc (ref book) conv <i>c</i>) void	двоичный ввод на файле
53.	<i>reidf</i> {переобознач}	(ref file <i>f</i> , string <i>t</i>) void	<i>stand back</i>
		(ref file <i>f</i> , string <i>idf</i>) void	Прочие процедуры
			выдает true, если литера <i>c</i> содержится в строке <i>s</i> ; в <i>i</i> помещается номер первого вхождения

вводе все спецификации игнорируются и значение вводится процедурой *get*.

Шаблон форматного позволяет внутри формата обращаться к другим форматам. При выборке шаблона форматного исполняется содержащееся в нем выдающее формат закрытое предложение. Управление редактированием передается полученному формату. При исчерпании нового формата управление возвращается старому формату.

Содержащиеся в тексте формата повторители могут быть статическими (целое) или динамическими (символ *n*, закрытое предложение).

Перед обменом под контролем шаблона осуществляется подготовка данного шаблона: совместно исполняются содержащиеся в нем закрытые предложения и каждый динамический повторитель заменяется выдачей соответствующего закрытого предложения.

Шаблоны в тексте формата могут объединяться в повторяемые наборы. Стоящий перед набором повторитель определяет сколько раз должен просматриваться набор при выборке шаблонов.

При двоичном обмене представление значений в книге недоступно пользователю. Значение, записанное в книгу процедурой двоичного вывода, может быть прочитано только лишь процедурой двоичного ввода.

Часть 3

СТАНДАРТИЗАЦИЯ

§ 10. СТАТУС ПУБЛИКАЦИЙ

Пересмотренное сообщение RR принято Рабочей группой 2.1, утверждено техническим комитетом 2 по Программированию и одобрено для публикации Генеральной Ассамблеей Международной федерации по обработке информации (ИФИП). Аналогичный статус имеет стандарт SHR.

Русский вариант ПС принят временной научно-технической комиссией по АЛГОЛу 68, одобрен Междуведомственной научно-технической комиссией по программному обеспечению ЭВМ и утвержден в качестве определяющего документа Государственным комитетом Совета Министров СССР по науке и технике.

Терминология ПС рекомендована для преимущественного употребления в публикациях по АЛГОЛу 68. Она используется и в данном обзоре.

Стандарт SHR на машинное представление АЛГОЛа 68 уточняет RR. Он призван унифицировать конкретные представления языка в различных реализациях и облегчить решение проблемы переносимости программ.

История создания SHR описана в [117]. SHR указывает внешний вид программы (вместо попытки указать внутренний вид). SHR применим при любой кодировке литер.

§ 11. ОСНОВНЫЕ ПОНЯТИЯ СТАНДАРТА SHR

Значащая литера — это одна из следующих 60 литер:

ABCDEFGHIJKLMN~~OP~~QRSTUVWXYZ

0 1 2 3 4 5 6 7 8 9

пробел "#\$ %' () * +, - . / : ; < = > @ [] - |

Представление программы на АЛГОЛе 68 определяется как последовательность значащих литер и новых строчек. Любой символ, представление которого в ПС соответствует значащей литере, представляется этой литерой.

Базисная литера — это некоторая «литера», имеющаяся на устройстве подготовки данных.

Разделитель — это особенность типографского набора, начало или конец текста программы или любая значащая литера, отличная от буквы, цифры или подчеркивания.

Две строки литер «прилегают», если между ними нет литер или типографских особенностей. Если говорится, что одна из строк «следует за» или «предшествует» другой, то они также прилегают.

Слово разбивается особенностями типографского набора и подчеркиваниями на подслова. Подслово — непустая последовательность букв или цифр. Слова различаются только тогда, когда различны конкатенации их подслов (например, «нос ок» и «но сок» эквивалентны). Выделенное слово не содержит особенностей типографского набора и подчеркиваний (см. § 12).

Для каждой значащей литеры реализация должна предусмотреть базисную литеру, отличающуюся от базисных литер для других значащих литер.

Реализация может присоединить к значащим литерам 26 (латинских) букв малого регистра. Два регистра букв эквивалентны, но не в строках и не при режиме UPPER (см. § 12).

ПС (и RR) предусматривает, что «конструкт в языке представления» получается заменой символов на их представления. В SHR предусмотрено представление для каждого символа в терминах значащих литер. Конструкты в языке представления кодируются для передачи и машинной обработки заменой каждой значащей литеры на соответствующую ей базисную литеру и вставкой особенностей типографского набора (там, где разрешено).

§ 12. ОТДЕЛЬНЫЕ ПРЕДСТАВЛЕНИЯ

Элементы строки. Множество элементов строки — это множество значащих литер (возможно дополненное регистром малых букв), исключая кавычку и апостроф, но включая 'символ образ кавычки' и 'символ образ апострофа'. Естественное значение каждой значащей литеры есть она сама; верхние и нижние регистры букв имеют различные естественные значения. 'Символ образ кавычки' записывается двумя прилежащими кавычками, и его естественным значением является кавычка. 'Символ образ апострофа' записывается двумя прилежащими апострофами, и его естественным значением является апостроф. Один апостроф может использоваться в реализациях для введения новых режимов.

Предусмотрена особенность типографского набора 'разрыв строки' для использования внутри изображений строк и литерных. Она записывается как

- кавычка, с последующими
- одной или более особенностями типографского набора, отличными от разрыва строки, с последующей
- еще одной кавычкой.

Когда изображение строки должно быть размещено на нескольких строчках, разрыв строки позволяет указывать количество пробелов в конце одной строчки и, не создавая двусмысленностей, определяет расположение следующей.

Прагматы. Прагматы (см. табл. 3), как и комментарии, не влияют на результат исполнения программы. Они призваны указать способ более удобного исполнения программы (например, с отменой какого-либо контроля). Прагматы также служат для указания на способ компиляции программы.

В SHR предусмотрены четыре стандартных элемента прагматов: PAGE, POINT, UPPER и RES. Во всех реализациях эти элементы должны распознаваться хотя бы в минимальной форме: $\cdot PR_{\pm}$ элемент $\cdot PR_{\pm}$. Каждый из этих четырех элементов — прагматов записывается как последовательность букв большого регистра, причем ему могут предшествовать или его могут завершать особенности типографского набора. Прагмат PAGE обеспечивает переход на новую страницу при распечатке программы. Остальные прагматы описаны ниже.

Особенности типографского набора. Особенности типографского набора являются пробел, новая строчка и разрыв строки. Новая строчка может быть одной базисной литерой или физическим явлением, подобным концу записи.

Слова и выделенные слова. Представление слов и выделенных слов определяется одним из трех «выделяющих режимов». Новый режим вводится прагматом, содержащим один из элементов прагмата POINT, UPPER и RES. Иногда приводимые ниже правила требуют наличия разделителя в некоторой позиции.

При необходимости, в качестве разделителей можно использовать особенности типографского набора.

Режим POINT. Выделенное слово начинается с “.”, за которой следуют соответствующие значащие буквы и цифры. Выделенное слово должно завершаться разделителем. Слово составляется из последовательности одного или более подслов, между которыми возможны особенности типографского набора. Подслово составляется из последовательности соответствующих значащих букв и цифр, за которой может следовать подчеркивание (подчеркивание, как и особенность типографского набора, не учитывается при определении тождественности слов). Если подслово не завершается подчеркиванием, то после него должен следовать разделитель.

Режим UPPER. Слова и выделенные слова представляются как в режиме POINT, но с использованием дополнительных правил:

— Верхние и нижние регистры букв не могут перемешиваться в выделенных словах.

— Точка может опускаться перед выделенным словом из больших букв, если ей предшествует разделитель, отличный от точки, малая буква или цифра, не являющаяся «большой цифрой». «Большая цифра» — это цифра, которой предшествует большая буква или большая цифра.

— За большим выделенным словом не обязательно ставить разделитель, если за ним следует малая буква.

— Большие буквы могут использоваться только в выделенных словах и в изображениях строк и литерных, а также в пояснениях.

Режим RES. «Зарезервированное слово» — это одно из выделенных в RR как представление для некоторого символа. Согласно RR их нельзя пересопределять.

Список зарезервированных слов (в фигурных скобках даны их эквиваленты в русском варианте АЛГОЛа 68): at {с}, begin {начало, нач}, bits {бит}, bool {лог}, by {шаг}, bytes {слог}, case {в}, channel {канал}, char {лит}, co, comment {прим}, compl {компл}, do {цк}, elif {инес}, else {иначе}, empty {пустое}, end {конец, кон}, esac {быв}, exit {выход, вых}, false {ложь}, fi {все}, file {файл}, flex {подв}, for {для}, format {формат}, from {от}, go, goto {на}, heap {глоб}, if {если}, in {выб}, int {цел}, is {есть}, isnt {несть}, loc {лок}, long {длин}, mode {вид}, nil {нил}, od {кц}, of {из}, op {оп}, ouse {ливывб}, out {либо}, par {пар}, pr {прагм}, pragmat, prio {прио}, proc {проц}, real {вещ}, ref {имя, имени}, sema {семафор}, short {кор}, skip {скип, пропуск}, string {строк}, struct {ст, структ}, then {то}, to {до}, true {истина, ист}, union {об}, void {пуст}, while {пока}, {ф} для \$.

В режиме RES слова и выделенные слова представляются как в режиме POINT, но с использованием дополнительных правил:

— Точка может опускаться перед зарезервированным словом, если ему предшествует разделитель, отличный от точки.

— К подслову должно прилагать подчеркивание, если его буквы и цифры соответствуют, в том же порядке, буквам и цифрам некоторого зарезервированного слова.

Составные представления. Некоторые представления, приведенные в ПС, выглядят составленными из последовательности двух или более знаков, не являющихся буквами {" " , =: , :=, |: , :=: , :=:}. Эти представления являются последовательностями значащих литер, соответствующих составляющим знакам.

Обмен. Представления объектов для обмена должны использовать только значащие литеры, чтобы ввод можно было подготавливать, а вывод интерпретировать без ссылки на конкретную реализацию. Запросы к обстановке зависят от значащих литер так: *FLIP*="T", *FLOP*="F", *ERROR CHAR*="*", *BLANK*="÷". Литеры ⊥ и ₁₀ представляются значащими литерами "I" и "E".

Два регистра литер эквивалентны, если, появившись в обмене, они представляют значения, отличные от значений видов 'литерное' или 'вектор из литерных'.

§ 13. СТАНДАРТ НА РУССКИЙ ВАРИАНТ АЛГОЛА 68

На заседании Комиссии по АЛГОЛу 68 в г. Ивантеевке (март, 1977 г.) принято решение о разработке стандарта на русский вариант АЛГОЛа 68 на базе ПС (см. § 16) и SHR. Решено расширить список значащих литер 32 русскими буквами большого регистра (не включая Ё), причем сходные по начертанию русские и латинские буквы считать различными. Использование малых русских букв аналогично использованию малых латинских букв в SHR.

Для сходных по начертанию литер русского и латинского алфавитов на устройствах как правило имеется один общий код. Поэтому предполагается ввести правило, регулирующее интерпретацию таких общих кодов. В пределах подслова или выделенного слова будет разрешено использование только одного алфавита, т. е. наличие хотя бы одной буквы, не имеющей сходной буквы в другом алфавите, определяет интерпретацию остальных кодов в подслове или выделенном слове. В других случаях (в подсловах или выделенных словах, состоящих из общих кодов, в изображениях и пояснениях) интерпретация общих кодов определяется режимом преобладания алфавита. Специальными средствами (прагматами или авторегистрами) будет переключаться режим преобладания алфавитов.

Имеются некоторые трудности в разработке стандарта на русский вариант АЛГОЛа 68. Например, считать ли виды *comp1* и *компл* эквивалентными, вводить ли стилизацию текстов *фор-*

матов и изображений битовых (в RR стилизации нет), как стилизовать процедуры обмена над битовыми и логическими, как исправить некоторые ошибки (главным образом в обмене), имеющиеся в описании языка RR. Кроме того группа специалистов [233] предполагает написать раздел обмена в АЛГОЛе 68 заново, в том же духе, но более эффективно. Основным понятием в новом описании обмена должна стать не книга, а строчка, играющая роль буфера. Новый обмен с точки зрения пользователя почти не будет отличаться от имеющегося.

Ввиду небольшого опыта использования АЛГОЛа 68 (смотри, однако, [68, 81, 85, 91, 155, 227, 244, 245]) и указанных выше обстоятельств, появление стандарта на русский вариант АЛГОЛа 68, такого как [45], следует ожидать (в соответствии с планом разработки стандартов СЭВ) не ранее 1983 года. Однако стандарт на машинное представление программ на русском варианте АЛГОЛа 68 (и более общий стандарт на представление программ на национальных вариантах АЛГОЛа 68) должен быть разработан в ближайшее время.

§ 14. ПЕРЕНОСИМОЕ ПРОГРАММИРОВАНИЕ

В приложении к SHR предложен следующий метод переноса программ.

Программа пишется значащими литерами в соответствии с SHR (если реализация является расширением SHR, то рекомендуется вводить прагмат PORTCHECK, контролирующий выполнение SHR) и набивается в некотором коде. Для переноса такой программы необходима только транслитерация. Для облегчения отладки подпрограммы транслитерации публикуемые программы рекомендуется обеспечивать файлом, содержащим: 1) одну или более строчек, содержащих все используемые в программе литеры (первыми должны идти значащие литеры в том порядке, как они вводятся в стандарте); 2) описание каждой литеры.

Для реализаций, имеющих общий носитель информации (например, перфоленту), можно использовать более удобный способ переносимости программ.

Перед текстом программы (или перед данными для процедур ввода) в фиксированном порядке располагаются значащие литеры, новая строчка (одна литера) и конец текста. (Например так: <новая строчка> <значащие литеры из SHR> | <латинские буквы нижнего регистра> | <русские буквы верхнего регистра> | <русские буквы нижнего регистра> | <конец текста>, причем каждая последовательность букв, стоящих между вертикальными чертами, может опускаться (если их нет на устройстве)). См. также Maslov A. N., Algol Bull., 1977, № 41, 7.

Можно написать универсальный перекодирующий, который,

получив так оформленную программу, выясняет ее кодировку и транслирует ее в местный код.

Однако указанный способ переносимости трудно унифицировать для всех устройств.

Замечание. На АЛГОЛе 68 возможно написание программ, которые на разных ЭВМ выполняются по-разному, например:

1) *if max int = 1023 then print (1) else print (2) fi* ϕ напечатается 1, если для представления целых используется 10 двоичных разрядов, а иначе напечатается 2 ϕ

2) *if abs «a» = 18 then print (1) else print (2) fi* ϕ напечатается 1, если литер «a» имеет код 18, а иначе напечатается 2 ϕ

3) *(print (3 elem 2r10))* ϕ напечатается *t*, если *bits width = 4*, а иначе напечатается *f* ϕ

Пример 3 показывает, что семантика изображения битового неудачна.

Однако подобные примеры носят экзотический характер, и переносимые программы легко пишутся на АЛГОЛе 68. Конечно, было бы желательно иметь формальный метод определения переносимости программ, но такой метод неизвестен.

Часть 4

ОПИСАНИЕ ЯЗЫКА

§ 15. ГРАММАТИКИ ВАН ВЕЙНГААРДЕНА

Синтаксис АЛГОЛа 68 описан с помощью предложенного ван Вейнгаарденом аппарата, более мощного, чем бесконтекстные грамматики.

В грамматике ван Вейнгаардена (используется также название *W*-грамматика) можно построить (в общем случае бесконечное) множество порождающих правил (таких же, как и в бесконтекстной грамматике), используя заданный конечный набор вспомогательных правил. (В отличие от публикаций по теории формальных грамматик нетерминалы в описании АЛГОЛа 68 называются понятиями, а терминалы символами.)

Часть вспомогательных правил называется метаправилами, а другая часть гиперправилами.

Метаправила образуют метаграмматику (бесконтекстную, но без начального понятия). Понятия метаграмматики называются метапонятиями. В RR и ПС они пишутся большими буквами, чтобы отличать их от символов метаграмматики, которые пишутся малыми буквами. Пусть *M* множество метапонятий. Предполагается, что каждое слово из *M*^{*} имеет единственное разбиение на слова из *M* (это свойство алгоритмически проверяемо).

Символами метаграмматики являются все малые буквы. Начав с некоторого метапонятия (как начального понятия грамматики) и применяя некоторую последовательность метаправил (пока это возможно), мы получим последовательность малых букв, называемую терминальным порождением этого метапонятия.

Гиперправила устроены аналогично правилам обычной бесконтекстной грамматики, но гиперпонятия (т. е. понятия из гиперправил) содержат малые буквы и метапонятия. Чтобы из гиперправила получить правило, нужно вместо каждого метапонятия подставлять некоторое его терминальное порождение, одно и то же вместо каждого вхождения данного метапонятия в гиперправило (в левую и правую часть гиперправила).

Вывод в грамматике ван Вейнгаардена ведется из некоторого начального понятия (понятия 'program' для RR или понятия 'программа' для ПС), используя правила (способ получения которых только что указан) до тех пор, пока не получится последовательность символов. Символы — это последовательности малых букв, заканчивающиеся буквами «symbol» в RR или начинающиеся буквами «символ» в ПС. Чтобы из порожденной таким образом последовательности получить реальную программу, нужно каждый символ заменить на его представление, например, becomes symbol заменить на := или long symbol заменить на long. Исключения представляют индикаторы, которые заменяются на подчеркнутую (или выделенную другим способом) последовательность представляющих индикатор букв и цифр. Определение представления для машинных носителей см. в §§ 11 и 12.

Множество символов не пересекается с множеством понятий, т. е. символы не могут встретиться в левой части правила.

Термином «конструкт» в RR, в ПС и в нашем изложении обозначается дерево (и соответствующая часть программы) порождения некоторого понятия (в дереве вывода данной грамматики). Смысл программы в RR и ПС объясняется посредством действий, которые конструкт предписывает совершить своим подконструктам.

Отметим, что не существует алгоритма, который по грамматике ван Вейнгаардена мог бы определить, порождает ли она хотя бы одну программу, и не существует алгоритма, который бы мог определить, порождается ли данная программа данной грамматикой ван Вейнгаардена [218]. Таким образом, грамматики ван Вейнгаардена оказывают мало помощи при создании транслятора с АЛГОЛА 68.

Причина отсутствия указанных алгоритмов заключается в том, что возможны два разных гиперпонятия, порождающих одно и то же понятие. В грамматике из RR или ПС такие гиперпонятия имеются, например:

'ЮПИСАНИЯ' и 'ЮПИСАНИЯ ЮПИСАНИЯ'.

Следует заметить, что не каждое понятие, полученное из какого-либо гиперпонятия, порождает последовательность символов (например, 'изображение имени целого'). Такие понятия, не имеющие терминальных порождений, называются тупиками (и в RR и в ПС понятиями называются только последовательности малых букв, не являющиеся тупиками).

Кроме гиперпонятий, служащих для получения понятий, порождающих последовательности символов (программы), в синтаксисе языка имеются предикаты (также являющиеся гиперпонятиями), из которых получаются либо понятия с пустыми терминальными порождениями (в этом случае предикат выполняется), либо тупики (предикат не выполняется). Таким образом, если во время вывода для некоторого понятия встречается предикат, то продолжение вывода возможно только тогда, когда этот предикат выполняется.

Использование предикатов в RR (в R их не было) позволило с одной стороны синтаксически описать всю статическую семантику языка (идентификацию, эквивалентность видов и т. д.), а с другой — сократить число его синтаксических правил и сделать многие из них более простыми и наглядными. Вместе с тем предикаты увеличивают «неоднозначность», см. ниже.

Ряд работ посвящен теоретическим исследованиям *W*-грамматик. В [218] показано, что *W*-грамматики порождают все рекурсивно-перечислимые языки и, следовательно, их нельзя использовать при построении анализирующей части компилятора.

В работах [59, 61, 156, 182] выделены подклассы *W*-грамматик, также порождающих все рекурсивно-перечислимые языки, и проводится сравнение *W*-грамматик с другими типами грамматик. Сравнение *W*-грамматик с Венским методом проведено в [101].

В работах [94, 95] рассматриваются подклассы *W*-грамматик, порождающих классы языков с разрешимой проблемой принадлежности.

Неразрешимость проблемы принадлежности для *W*-грамматик (а также неудобство чтения ряда мест в RR) связана с неоднозначностью понятий, т. е. с возможностью породить некоторое понятие двумя различными способами (из разных гиперпонятий, либо из одного гиперпонятия, но с помощью подстановки вместо некоторых метапонятий разных терминальных порождений). В работах [31, 32] рассмотрены не порождающие неоднозначных понятий *W*-грамматики, названные *UW*-грамматиками. Показано, что класс *UW*-грамматик эквивалентен классу \mathcal{S}_3 грамматик, который является расширением класса индексных грамматик. Для класса \mathcal{S}_3 , и как следствие,

для UW -грамматик, доказана разрешимость проблемы принадлежности.

Однако алгоритм преобразования UW -грамматик в \mathcal{E}_3 -грамматики достаточно сложен. В настоящее время нет необходимости в его реализации, так как неясно, будут ли W -грамматики использоваться для описания других языков программирования (кроме АЛГОЛа 68 W -грамматики применялись для описания языка ЛИСП [213]).

В настоящее время в реализациях АЛГОЛа 68 используется описание синтаксиса с помощью бесконтекстных грамматик [2, 10, 72, 79, 91].

§ 16. РУССКИЙ ВАРИАНТ АЛГОЛА 68

Создание АЛГОЛа 68 — русского варианта языка ALGOL 68 — включило в себя разработку: терминологии, русских вариантов представлений символов, стандартных операций и процедур, описания языка.

Разработка описания АЛГОЛа 68 не явилась простым переводом RR, поскольку последнее существенно ориентировано на английский язык (например, формальное использование синтаксиса языка ALGOL 68 основано на неизменяемости английских слов в предложениях, а описание вида `comp1` явно содержит указатели полей *re* и *im*). В связи с этим потребовалась модификация метода описания синтаксиса, позволившая использовать в качестве синтаксических правил естественные фразы русского языка. Основная идея этой модификации заключается в том, что каждое метапонятие и гиперпонятие выглядит как грамматически правильная русская фраза, а при выводе используется ее «приведенная форма», полученная заменой всех содержащихся в этой фразе изменяемых слов на их словарную форму (т. е. именительный падеж, единственное число, и, если возможно, средний род); так, например, «формальный описатель ЗНАЧЕНИЯ В СРЕДЕ» заменяется на «формальный описатель ЗНАЧЕНИЕ в СРЕДА». Для предотвращения путаницы при использовании метапонятий, приведенной формой которых служат слова во множественном числе, введен особый знак «!», что позволяет отличать метапонятие «ЗНАЧЕНИЯ» (множественное число) и «ЗНАЧЕНИЯ» (родительный падеж для «ЗНАЧЕНИЕ»).

При разработке русских представлений символов использовалась обеспечиваемая синтаксисом стилизация, которая предписывает одинаковое «оформление» конкретной конструкции, но не ограничивает число стилей оформления. Например, «замкнутое предложение» может охватываться скобками «(» и «)», «begin» и «end», «начало» и «конец», «нач» и «кон» (но не «begin» и «конец», в чем и состоит смысл стилизации), а «условное предложение» может использовать как разделители «если»,

«то», «иначе» и т. д., так и разделители «if», «then», «else» и т. д.

Кроме того было заменено порождающее правило для метапонятия 'АЛФАВИТ' (что допускается синтаксисом), в результате чего появилась возможность использовать в представлении программы на АЛГОЛе 68 как латинские, так и русские буквы. Для нестилизуемых символов, обозначений стандартных операций и процедур были введены альтернативные русские представления. Дополнительные стилизованные символы и альтернативные представления перечислены в §§ 8, 9 и 13. Введен дополнительный стандартный вид компл с указателями полей «вч» и «мч», семантически эквивалентный виду `comp1`, с соответствующими стандартными операциями. В изображениях и текстах формата латинские буквы *a, b, c, d, e, f, g, i, k, l, n, p, q, r, s, x, y* и *z* можно заменить на русские *a, б, ц, д, е, ф, г, и, к, л, н, п, ю, я, ш, х, у* и *ж* соответственно*.

Таким образом АЛГОЛ 68 — это надъязык языка ALGOL 68; любая собственно программа языка ALGOL 68 является таковой и на АЛГОЛе 68, однако на АЛГОЛе 68 можно писать и программы с использованием только русских букв.

Часть 5

ПРОБЛЕМЫ РЕАЛИЗАЦИИ

§ 17. ПЕРЕЧЕНЬ ИЗВЕСТНЫХ РЕАЛИЗАЦИЙ

Ниже приведены краткие сведения относительно некоторых реализаций АЛГОЛа 68.

1. **ALGOL 68 R.** Диалект АЛГОЛа 68, названный АЛГОЛ 68 R [91, 245], реализован в 1970 году в Королевском Радарном Обществе (Англия) для ICL 1900. Компилятор односмотровый, занимает 36 К 24 разрядных слов. Этот компилятор активно использовался и программисты, имевшие с ним дело, высказывали удивление, как это они раньше обходились без таких видов, как структура и имена. АЛГОЛ 68 R преподается в Англии во многих университетах. Опыт реализации АЛГОЛа 68 R оказал определенное влияние на разработку RR (см. также [155]).

2. **ALGOL 68 MBLE.** Компилятор [80, 81, 82] для Electrologica-X8 с языка, почти не отклоняющегося от R, вступил в строй в фирме MBLE (Бельгия) в 1975 году. Он перенесен также на BS и UNIDATA 7720. Время компиляции N страниц (по 60 карт) оценивается в $5+6N$ сек. Вся система (включая сервис) занимает 100 К 27 разрядных слов. Основное внимание

* Указанное соответствие возможно будет пересматриваться.

при разработке было уделено созданию эффективной рабочей программы [7]. Компилятор ALGOL 68 MBLE оказался эффективнее, чем компилятор с АЛГОЛа 60 для Electrológica-X8.

3. **MARY.** Значительное распространение получил язык MARY — вариант АЛГОЛа 68 для мини-машин [197, 198, 199]. Создана ассоциация пользователей языка MARY.

4. Задачей Пражской реализации [153, 181, 183] подъязыка АЛГОЛа 68 было создание компилятора для TESLA 200, более эффективного, чем имеющийся компилятор с КОБОЛа. В подъязыке ограничено использование круглых скобок, чтобы улучшить анализ ошибок.

5. **ALGOL 60+8.** Описание реализации подъязыка АЛГОЛа 68, близкого к АЛГОЛУ 60, приведено в [22, 142].

6. Переносимый компилятор [146] с АЛГОЛа 68 написан на языке реализаций CDL [150] студентами Мюнхенского технического университета.

7. **ALGOL 68/19.** Экспериментальный компилятор [107] с подъязыка АЛГОЛа 68 (без объединений, структур и форматов, без описаний видов и описаний операций, без глобальной памяти, с ограниченными вырезками и более примитивными процедурами обмена), используемый для обучения слушателей Королевской военной школы (Брюссель), реализован на IBM 360/30. Скорость компиляции у него в 10 раз выше, чем у D-компилятора с ПЛ/1. Предполагается расширение реализации.

8. Подъязык французского варианта АЛГОЛа 68 реализован [227] для Univac 1100 (супервизор EXEL VIII). Подъязык обладает следующими особенностями. Требуется, чтобы описания предшествовали использованию. Число приставок long не более одной. Использован упрощенный вариант эквивалентности видов (при котором два эквивалентных вида (§ 16), использующих рекурсию, могут оказаться различными). Стандартное вступление реализовано не полностью, но оно расширяемо. Реализованы подвижные массивы. Реализована раздельная компиляция процедур.

Компилятор занимает 20 000 карт на ФОРТРАНе. Для его работы требуется 65 К слов памяти.

Представление программ в этом компиляторе повлияло на разработку SHR.

9. Переносимый компилятор [206] полуинтерпретационного типа с подъязыка АЛГОЛ 68 (основы не могут чередоваться с описаниями, нет структур и объединений, ограничены имена и процедуры, нет описаний видов и операций, нет глобальной памяти) для IBM 1130 разработан в Оклахомском университете. Компилятор написан на ФОРТРАНе. Для его работы требуется 16 К оперативной памяти.

10. Подъязык АЛГОЛа 68 реализован [50, 48, 49] для CONTROL DATA 6000 и CYBER. Подъязык ориентирован на численные задачи (устранены структуры и объединения).

11. Компилятор с полного АЛГОЛа 68 (включая обмен) для IBM 370, с хорошей диагностикой (включая динамическую диагностику) и приемлемым временем компиляции, предназначенный для обучения студентов, написан в Саскатунском университете (Канада) [85].

12. **ALGOL 68 H.** Заслуживающий изучения компилятор с синтаксически полного (согласно RR) АЛГОЛа 68 написан в 1972—1975 гг. Х. Боомом [71, 72]. Анализатор компилятора построен на основе аффиксной трансдукционной грамматики, но аффиксы отличаются от введенных в [148], а трансдукционные правила отличаются от введенных в [159].

Процесс компиляции состоит из пяти этапов. Сканер заносит слова и выделенные слова в таблицу имен (каждое в одном экземпляре), разбирает изображения и обозначения для операций, выделяет блочную структуру, заносит в блочные таблицы описания приоритетов, видов и операций. Анализатор основан на LR(1)-грамматике; так как идентификация еще не проведена, некоторые конструкции не различаются. Обработка видов сохраняет из эквивалентных видов только один экземпляр. Для приведенных и идентификации используется h -функция (см. [70] и § 18). Генерация кода в публикациях не описана.

Промежуточные представления хранятся в оперативной памяти, поэтому для компиляции реальных программ требуется не менее 400 К байтов памяти. Компилятор «ALGOL 68 H» написан на АЛГОЛе W.

13. **ALGOL 68 C.** Переносимый компилятор с АЛГОЛа 68 написан в Кембриджском университете [68]. Он реализован на IBM 360, IBM 370, PDP 11/45 и ICL 4130. У компилятора хорошая диагностика, но статический контроль области действия (§ 19) разработчики сочли излишеством. Компилятор используется для обучения студентов; на нем пишется операционная система для экспериментальной ЭВМ, а также система для работы в реальном времени. Для работы системы требуется 120—150 К байтов памяти.

14. В [62] описана структура компилятора, написанного с помощью событий и ко-процедур. Нехватка информации вызывает прерывание ко-процедур. При написании компилятора было мало ошибок. Расход памяти оказался непродуктивным, но время компиляции приемлемым.

15. Имеются сведения о поставках компиляторов с АЛГОЛа 68 фирмами IBM и CDC.

16. Исследования [174, 176, 177, 178, 179] проводились в связи с разработкой переносимого компилятора с полного АЛГОЛа 68.

17. Компилятор полуинтерпретационного типа с языка, промежуточного между R и RR, для M4030 разработан в ИВЦ Киевского завода электронных вычислительных и управляющих машин. Компилятор ориентирован на решение экономических задач. Память под массивы он выделяет на внешних устройствах, чем в

значительной мере компенсирует отсутствие базы данных в АЛГОЛе 68. Компилятор проходит опытную эксплуатацию в ряде организаций.

18. Компилятор с полного АЛГОЛа 68, включая обмен, разрабатывает Ленинградский университет для ЕС ЭВМ. Анализ в компиляторе основан на построении видонезависимой грамматики АЛГОЛа 68, допускающей распознавание двухпросмотровым (один прямой, другой обратный) магазинным автоматом. Основное внимание в реализации уделено построению эффективной рабочей программы [39, 40]. В этом отношении на разработку некоторое влияние оказал компилятор «ALGOL 68 MBLE». Использованный метод распределения памяти позволяет эффективно реализовать подвижные массивы и параллельные процессы. В реализации предусмотрены средства отладки и исследования программ.

Компилятор проходит экспериментальную эксплуатацию. Структура компилятора подробно описана в [2, 5, 38].

19. Компилятор с полного АЛГОЛа 68 разрабатывается для ВК «Эльбрус 1». Компилятор создает эффективную рабочую программу. Предусмотрено создание развитой диагностики.

20. В [28] изложен проект реализации АЛГОЛа 68 на АЛГОЛе 68 (с раскруткой), ориентированный на решение экономических задач. Проект отличается необычным механизмом распределения памяти, сходным с используемым в реализациях СИМУЛы 67.

§ 18. КОНТРОЛЬ ВИДОВ

Контроль видов является специфической чертой компиляторов с АЛГОЛа 68, значительно ускоряющей процесс отладки.

1. Эквивалентность видов. Два вида эквивалентны, если совпадают (при некоторой перестановке коммутативных стрелок) их, вообще говоря бесконечные, видовые деревья.

Так, виды i и j , где

$$\text{mode } i = \text{struct } (\text{int } a, \text{ref } i \ b);$$
$$\text{mode } j = \text{struct } (\text{int } a, \text{ref struct } (\text{int } a, \text{ref } j \ b));$$

эквивалентны, хотя и имеют разные описатели. В R эквивалентность видов не была точно определена [229], в работах [147, 173, 247] эта погрешность устранена и сформулирована концепция вида, использованная в RR.

Совпадение (даже бесконечных) видовых деревьев можно проверить с помощью следующего алгоритма.

Занумеруем все описатели и подописатели, используемые в программе. Пусть их всего N штук.

Для каждого целого i можно определить отношение эквивалентности i на множестве видовых деревьев: $A \sim_i B$, если, обрубив A и B на i -том ярусе (т. е. отбросив все вершины, лежащие от корня на расстоянии большем i), мы получим совпадающие

(с точностью до перестановки коммутативных стрелок) дерева. В частности, $A \sim B$, если корни деревьев A и B одинаково помечены.

Пусть $D(i) = \{D_1(i), \dots, D_{|D(i)|}(i)\}$ классы эквивалентности отношения i на множестве видовых деревьев описателей и подописателей из данной программы, а $|D(i)|$ — число элементов в $D(i)$. Ясно, что $|D(i+1)| \geq |D(i)|$. Алгоритм основывается на следующем утверждении.

Если $D(i) = D(i+1)$, то и $D(i+1) = D(i+2)$.

При написании алгоритма эквивалентности видов следует учитывать некоторые особенности стандартных видов (например, виды `long compl` и `struct (long real re, im)` эквивалентны, а виды `long long int` и `long long long int` неэквивалентны), а также учитывать отличие виртуальных и формальных описателей.

Практически более удобные алгоритмы эквивалентности изложены в [2, 247].

2. Реализация объединенных видов. Для эффективной реализации объединенных видов при независимой компиляции необходимо определение порядка на видах. Алгоритмы упорядочивания видов изложены в работах [2, 8].

После объединения значение удобно хранить вместе с ярлыком (номером представителя в объединении в том, например, порядке, который определяется сравнением минимальных видовых графов). При дообъединении ярлык будет изменяться, а при исполнении сопоставляющего предложения выбор будет определяться по ярлыку.

Предположим теперь, что имеется текст процедуры, вид одного из параметров которого начинается на `union` и в котором не используется описаний из внешних блоков.

Такой текст процедуры может быть откомпилирован независимо от объемлющей программы (и может использоваться затем в нескольких объемлющих программах).

В настоящее время находятся в стадии обсуждения проекты унификации более сильных способов введения независимой компиляции [73, 74, 90, 163, 164, 165, 222].

При использовании независимо откомпилированной процедуры требуется, чтобы ярлык фактического параметра, подаваемого этой процедуре при вызове, был таким же, каким он предполагался при компиляции. Алгоритмы упорядочивания [2, 8] обеспечивают такое совпадение ярлыков, т. к. способ задания описателя не оказывает влияния на определяемый порядок.

3. Правильность описателей. Используя алгоритм эквивалентности видов, легко определить последовательность приведенный между видами (если таковая имеется). Но в R при использовании объединения в приведениях имелась двусмысленность. Например, вид `ref union (int, real)` можно привести двумя способами к виду `union (int, real, ref union (int, real))`, а именно: мож-

но либо дообъединить, либо сначала разыменовать, а потом дообъединить.

Чтобы устранить подобные двусмысленности в RR введено ограничение: каждый представитель объединения не может приводиться в позиции *k4* к другому представителю или к объединению некоторого числа других представителей.

Указанное ограничение, вместе с ограничениями на конструирование описателей, запрещающими описания типа `mode a = = ref a` (см. §§ 2, 3), устраняет все двусмысленности в приведенных.

4. Крепкая связанность. Два вида m_1 и m_2 называются крепко связанными, если имеется вид, приводимый в позиции *k4* и к виду m_1 , и к виду m_2 . Можно проверить, что два вида крепко связаны тогда и только тогда, когда один из них сам, или его представитель в позиции *k4* приводим к другому.

Обозначение операции может быть описано в одном блоке несколько раз (в отличие от всех других типов описаний), но с разными видами операндов. Требуется, чтобы виды операндов не были крепко связаны, иначе могли бы встречаться формулы, в которых неясно, какую из двух одинаково обозначенных операций исполнять, например, программа `(op + = (ref int a) - int : a, + = (int a) int : abs a; int x; +x)` недопустима из-за крепкой связанности операндов в двух описаниях операции.

На первый взгляд представляется, что в случае уравнивания в операнде при идентификации операции, обозначение которой описано несколько раз, необходим перебор всех вариантов уравнивания. Но существует метод, предложенный Х. Боомом [70], в котором из уравниваемых видов ИВМ выбирает «главный» вид, и по нему ведется идентификация операции. Выбранный «главный» вид во всех случаях приводится в позиции *k4*, а остальные виды в позиции *k5*.

Для того чтобы метод Боома осуществлялся, в RR введено специальное ограничение: каждое описание операции из данного блока делает «недоступными» внутри данного блока (и всех его подблоков) операции из внешних блоков с тем же обозначением и крепко связанными (с данной операцией) видами операндов.

Если бы указанного ограничения не было, то при уравнивании видов `union (int, real)` и `union (int, bool)` в операнде унарной операции `+` ни один из них нельзя было бы объявлять «главным». Действительно, пусть «главным» считается вид `union (int, real)`; тогда при наличии в данном блоке унарной операции `+` с операндом вида `union (int, real)` и описания унарной операции `+` во внешнем блоке с операндом вида `union (int, real, bool)` идентификация операции по виду `union (int, real)` будет произведена неправильно. Выбор другого вида в качестве главного приводит к аналогичной ошибке, если описание опера-

ции + в данном блоке имеет операнд вида `union (int, bool)`, а во внешнем как выше.

5. Правила идентификации. Опишем правила идентификации в АЛГОЛе 68. Идентификация индикаторов сложнее, так как (в отличии от индикаторов) индикатор операции (но не индикатор вида или приоритета) может быть описан в пределах блока дважды. Если мы ищем описание некоторого идентификатора, то мы движемся по блоку (исключая вложенные блоки), затем переходим в объемлющий блок, и т. д. Если же мы ищем описание некоторого индикатора, то

— если мы встречаем описание вида с нужным индикатором, а нам нужен (по смыслу использования) индикатор приоритета или операции, то в программе ошибка;

— если мы ищем описание вида, а встречаем описание приоритета или операции с нужным нам индикатором, то в программе ошибка;

— если мы ищем описание приоритета/операции, а находим описание операции/приоритета с нужным нам обозначением, то поиск продолжается в обычном порядке;

— если мы ищем описание унарной/бинарной операции, а находим описание бинарной/унарной операции с нужным нам обозначением, то поиск продолжается в обычном порядке;

— если мы ищем описание унарной/бинарной операции и находим описание O унарной/бинарной операции с нужным нам обозначением, но в формуле исходный вид операнда/операндов не приводим в позиции $k4$ к виду операнда/операндов в описании O , то в случае крепкой связанности исходного вида операнда/операндов с видом операнда/операндов из описания O в программе ошибка (см. «специальное ограничение» выше), а иначе поиск продолжается обычным образом.

6. Уравнение видов. При компиляции программ с АЛГОЛа 68 возникает проблема обработки конструкторов с уравнением видов. Различные способы окончания конструктора могут иметь различные виды m_1, \dots, m_h (например, окончания предложения

```
[1 : n] real x, int y; real z = 5; if n > 0 then goto l1 fi; x exit  
l1 : if n < 10 then goto l2 fi; y exit l2 : z могут иметь виды  
ref [ ] real, ref int или real). Правила АЛГОЛа 68 требуют,  
чтобы один из видов  $m_i$  («главный» вид) приводился в той позиции,  
в которой стоит конструктор, а остальные виды приводились бы в позиции  $k5$ .
```

Метод Боома [70] позволяет по набору видов определить, какой из них главный. Прежде всего отбрасываются те способы окончания конструктора, которые всегда стоят в позиции $k5$ (это циклы, совместные предложения, переходы, пропуски и псевдоимена). Если все способы окончания будут отброшены, то весь конструктор должен стоять в позиции $k5$ и никакого уравнения

нет (вид результата в позиции $k5$ определяется объемлющим конструктом).

Пусть, однако, виды m_1, \dots, m_n не отброшены. Далее используется специально подобранная функция h , определенная на видах АЛГОЛа 68 и имеющая значения в множестве упорядоченных пар целых чисел.

На множестве упорядоченных пар целых чисел (a, b) определены сложение и лексиграфический порядок:

$$(a, b) + (c, d) = (a+c, b+d)$$

$$(a, b) < (c, d), \text{ если } a < c \text{ или } a = c \text{ и } b < d.$$

Пара $(a, 0)$ отождествляется с a .

Функция h устроена так, что вид (быть может не единственный) с наибольшим значением h (среди видов m_1, \dots, m_n) всегда можно выбрать в качестве главного.

Функции h задается набором правил, причем на каждом шаге должно применяться первое из применимых правил:

1) $h(\text{real}) = 3$

2) $h(\text{compl}) = 6$

3) $h(\text{row } m) = h(m) + 3$

4) $h(\text{union}(m_1, \dots, m_k)) = 3 \times k + 1 \leq i \leq k (h(m_i))$ в предположении, что все m_i различны и не начинаются на **union**

5) $h(\text{ref stowed}) = h(\text{stowed}) - 1$

6) $h(\text{ref } m) = h(m) - (0, 1)$

7) $h(\text{proc } m) = h(m)$

8) $h(m) = 0$.

Здесь подразумевается, что в правилах 1 и 2 **real** и **compl** обозначают виды, эквивалентные видам **real** и **compl** с любым числом приставок, **row** обозначает размерность (например, $h([\]m) = h(m) + 3$ и $h([\ ,]m) = h([\]m) + 3$), m и m_i — это любые виды, **stowed** это виды, начинающиеся на **struct** или на «[», или на **flex**.

Метод Боома основывается на следующем утверждении: Если вид **a** приводится к виду **c** в позиции $k5$, но не приводится в позиции ki , $i < 5$, а вид **b** приводится к виду **c** в позиции ki , то $h(a) < h(b)$.

Доказательство проводится перебором позиций и правил.

Заметим, что правило 4 нельзя заменить на

$$h(\text{union}(m_1, \dots, m_k)) = 3 + \max(h(m_i)),$$

так как тогда дообъединение не увеличивало бы функцию h .

Таким образом, если $h(m_i) \geq h(m_j)$ для всех j , то вид m_i можно выбрать главным.

Боом определил функцию h , по-видимому, путем многочисленных эмпирических попыток, заранее надеясь на ее существование. См. также [2].

§ 19. КОНТРОЛЬ ОБЛАСТИ ДЕЙСТВИЯ

Для определения области действия используется вспомогательное понятие доступная вершина. Доступной вершиной называется всякая вершина дерева вычислений, образованная последним исполнением конструктора (текста процедуры или закрытого предложения), объемлющего исполняемый конструктор. Доступные вершины упорядочены согласно вложенности соответствующих им конструкторов.

Областью действия имени, создаваемого локальным генератором или локальным описанием переменной, является самая старшая из доступных вершин, которая соответствует либо тексту процедуры, либо закрытому предложению, содержащему описание, отличное от описаний приоритета, счетчика цикла, спецификации и метки.

Областью действия процедуры (формата), вырабатываемого текстом процедуры (текстом формата), является самая старшая из доступных вершин, которая соответствует конструктору, содержащему некоторое описание, используемое в тексте процедуры (тексте формата), но индикаторы вида, используемые как формальные или виртуальные описатели, не учитываются.

При исполнении ряда конструкторов (§ 7) необходимо осуществлять контроль области действия. Чтобы уменьшить количество динамических проверок, можно использовать алгоритм статического контроля области действия [174]. Алгоритм основан на вычислении интервала области действия (содержащего в порядке старшинства все вершины дерева вычислений между двумя доступными вершинами) для каждого конструктора. В ряде случаев сравнение интервалов области действия позволяет отменить динамическую проверку, либо выявить ошибку.

Стремление упростить динамический контроль области действия приводит к распределению памяти с использованием «стека стеков» и кучи. Такое распределение памяти используется в большинстве реализаций. Однако в [28] предложено другое распределение памяти (сохраняющее все ячейки, на которые остались ссылки), при котором контроль области действия не нужен, но задача резервирует памяти больше, чем ей реально требуется.

§ 20. РАЗВИТИЕ АЛГОЛОПОДОБНЫХ ЯЗЫКОВ

Разработка. Язык программирования АЛГОЛ 60 [184] отражает опыт программирования в части управления порядком вычислений, имевшийся к 1960 году. Он более приемлем для математика, чем, например, ФОРТРАН, но в АЛГОЛе 60 не хватает многих практически важных средств (нет доступа к ячейке ЭВМ, нет средств работы с разнородной информацией, нет развитого обмена).

Впоследствии АЛГОЛ 60 был дополнен средствами обмена, кроме того появились интересные расширения АЛГОЛа 60: АЛГОЛ W, ПАСКАЛЬ [12], СИМУЛА 67 [15], КО-АЛГОЛ [205] и другие. Подъязык АЛГОЛа 60, названный АЛГАМС, принят в СССР в качестве стандарта [45].

АЛГОЛ 60 оказал чрезвычайно большое влияние на развитие методов программирования.

Рабочая группа 2.1 ИФИП с 1963 года обсуждала развитие АЛГОЛа X, преемника АЛГОЛа 60 [106, 112, 124, 128, 185, 215, 237]. Предварительные варианты языка [238] использовались в различных курсах лекций, и опыт, полученный при изложении языка компетентной аудитории, повлиял на последующие варианты. В 1968 году вариант языка под названием АЛГОЛ 68 был одобрен для публикации ИФИП и впоследствии переведен на русский [1], немецкий [67], французский [92] и болгарский [3] языки.

Первоначальный вариант АЛГОЛа 68 вызвал резкую критику ряда членов рабочей группы, отраженную в «сообщении меньшинства» [96]. Д. Росс пытался найти компромиссную точку зрения [208], но в итоге пришел к выводу, что сообщение об АЛГОЛе 68 не читабельно, и что одобрение ИФИП явилось неоправданным противовесом возражениям меньшинства.

Одновременно с «сообщением меньшинства» появилось сообщение об АЛГОЛе N [138], упрощенной модификации АЛГОЛа 68, получившей распространение в Японии. Авторы АЛГОЛа N также назвали сообщение об АЛГОЛе 68 трудным для изучения.

Однако многие организации (см. § 17) предприняли попытку реализовать АЛГОЛ 68. Под влиянием опыта реализации, особенно [91, 244, 245] и [80, 81], и критики, особенно письма [229], было решено внести изменения в язык. Из рассматривавшихся предложений [76, 78, 104, 105, 160, 167, 181, 189, 196, 201—204, 235] были выбраны лишь те, которые улучшали язык, а не изменяли его. Пересмотренный вариант по-прежнему сохранил название АЛГОЛа 68. В нем с помощью грамматик ван Вейнгаардена описывается не только синтаксис, но и статическая семантика. Изложение языка стало более доступным, но для первого чтения по-прежнему следует рекомендовать неформальное описание [155, 188, 220].

Разработаны национальные варианты АЛГОЛа 68, в том числе русский, французский и арабский.

Сравнение АЛГОЛа 68 с другими языками программирования см. в работах [23, 53, 226, 232].

Влияние реализаций. Из многочисленных реализаций АЛГОЛа 60 особое значение имеет Киргроувудский компилятор для КДФ9, описанный в [37]. Этот сравнительно простой компилятор создает рабочую программу, состоящую из последовательности вызовов специальных подпрограмм. Каждой конструк-

ции, входящей в программу на АЛГОЛе 60, соответствует вызов определенной подпрограммы, которая получает в качестве параметров результат исполнения других подпрограмм, соответствующих вложенным конструкциям. Такой полуинтерпретационный компилятор не очень эффективен, но удобен при отладке программ и сравнительно легко переносим.

Специалисты фирмы Бэрроуз решили повысить эффективность вышеописанной реализации АЛГОЛа 60. Созданная ими машина В5500 (и затем В6500 и 6700) одной командой осуществляет почти каждую подпрограмму, содержащуюся в рабочей программе, откомпилированной с АЛГОЛа 60 согласно [37]. При аппаратной реализации многие составляющие подпрограммы выполняются одновременно, что делает реализацию АЛГОЛ 60 для В5500 весьма эффективной во всех отношениях (быстрота трансляции, эффективность рабочей программы и самое главное — короткий код).

При разработке В5500 выяснилось, что много новых полезных конструкций целесообразно включить в АЛГОЛ 60, чтобы использовать все возможности разрабатываемой аппаратуры. Таким образом появился расширенный вариант АЛГОЛа 60.

АЛГОЛ 68 разработан как новый язык, а не как расширение АЛГОЛа 60. Он отражает накопленный к 1970 году опыт программирования в части управления порядком действий, введения видов значений и организаций обмена. Большинство идей АЛГОЛа 68 (не включая такие важные, как контроль видов, контроль области действия и обмен, управляемый видами значений) заложены еще в языке ПЛ/1. Семантика пересмотренного варианта АЛГОЛа 68 разработана М. Синцовым под влиянием ЭВМ В6700. Эта семантика определяется в виде действий некоторой идеализированной вычислительной машины (ИВМ). Описание семантики занимает в совокупности 15 страниц (не считая описания ИВМ) и читается очень легко. Описание семантики не подвергалось критике, в отличие от описания других разделов АЛГОЛа 68.

Для АЛГОЛа 68 также можно построить компилятор, создающий рабочую программу в виде последовательности вызовов специальных подпрограмм. Полученные подпрограммы будут зависеть от некоторых видов, поэтому такая рабочая программа окажется слишком неэффективной.

Целесообразно во время компиляции проделать все вычисления, зависящие только от видов, и получить рабочую программу от видов не зависящую.

Аппаратная реализация* видонезависимых подпрограмм полученной рабочей программы (например, подпрограмм для циклов, вырезок, некоторых генераторов и выбирающих предложений) позволила бы весьма эффективно использовать

* Аппаратная реализация подмножества АЛГОЛа 68 предпринималась, но авторы не располагают точной информацией по этому вопросу.

АЛГОЛ 68, а также выявить новые полезные языковые средства.

Подъязыки. АЛГОЛ 68 задуман и удачно спроектирован для весьма широкого круга возможных применений. Однако для некоторых классов применений или особых групп пользователей АЛГОЛ 68 может оказаться избыточным. В этих случаях целесообразно использовать подъязыки АЛГОЛа 68: либо более легкие в изучении, либо более простые для компиляции, либо не требующие сложной динамической поддержки (для использования в управляющих системах). Выделение подъязыков производится устранением ряда конструкций языка, с возможным введением проблемно-ориентированного библиотечного вступления. Стандартизация подъязыков развита в настоящее время недостаточно (см. [110, 119, 120, 200]). Следует ожидать продолжение работ по стандартизации подъязыков АЛГОЛа 68.

АЛГОЛ 68 явился примером для подражания в построении некоторых систем программирования. Особенно следует отметить формирование языков для решения специальных классов задач. Основные понятия АЛГОЛа 68 оказали влияние на разработку языка АНАЛИЗ [25], предназначенного для записи программ аналитических выкладок на ЭВМ. В языке АНАЛИЗ использованы понятия видов значений, описания операций и приоритетов, а также обобщение понятия формулы.

АЛГОЛ 68 с успехом применяется для построения языков программирования, близких по эффективности к автокодам.

В настоящее время необходимость для каждой конкретной ЭВМ базового языка программирования, отличного от кода машины, не вызывает сомнений. Уровень изобразительных средств и полнота представления данных и операций базового языка определяет стиль и эффективность последующего программирования. Получение программ с хорошими характеристиками не может быть обеспечено в различных вычислительных системах фиксированным языком. Необходимо разрешить противоречие между машинной зависимостью языка и требованиями языков высокого уровня. Традиционные автокоды «один в один» малоэффективны, поскольку в них отсутствуют рекурсивные структуры. Для разрешения этого противоречия в работах [26, 29] используется следующий подход. АЛГОЛ 68 проецируется на архитектуру данной ЭВМ посредством исключения из него тех видов значений и конструкций, которые не поддерживаны аппаратно. В стандартном вступлении остаются только те операции, которые имеются (или просто реализуются) в архитектуре ЭВМ. В результате такой проекции получается подмножество языка АЛГОЛ 68, которое будем называть формульный автокод (ФРАК) [26].

По существу ФРАК — язык высокого уровня, представляющий архитектуру ЭВМ для пользователей или системных программистов. В языках типа ФРАК выделяются два уровня —

ядро и оболочка. Ядро определяется при проекции языка АЛГОЛ 68 на архитектуру ЭВМ. Оболочка определяет:

— дополнительные виды значений и операций над ними, учитывающие проблемную ориентацию применения ЭВМ,

— единицы действия, являющиеся сокращениями часто встречающихся композиций операций,

— средства для изображения операций, свойственных только архитектуре данной ЭВМ (вносятся в библиотечное вступление),

— дополнительные сервисные возможности в виде прагматов для улучшения отладочных свойств системы, построенной на базе языка ФРАК.

Языки типа ФРАК разработаны и реализованы для нескольких специализированных ЭВМ.

Эффективность проекции оценивается такими фактами:

— на языке ФРАК можно представить все операции ЭВМ,

— каждый ограничитель транслируется только в одну операцию ЭВМ,

— транслирующая система имеет всего два просмотра,

— уровень программирования соответствует уровню языка АЛГОЛ 68,

— отпадает необходимость глобальной оптимизации,

— язык ФРАК прост для обучения пользователя, впервые столкнувшегося с программированием, поскольку его изобразительные средства сами диктуют применения операций для записи алгоритмов.

Надъязыки. Среди возможных направлений расширения АЛГОЛа 68 отметим два: введение модулей и введение средств обработки прерываний.

Важным инструментом программирования является использование модулей. Хотя АЛГОЛ 68 и допускает реализацию (см. § 18), создающую приемлемую для организации модулей рабочую программу, средства для организации независимой компиляции процедур на АЛГОЛе 68 находятся в стадии обсуждения [69, 73, 74].

Возникающая при этом проблема введения переменного вида (мода) также не решена [164]. ИФИП предполагает стандартизовать средства введения модулей в АЛГОЛ 68.

Во многих реализациях будет введена развитая система прерываний. Маскирование прерываний и задание реакций на прерывания можно осуществлять через процедуры библиотечного вступления, аналогичные процедурам присоединения (§ 9) в обмене. Однако для обработки внешних прерываний необходимо введение дополнительных конструкторов. Например, для обработки прерываний от таймера можно использовать временной цикл **timer** время **from** начало **by** фаза **to** конец **while** условие **do** тело **od**, где идентификатор «время» обладает астрономическим временем, прошедшим после «начало»; «тело» цикла исполняется после момента «начало» через интервал времени «фаза», если

выполняется «условие»; если «тело» не выполнено за интервал времени «фаза», то происходит прерывание.

Перспективы внедрения АЛГОЛа 68. К настоящему моменту разработано около 3000 языков программирования. Все более актуальной становится проблема замены этого несистематизированного и необозримого множества языков на один универсальный язык программирования (или на небольшое число языков). АЛГОЛ 68 на наш взгляд может стать первым шагом в решении этой проблемы, по той причине, что он в большей степени, чем другие распространенные языки, отвечает современным потребностям программирования.

Разработка нового языка программирования займет не менее 10—15 лет, поэтому эксперты, по-видимому, будут рекомендовать проект внедрения АЛГОЛа 68 в качестве унифицированного языка программирования с широкой областью применения.

Этот проект должен включать, во всяком случае, следующие этапы:

1) стандартизация машинного представления программ на национальных вариантах и разработка компиляторов с АЛГОЛа 68 для различных ЭВМ; разработка системы тестов для аттестации компиляторов; исследование методов переносимости программ; расширение преподавания и создание учебно-методических материалов по АЛГОЛу 68;

2) опытная эксплуатация компиляторов; разработка методов переносимости программ; разработка приемов программирования и создание библиотек алгоритмов на АЛГОЛе 68; развитие надъязыков и подъязыков;

3) Стандартизация языка; практическое использование компиляторов и создание систем программирования на базе АЛГОЛа 68;

4) расширение сферы использования АЛГОЛа 68; стандартизация подъязыков и надъязыков; создание пакетов прикладных программ.

Уже этот неполный перечень показывает необходимость координации работ по внедрению. Однако значительные усилия, необходимые для внедрения языка, вполне оправдываются повышением эффективности работы программистов при использовании АЛГОЛа 68, его удобством для обучения программированию и возможностями обмена программами на нем между пользователями разнотипных ЭВМ.

Авторы признательны В. Л. Лейтесу, В. Б. Яковлеву, В. М. Гушину, В. М. Курочкину, Н. Н. Непейводе, М. Р. Левинсону и Э. А. Аслаяну за обсуждение различных аспектов данного обзора.

Обзор не является официальной публикацией упомянутой выше Комиссии по АЛГОЛу 68, но его написание в значительной степени стимулировалось дискуссиями на заседаниях Комиссии.

БИБЛИОГРАФИЯ

1. Алгоритмический язык АЛГОЛ-68. Ред. Ершов А. П., Кибернетика, 1969, № 6, 17-144 и 1970, № 1, 12-60 (РЖМат, 1970, 9В485 и 1971, 1В580)
2. АЛГОЛ 68. Методы реализации. Ред. Цейтлин Г. С., Л., ЛГУ, 1976
3. Алгоритмичният език АЛГОЛ 68. Перевод на български език. Д. Тошкова и Ст. Бъчварова, «Наука и изкуство», София, 1971
4. Арнаудов Д. Д., О некоторых характерных чертах АЛГОЛ-68 и его связи с АЛГОЛ-60. В сб. «Цифр. вычисл. техника и программир.». Вып. 5. М., «Сов. радио», 1969, 119-122 (РЖМат, 1970, 5В530)
5. Балухев А. Н., Братчиков И. Л., Некоторые особенности промежуточного языка транслятора с АЛГОЛа-68. В сб. «Теория языков и методы построения систем программир.» Киев-Алушта, 1972, 391-396 (РЖМат, 1974, 11В553)
6. Бауэр Ф. Л., Гооз Г., Информатика. Вводный курс. М., «Мир», 1976. 488 стр. (РЖМат, 1977, 3В745К)
7. Бранкар Р., Кардинал Дж. Р., Леви Дж., Делексаи Дж. Р., Бегин М. ван., Простой транслирующий автомат, позволяющий генерировать оптимальный код. Тр. Всес. симпоз. по методам реализации новых алгоритм. языков. ч. 2. Новосибирск, 1975, 38-49
8. Броль В. В., Упорядочивание множеств деревьев. Программирование, 1976, № 3, 40-47 (РЖМат, 1976, 10В749)
9. Бублик В. В., Об одном подходе к построению синтаксического процессора. В сб.: «Языки систем. программир. и методы их реализации». Киев, 1974, 164-171 (РЖМат, 1975, 6В940)
10. Бублик В. В., Обобщенная контекстно-свободная грамматика алгоритмического языка АЛГОЛ 68. Кибернетика, 1975, № 6, 7-14 (РЖМат, 1976, 6В84)
11. Васильев В. А., Язык АЛГОЛ-68. Основные понятия. М., «Наука», 1972, 128 стр. (РЖМат, 1973, 1В800К)
12. Вирт Н., Язык программирования ПАСКАЛЬ. В сб. «Алгоритмы и организов. решения экон. задач», вып. 3., М., Статистика, 1974, 38-67 (РЖМат, 1974, 12В640)
13. Грушецкий В. В., Декомпозиция входных программ в многоязыковом трансляторе. Тр. Всес. симпоз. по методам реализ. новых алгоритмич. языков. ч. 1. Новосибирск, 1975, 82-913
14. Дал У., Дейкстра Э., Хоор К., Структурное программирование. М., «Мир», 1975, 248 с. (РЖМат, 1976, 3В904К)
15. —, Мюрхауг Б., Ньюгорт К., СИМУЛА 67. Универсальный язык программирования. М., «Мир», 1969, 99 стр. (РЖМат, 1970, 3В578К)
16. Дейкстра Э., Взаимодействие последовательных процессов. В сб. «Языки программирования», М., «Мир», 1972
17. Деканосидзе Е. Н., Использование алгоритмического языка АЛГОЛ-68 для описания экономических задач. Сакартвелос ССР Мецниеребата Академис Гамотвлиги центрис Шромеби, Тр. Вычисл. центра АН Груз. ССР, 1970, 10, № 1, 5-15 (РЖМат, 1971, 4В637)
18. Ершов А. П., Система БЕТА-сравнение постановки задачи с пробной реализацией. Тр. Всес. симпоз. по методам реализ. новых алгоритмич. языков. Ч. 1, Новосибирск, 1975, 73-81 (РЖМат, 1976, 11В910)
19. Йонеда Нобуо, Новый алгоритмический язык АЛГОЛ 68. I. «Сури кагаку, Math. Sci.», 1970, 8, № 2, 78-83 (Японск.)
20. —, Новый алгоритмический язык АЛГОЛ-68. II. «Сури кагаку, Math. Sci.», 1970, 8, № 4, 68-72 (Японск.)
21. Касьянов В. Н., Трахтенброт М. Б., Анализ структур программ в глобальной оптимизации. Тр. Всес. симпоз. по методам реализ. новых алгоритмич. языков. Ч. 1. Новосибирск, 1975, 143-160 (РЖМат, 1976, 9В750)
22. Кернер И. О., Одно подмножество языка АЛГОЛ-68 и его реализация.

- Тр. Всес. симпози. по методам реализ. новых алгоритмич. языков. ч. 2. Новосибирск, 1975, 88-93 (РЖМат, 1976, 9В703)
23. *Кикадзе Н. Н.*, Сопоставление конструкций языка ЛИСП с конструкциями языка АЛГОЛ-68. Тр. Вычисл. центра АН СССР, 1975, № 2, 13-14, 143-155 (РЖМат, 1975, 12В919)
 24. *Королев Л. Н.*, Структуры ЭВМ и их математическое обеспечение., М., «Наука», 1974, 254 с. (РЖМат, 1975, 1В936К)
 25. *Красилов А. А.*, Герасимов Ю. А., Автоматизация аналитических выкладок на ЭВМ. Доклад на XIX науч. конф. Моск. физ.-техн. ин-та, Долгопрудный, 1973
 26. *Красилов А. А.*, Лейтес В. Л., Лоскутов В. Г., Использование свойств языка АЛГОЛ-68 в автокоде. Тр. XIX Науч. конф. Моск. физ.-техн. ин-та, Сер. Аэромех. и процессы упр., Долгопрудный, 1974, 143-146 (РЖМат, 1975, 6В865)
 27. *Левина Е. Л.*, О некоторых алгебраических свойствах операций АЛГОЛа-68. В сб. «Системное и теор. программирование». Новосибирск, 1973, 124-142 (РЖМат, 1974, 9В1031)
 28. *Левинсон М. Р.*, Метареализация АЛГОЛа 68. Тр. Всес. симпози. по методам реализ. новых алгоритмич. яз. Ч. 2. Новосибирск, 1975, 119-144 (РЖМат, 1976, 9В759)
 29. *Лейтес В. Л.*, Масленников В. Б., Реализация параллельных вычислений на многопроцессорной ЭВМ. Доклад на XXI Науч. конф. Моск. физ.-техн. ин-та., Москва, 1975
 30. *Линдси Ч.*, ван дер Мюйлен С., Неформальное введение в АЛГОЛ 68, М., «Мир», 1973
 31. *Маслов А. Н.*, Индексные грамматики и грамматики Вейнгаардена. Пробл. передачи информ., 1975, II, № 3, 81-89 (РЖМат, 1976, 1В1395)
 32. —, Использование расширения индексных грамматик для синтаксического анализа. Тр. Всес. симпози. по методам реализ. новых алгоритмич. яз. Ч. 2. Новосибирск, 1975, 202-209 (РЖМат, 1976, 10В828)
 33. *Пенгковский В. П.*, Средства структурного программирования для языка высокого уровня., ИТМ и ВТ, М., 1976
 34. Пересмотренное сообщение об АЛГОЛе 68. Ред. Ершов А. П., М., «Мир», 1977
 35. *Покровский С. Б.*, Семантическая унификация в многоязыковом трансляторе. Тр. Всес. симпози. по методам реализ. алгоритмич. языков. ч. 1. Новосибирск, 1975, 94-112
 36. *Поттоси И. В.*, Глобальная оптимизация; практический подход. Тр. Всес. симпози. по методам реализ. новых алгоритмич. языков. Ч. 1. Новосибирск, 1975, 113-128 (РЖМат, 1976, 9В773)
 37. *Ренделл Б.*, Расселл Л., Реализация АЛГОЛа 60. М., «Мир», 1967
 38. *Терехов А. Н.*, Процессы идентификации и структура компилятора с языка АЛГОЛ-68. Программирование, 1975, № 2, 61-67 (РЖМат, 1975, 12В954)
 39. —, Цейтин Г. С., Средства эффективного синтеза объектной программы. Программирование, 1975, № 6, 38-48 (РЖМат, 1976, 5В1019)
 40. —, —, Язык синтеза объектной программы с учетом последующего контекста. Тр. Всес. симпози. по методам реализ. новых алгоритмич. яз. Ч. 2. Новосибирск, 1975, 227-236 (РЖМат, 1976, 10В744)
 41. *Трахтенброт Б. А.*, Бардзинь Я. М., Конечные автоматы (поведение и синтез), «Наука», 1970, 400 стр. (РЖМат, 1970, 9В330)
 42. *Цейтин Г. С.*, Реализация параллельного исполнения и гибких имен в АЛГОЛе 68. Тр. Всес. симпози. по методам реализ. новых алгоритмич. яз. Ч. 2. Новосибирск, 1975, 74-87 (РЖМат, 1976, 9В765)
 43. *Чернюгов В. В.*, Реализация фрагмента языка АЛГОЛ-68. В сб. «Вопр. систем. программир. ЦЭМИ АН СССР». Вып. 1. М., 1974, 71-91 (РЖМат, 1976, 8В851)
 44. *Шагалов В. В.*, Об однозначности приведений в АЛГОЛе 68. Дипломная работа. УдГУ, 1977

45. Язык программирования АЛГАМС. ГОСТ 21551-76. Издательство стандартов, 1976
46. *Ataylioglu A., Evans G. A., Hyslop J.*, Automatic generation of quadrature formulae for oscillatory integrals. *Comput. J.*, 1975, 18, № 2, 173-176 (ПЖМат, 1976, 2Б1038)
47. —, —, —, The use of Chebyshev series for the evaluation of oscillatory integrals. *Comput. I.*, 1976, 19, № 3, 258-267 (ПЖМат, 1977, 4Б984)
48. *Ammeraal L.*, On the design of programming languages including mini ALGOL 68. «Lect. Notes Comput. Sci.», 1975, 34, 500-504 (ПЖМат, 1976, 5Б1279)
49. —, Mini ALGOL 68 user's guide. «Math. Cent. Afd. Inform.», 1975, IW, № 32, 11, 21 pp. (ПЖМат, 1975, 9Б925)
50. —, An implementation of an ALGOL 68 sublanguage. «Math. Cent. Afd. Inform.», IW, 31/75, Amsterdam, 1975
51. *Anderson G. G., Anderson V. E.*, A model for extensible-contractible language compilers. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 233-252
52. *Apostolatos N., Christ H., Santo H., Wippermann H.*, Rounding control and the algorithmic language ALGOL 68. *ALGOL Bull.*, 1968, № 29, 20—26
53. *Bachmann Karl-Heinz*, Die Programmierungssprachen PASCAL und ALGOL 68. Berlin, Akad.-Verl., 1976. 220 S. (ПЖМат, 1977, 1Б794К)
54. *Baecker H. D.*, The use of ALGOL-68 for trees. *Comput. J.*, 1970, 13, № 1, 25-27 (ПЖМат, 1970, 9Б500)
55. —, ASERC, a code for ALGOL 68 basis tokens. *ALGOL Bull.*, 1968, № 28, 70-75
56. —, Garbage collection for virtual memory computer systems. *Communs ACM*, 1972, 15, № 11, 981-986 (ПЖМат, 1974, 4Б638)
57. —, A reasand record-classes. *Comput. J.*, 1975, 18, № 3, 223-226
58. *Bährs A. A., Ershov A. P., Rar A. F.*, Adaptation of ALGOL 68 to national languages and implementation peculiarities. *ALGOL Bull.*, 1973, № 35, 27-28 (ПЖМат, 1974, 9Б1027)
59. *Baker J. L.*, The syntax of ALGOL 68, property grammars, and context-sensitive languages. The Univ. of Calgary, 1971, 62 p.
60. —, An unintentional omission from ALGOL 68. *Inf. Process. Lett.*, 1972, 1, № 6, 244-245 (ПЖМат, 1973, 10Б642)
61. —, Grammars with structured vocabulary: a model for the ALGOL-68 definition. *Inform. and Contr.*, 1972, 20, № 4, 351-395 (ПЖМат, 1972, 10Б676)
62. *Banatre J. P., Routeau J. P.*, A one pass compiler using events. Proc. 1975 Internat. conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 38-52
63. *Bedö Arpád.*, ALGOL 68 a matematikusok programozási nyelve. *Számológép.*, 1971, 1, № 3, 89-91 (ПЖМат, 1972, 4Б534)
64. *Bélic H.*, An introduction to ALGOL 68. «Annu Rev. Automat. Program.» 1973, 7, № 3, 143-169
65. *Bekkers Y., Herman D., Raynal M., Verjus J. P.*, A standard prelude for a multi user operating system based on ALGOL 68. Proc. 1975 Internat. conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 190-201
66. *Bell D.*, Programmer selection and programming errors. *Comput. J.*, 1976, 19, № 3, 202-206 (ПЖМат, 1977, 2Б880)
67. Bericht über die algorithmische Sprache ALGOL 68. Berlin, Akad.-Verl., 1972. 381 S. (ПЖМат, 1972, 9Б587К)
68. *Birrell A. D.*, Problems in implementing ALGOL 68 C. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 2-12
69. —, *Lindsey C. H.*, Conclusions of task-forse meeting 19 May 1977, Preprint, Quiberon, France, 1977

70. *Boom H.*, Note on balancing in ALGOL 68. ALGOL Bull., 1973, № 36, 17-24 (PJKMar, 1974, 8B801)
71. —, Experience with the use of ALGOL W as SIL. ALGOL Bull., 1973, № 37, 63-67
72. —, Textual management in an ALGOL 68 compiler. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 64-98
73. —, Separate compilation in MC68. Preprint, Math. Cent., Amsterdam, 1977
74. *Boom H.*, Separate compilation, definition modules, and block structured languages, Preprint. «Math. Cent. Afd. Inform.», 1977, IW, 77/77, Amsterdam, 1977
75. *Bourne S. R.*, *Guy M. J. T.*, Suggestion for improvements to ALGOL 68. ALGOL Bull., 1972, № 33, 47-53 (PJKMar, 1974, 2B921)
76. —, —, Comments on suggested improvements to ALGOL 68. ALGOL Bull., 1972, № 33, 54-58 (PJKMar, 1974, 2B922)
77. *Boussard J. C.*, *Pair C.*, Introduction a ALGOL 68. Rev. franc. inform. et rech. oper., 1969, 3, № B-3, 17-52 (PJKMar, 1970, 11B498)
78. *Bowlden H. J.*, ALGOL 68 — comment and recommendations. ALGOL Bull., 1970, № 31, 28-32
79. *Branquart P.*, *Cardinael J. P.*, *Delescaille J. P.*, *Lewi J. A.*, A context-free syntax of ALGOL 68. Inf. Process. Lett., 1972, 1, № 4, 141-148 (PJKMar, 1972, 12B437)
80. —, —, *Lewi J.*, Optimized translation process, application of ALGOL 68. Int. Comput. Symp. 1973, Amsterdam, e. a. 1974, 101-107 (PJKMar, 1975, 6B948)
81. —, —, —, *Delescaille J. P.*, *Vanbegin M.*, An optimized translation process and its application to ALGOL 68, Lect. Notes in Comput. Sci. 1976, 38
82. —, *Lewi J.*, On the implementation of coercions in ALGOL 68. M. B. L. E. Lab de Recherches, 1970, Report R123, 23 pp.
83. —, *Cardinael J. P.*, Analysis of the parenthesis structure of ALGOL 68. M. B. L. E. Lab. de Recherches, Report R130, 1970, 44 pp.
84. —, —, *Sintzoff M.*, *Wodon P. L.*, The composition of semantics in ALGOL 68. Commun. ACM, 1971, 14, № 11, 697-708 (PJKMar, 1973, 6B745)
85. *Broughton C. G.*, *Thompson C. M.*, Aspects of implementation an ALGOL 68 student compiler. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 23-37
86. *Cleveland J. C.*, Redundant specification in programming languages through POUCHES. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 318-327
87. *Cohen J.*, *Trilling L.*, *Wegner P.*, A nucleus of a theorempower described in ALGOL-68. Int. J. Comput. and Inform. Sci., 1974, 3, № 1, 1-31
88. *Cornelius B. J.*, *Kirby G. H.*, Depth of recursion and the Ackermann function. BIT, (Sver), 1975, 15, № 2, 144-150 (PJKMar, 1976, 3B1210)
89. *Cunin P. J.*, *DeLauny M.*, *Simonet M.*, *Voidron J.*, Code generation from a decorated tree. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 170-189
90. *Currie I. F.*, Modular programming in ALGOL 68. ALGOL Bull., 1976, № 39, 13-19 (PJKMar, 1976, 10B735)
91. —, *Bond S. G.*, *Morison J. D.*, ALGOL 68-R, its implementation and use. Inform. Process. 71. Proc. IFIP Congr. 71, Ljubljana, 1971. Vol. 1. Amsterdam-London, 1972, 360-363 (PJKMar, 1974, 10B832)
92. Definition du langage algorithmique ALGOL 68. Paris, Hermann, 1972, VII, 222 p.
93. *Denert E.*, *Ernst G.*, *Wetzell H.*, Graphex 68. Grafical language features in ALGOL 68. Comput. and Graphics. 1975, 1, № 2/3, 195-202
94. *Deussen P.*, A decidability criterion for van Wijngaarden grammar. Acta. Inform., 1975, 5, № 4, 353-375 (PJKMar, 1976, 5B1309)

95. —, Mehlhorn K., van Wijngaarden grammars and the space complexity class EXPACE. Acta Inform., 1977, 8, № 3, 193-199
96. Dijkstra E. W., Duncan F. G., Garwick I. V., Hoare C. A. R., Randell B., Seegmueller G., Turski W. M., Woodger M., «Minority report». ALGOL Bull., 1970, № 31, 7
97. Dürr D., SL3 — eine Systemprogrammiersprache auf ALGOL 68-Basis als Grundsprache für die Prozeßrechner line AEG 80. Angew. Inform., 1975, 17, № 9, 393—399 (PЖMar, 1976, 3B922)
98. —, Elchentopf S., Prahl U., Siegel G., Tebling G. SL3-eine maschinenorientierte Programmiersprache auf ALGOL 68-Basis. Lect. Notes Comput. Sci., 1974, 12, 517-527 (PЖMar, 1975, 3B832)
99. Earnshaw R. A., Graph plotting in ALGOL 68-R., Software-Practice and Experience, 1975, 6, № 1, 51-60
100. Feldmann H., An interpretation for making references (in ALGOL 68). ALGOL Bull., 1974, № 38, 45-51 (PЖMar, 1975, 7B808)
101. Feyock S., Toward an implementation of the vienna definition language Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 370-384
102. Final report on improvements to ALGOL 68. ALGOL Bull., 1973, № 36, 8-16
103. Fith J., Norman A., Double by single length division. Sigsam Bull., 1975, 9, № 2, 7
104. Freeman W., Suggestions regarding certain representations in ALGOL 68. ALGOL Bull., 1972, № 34, 41-44 (PЖMar, 1974, 10B824)
105. Furter report on improvements to ALGOL 68. ALGOL Bull., 1973, № 35, 5-13 (PЖMar, 1974, 9B1024)
106. Garwick J. V., Merner J. M., Ingerman P. Z., Paul M., Report of the ALGOL-X-I-O. 1966, Subcommittee, WG 2.1 working paper.
107. Gennart P. E., Implementation and usage of the ALGOL 68/19 compiler. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 13-15
108. Gils A. J. M. van, The philosophy of an extensible language. ALGOL Bull., 1972, № 34, 82-88 (PЖMar, 1974, 10B825)
109. Golde H., Parentheses and collateral phrases. Univ. of Washington, 1971, 7 pp.
110. —, Kerner I. O., Meulen S. van der, Sublanguages. Report WG 2.1 Subcommittee. ALGOL Bull., 1972, № 33, 43-45
111. Goos G., Einige Eigenschaften von ALGOL 68. Elektron. Datenverarb., 1969, 11, № 9, 437-442 (PЖMar, 1970, 4B604)
112. —, Scheidig H., Seegmueller G., Walther H., Another proposal for ALGOL 67. 1967, Bavarian Acad. Sci., Munich
113. Grune D., Generating parsers for affix grammar. Commun. ACM., 1972, 15, 728-738
114. —, The MC ALGOL 68 test set. Math. Cent. Dep. Comput. Sci IW (formerly: «Math. Cent. Afd. Inform. IW») 1975, № 53, 143 pp. (PЖMar, 1976, 8B904)
115. Guernic P., Trilling L., Extended serial clauses and their implementation. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 156-169
116. Haentjens R., Proposal for a simpler syntax for the ALGOL 68 unit. ALGOL Bull., 1976, № 40, 21-23
117. Hansen W. J., Two acts in search of the «Standard hardware representation for revised ALGOL 68». Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 328-351
118. —, Boom H., The report on the standard hardware representation for ALGOL 68, SIGPLAN Notices, 1977, 12, № 5, 80-87 (а также ALGOL Bull., 1976, № 40, 24-43)
119. Hibbard P. G., A minimum general purpose sublanguage of ALGOL 68. ALGOL Bull., 1973, № 35, 14-23 (PЖMar, 1974, 9B1025)

120. —, A proposed sublanguage of ALGOL 68. ALGOL Bull. 1973, № 37, 30-53 (PЖMar, 1975, 5B904)
121. —, ALGOL 68 for a minicomputer. Minicomput. Forum. Conf. Proc., 1975. Uxbridge, s. a., 97-107 (PЖMar, 1977, 3B951)
122. *Higman B.*, The place of own variables in programming language theory. Comput. J., 1976, 19, № 3, 225-228 (PЖMar, 1977, 3B796)
123. *Hilden J.*, Integral division once more. ALGOL Bull., № 40, 8-9
124. *Hill I. D.*, Some remarks on the draft report. ALGOL Bull., 1968, № 28, 65-69
125. *Hill U.*, Automatische rekursive Adressenberechnung für höhere Programmiersprachen, Insbesondere für ALGOL 68. Diss., Dokt. Naturwiss. Fak. Allgem. Wiss. Techn. Hochschule München, 1969, 111, 109S., (PЖMar, 1970, 4B624)
126. *Hoare C. A. R.*, Record Handling. 1965 Programm Languages. London-New York, Acad. Pres, 1968, 291-347 (PЖMar, 1969, 8B378)
127. —, Set manipulation. ALGOL Bull., 1968, № 27, 29-37
128. —, Critique of MR 93. ALGOL Bull., 1968, № 29, 27-29
129. —, Initialisation of variables. ALGOL Bull., 1968, № 29, 30-32
130. —, Subscript optimisation and checking. ALGOL Bull., 1968, № 29, 33-44
131. —, Text processing. ALGOL Bull., 1968, № 29, 45-60
132. —, Sets again. ALGOL Bull., 1969, № 30, 4
133. *Hodges Paul*, Intialized generators. ALGOL Bull., 1976, № 40, 6
134. —, The name of the language. ALGOL Bull., 1976, № 40, 7
135. *Housden R. J. W.*, On string concepts and their implementation. Comput. J., 1975, 18, № 2, 150-156 (PЖMar, 1976, 1B1074)
136. *Howenstine R.*, O'Dell M. D., Thompson J. C., Feyock S., Extension features of a macro processor. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 265-292
137. *Igarishi S.*, Comments on the formal treatment of types including Sets. ALGOL Bull., 1968, № 29, 12-15
138. —, *Iwamura T.*, *Sakuma K.*, *Simauti T.*, *Simuzu T.*, *Takasu S.*, *Wada E.*, *Yoneda N.*, ALGOL N. ALGOL Bull., 1969, № 30, 38-85
139. *Jorrand Ph.*, *Paul M.*, *Poel W. L. vd.*, *Rohlfing H.*, *Samelson K.*, *Schuman S.*, *Suzuki N.*, Operating systems and conversational languages, Report WG 2.1 subcommittee. ALGOL Bull., 1972, № 33, 46 pp
140. *Kennedi K. W.*, The ALGOL 68 batch editor. Houston, 1973
141. *Kerangueven A.*, Implantation et gestion sur le calculateur C II 10070 des structures necessaires a l'execution d'un systeme base sur ALGOL 68. Rev. franc. automat., inform., rech. oper., 1974, B8, № 2, 7-18 (PЖMar, 1975, 5B929)
142. *Kerner I. O.*, Al-Sheikh-Khalil Hassan. Algorithmische Sprache ALGOL 60+8. Elektron. Informationsverarb. und Kybern., 1974, 10, № 2-3, 71-107 (PЖMar, 1975, 3B835)
143. —, The two-level grammars for the definition of semantics. Electron. Informationsverarb. und Kybern., 1975, 11, № 4-6, 252-257 (PЖMar, 1976, 2B1083)
144. —, *Lorenzen H.-P.*, Zur Frage der Fachspachen in der Programmierung. Rechentechn. Datenverarb., 1975, 12, № 9, 26-30 (PЖMar, 1976, 6B846)
145. *King P. R.*, W. G. 2.1. subcommittee on ALGOL 68 support. ALGOL Bull., 1973, № 37, 4-11 (PЖMar, 1975, 3B1020)
146. *Koch W.*, *Oeters C.*, An abstract ALGOL 68 machine and its application in a machine independent compiler. Lect. Notes Comput. Sci. 1975, 34, 642-653 (PЖMar, 1976, 4B949)
147. *Koster C. H. A.*, On infinite modes. ALGOL Bull., 1969, № 30, 86-89
148. —, Affix grammar. Proc. IFIP Work. conf. on ALGOL 68 Implementation, 1970

149. —, Two-level grammars. Adv. Course Compil. Constr. Lect. Not. Techn. Univ. Munich, 1974. S. 1., s. a., Ch. 2F, 1-10 (PЖMat, 1976, 3B1207)
150. —, CDL-A compiler implementation language. Тр. Всес. симпоз. по методам реализ. новых алгоритм. языков. ч. 2. Новосибирск, 1975, 237-247
151. —, The mode-system in ALGOL 68. New Dir. Algorithmic Lang., 1975. Roequencourt-Le Chesnay, s. a., 99-114 (PЖMat, 1977, 3B944)
152. *Kral J.*, The equivalence of modes and the equivalence of finite automata. ALGOL Bull., 1973, № 35, 34-35 (PЖMat, 1974, 10B828)
153. —, *Moudry J.*, *Nadrchal I.*, Syntax parser for an ALGOL 68 compiler based on a semi-top-down method of syntactical analysis. Inform. Proces. Machin. 1975, № 19, 87-104 (PЖMat, 1977, 1B828)
154. *Laski J. G.*, Sets and other types. ALGOL Bull., 1968, № 27, 41-48
155. *Learner A.*, *Powell A. I.*, An introduction to ALGOL 68 through problems. The Macmillan Press Ltd., 1974.
156. *Leeuwen I. van.*, Recursively enumerable languages and van Wijngaarden grammars. «Proc. Kon. ned. akad. wetensch. A-Indag. math.», 1977, 80, № 1, 29-30
157. *Leroy A.*, *Moonner E.*, *Picard F.*, *Ployette F.*, On the adequacy of the ALGOL 68 environment compared with an existing current operating system and problems of I/O implementation Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 202-219
158. *Levinson M. R.*, Simulation with ALGOL 68. ALGOL Bull., 1974, № 38, 43-44 (PЖMat, 1975, 7B767)
159. *Lewis P. M. II.*, *Stearns R. E.*, Syntax-directed transduction. J. Assoc. Comput. Mach. 1968, 15, № 3, 465-488 (PЖMat, 1969, 3B409)
160. *Lindsey C. H.*, Proposals for amendment of ALGOL 68. ALGOL Bull., 1968, № 28, 50-57
161. —, An ISO-Code representation for ALGOL 68. ALGOL Bull., 1970, № 31, 37-60
162. —, ALGOL 68 with fewer tears. Comput. J., 1972, 15, № 2, 176-188; а также ALGOL Bull., 1968, № 28, 9-49
163. —, Partial parametrization. ALGOL Bull., 1973, № 37, 24-26 (PЖMat, 1975, 3B1022)
164. —, Modals. ALGOL Bull., 1973, № 37, 26-29 (PЖMat, 1975, 3B1023)
165. —, Specification of partial parametrization proposal. ALGOL Bull., 1976, № 39, 6-9
166. —, Proposal for a modules facility in ALGOL 68. ALGOL Bull., 1976, № 39, 20-29 (PЖMat, 1976, 9B701)
167. —, *Koster C. H. A.*, *Fox A. J.*, Data processing and transput. Report WG 2.1 subcommittee. ALGOL Bull., 1972, № 33, 26-42
168. —, —, Maintenance of, and improvements to, ALGOL 68. Report WG 2.1 subcommittee. ALGOL Bull., 1972, № 33, 21-25
169. *Loeper H.*, *Horn M.*, *Nyderle W.*, Investigations on the application of a precedence-controlled syntactic analysis method to a sublanguage of ALGOL 68. Electron. Informationsverarb. und Kybern., 1975, 11, № 4-6, 385-392 (PЖMat, 1976, 5B1039)
170. *Matuszyński L.*, *Stepowska I.*, Propozycja terminologii polskiej dla ALGOLU 68. Pr. CO PAN, 1972, № 89, 24S. (PЖMat, 1973, 12B694)
171. *Meek B. L.*, Comments on MR93. ALGOL Bull., 1969, № 30, 5-10
172. —, A proposal for a «halfway house» ALGOL. Comput. Bull., 1972, 16, № 11, 537-539
173. *Meertens L. G. L. T.*, On the generation of ALGOL 68 programs involving infinite modes. ALGOL Bull., 1969, № 30, 90-92
174. —, On static scope checking in ALGOL 68. ALGOL Bull., 1973, № 35, 45-58 (PЖMat, 1974, 9B1029)
175. —, A note on integral Division. ALGOL Bull., 1976, № 39, 30-32 (PЖMat, 1976, 10B738)
176. —, A space-saving technique for assigning ALGOL 68 multiple values. Inform. Process. Lett. 1976, 5, № 4, 97-99

177. —, Viet J. C. van. Parsing ALGOL 68 with syntax-directed error recovery. «Math. Cent. Afd. Inform.», IW, 1975, № 54, 37 pp (PЖMar, 1977, 1B959)
178. —, —, Repairing the parenthesis skeleton of ALGOL 68 programs. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 99-117
179. —, —, Parsing ALGOL 68 with syntaxdirected recovery. Proc. 1975, Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 118-155
180. *Morris James H., Jr.*, A bonus from van Wijngaarden's device. Commun. ACM, 1973, 15, № 8, 773 pp (PЖMar, 1974, 2B923)
181. *Moudry J., Král J., Nadrchal J., Socol J.*, Recognition of routine-denotations in ALGOL 68. ALGOL Bull., 1973, № 35, 32-33 (PЖMar, 1974, 10B829)
182. *Mazurkiewicz A. W.*, A note on enumerable grammars. Inform. and Control, 1969, 14, № 6, 555-558 (PЖMar, 1970, 4B600)
183. *Nadrchal J., Král J., Sklenár I., Kriz V., Moudry J.*, Error Recovery and other features of the ALGOL 68 Prague Impementation. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 52-58
184. *Naur P.* Revised report on the algorithmic language ALGOL 60. Comput. J., 5, (1962/63), 349—367, (Русск. перевод: Алгоритмический язык Алгол 60. Пересмотренное сообщение. М. «Мир», 1965)
185. —, Proposal for introduction on aims. 1965, WG 2.1 working paper
186. —, Successes and failures of the ALGOL effort. ALGOL Bull., 1968, № 28, 58-62
187. *Pagan F. G.*, On interpreter-oriented definitions of programming languages. Comput. J., 1976, 19, № 2, 151-155 (PЖMar, 1977, 1B960)
188. —, A practical guide to ALGOL 68. John-Wiley & Sons. Ltd., 1976
189. *Pair C.*, Concerning the syntax of ALGOL 68. ALGOL Bull., 1970, № 31, 16-27
190. *Palmé J.*, «Atoms» as a measurement of program length. ALGOL Bull., 1970, № 31, 33-34
191. —, Part-compilation in high-level languages. BIT (Sver), 1972, 12, № 4, 534-542 (PЖMar, 1973, 10B635)
192. *Peck J. E. L.*, Draft of an ALGOL 68 companion. Univ. of British Columbia, 1971, 83 pp.
193. —, Two-level grammars in action. Inform. Process. 74. Amsterdam-London, 1974, 317-321 (PЖMar, 1975, 8B735)
194. —, *Sintzoff M., Watt J. M.*, ALGOL 68 syntax chart. ALGOL Bull., 1974, № 37, 68
195. *Peol W. L.*, Comment on the composition of semantics in ALGOL 68. Commun. ACM, 1973, 15, № 8, p. 772 (PЖMar, 1974, 1B876)
196. Proposals for revision of the transput sections of the report (Fontainebleau 10). ALGOL Bull., 1972, № 34, 28-40 (PЖMar, 1974, 9B1030)
197. *Rain M.*, Some formal language aspects of MARY. ALGOL Bull., 1972, № 34, 45-81 (PЖMar, 1974, 10B845)
198. —, Declaration groups and values of declarations in MARY. ALGOL Bull., 1973, № 35, 36-37 (PЖMar, 1974, 10B827)
199. —, Operation expressions in MARY. ALGOL Bull., 1973, № 35, 38-44
200. Report on sublanguages. ALGOL Bull., 1972, № 33, 43-46 (PЖMar, 1974, 2B920)
201. Report of the subcommittee on maintenance of and improvements to ALGOL 68. ALGOL Bull. 1972, № 33, 21-25 (PЖMar, 1974, 2B918)
202. Report on the meeting of working group 2.1 held at Fontainebleau from April 7th to 11th 1972. ALGOL Bull. 1972, № 34, 3-5
203. Report on considered improvements (Fontainebleau 9). ALGOL Bull., 1972, № 34, 6-27 (PЖMar, 1974, 10B823)
204. *Rine D. C.*, A proposed multi-valued extension to ALGOL 68. Kybernetes, 1973, 2, № 2, 107-111 (PЖMar, 1973, 12B797)

205. *Roberts J. D.*, CO-ALGOL. ALGOL Bull., 1969, № 30, 17-37
206. *Robertson A.*, *Hendrick G. E.*, A portable compiler for an ALGOL 68 subset. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 59-63
207. *Rosen S.*, Programming systems and languages 1965-1975. Commun. ACM, 1973, 15, № 7, 591-600
208. *Ross D. T.*, Concerning a minority report on ALGOL 68. ALGOL Bull., 1969, № 30, 11-16
209. *Rushworth T.*, *Venema T.*, *Abramson H.*, TOSI. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 293-305
210. *Schneider V. B.*, A translation grammar for ALGOL 68. Int. Comput. Symp. 1970, Bonn. Proc. Frankfurt, 1973, 436-459 (PJKMar, 1977, 2B792)
211. *Scheidig H.*, *Wössner H.*, Überblick über die ALGOL 68-Implementierung an der TU München Lect. Notes Econ. and Math. Syst. 1972, 75, 140-156 (PJKMar, 1973, 6B746)
212. *Schwartz J. T.*, Optimization of very high level language. I. Value transmission and its corollaries. Comput. Lang., 1975, 7, № 2, 161-194 (PJKMar, 1976, 10B835)
213. *Schwartz R.*, A W-grammar description LISP. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 306-317
214. *Scowen R. S.*, Type checking at run time. ALGOL Bull., 1968, № 27, 27-28
215. *Seegmueller G.*, A proposal for a basis for a report on a successor to ALGOL 60. 1965, Bavarian Acad., Sci., Munich
216. *Shearn D. C. S.*, A view on simulation in ALGOL 68. ALGOL Bull., 1974, № 3, 39-42 (PJKMar, 1975, 7B769)
217. —, Discrete event simulation in ALGOL 68. Software — Pract. and exper., 1975, 5, № 3, 279-293 (PJKMar, 1976, 1B1081)
218. *Sintzoff M.*, Existence of a van Wijngaarden syntax for every recursively enumerable set. Ann. Soc. scient. Bruxelles, 1967, Ser. 1, 81, № 2, 115-118 (PJKMar, 1968, 6B512)
219. Introduction a la description de l'ALGOL 68. Rev. franc. inform. et rech. oper., 1969, 3, № B-3, 4-16 (PJKMar, 1970, 11B499)
220. On the revised ALGOL 68 report. ALGOL Bull., 1973, № 36, 28-39 (PJKMar, 1974, 8B803)
221. A brief review of ALGOL 68. ALGOL Bull., 1973, № 37, 54-62 (PJKMar, 1975, 3B1024)
222. *Sklenar I.*, A method of implementation of independently compiled routine texts in ALGOL 68. ALGOL Bull., 1976, № 40, 10-20
223. *Stier J.*, *Akkoyunlu E. A.*, On the terminal objects of ALGOL 68. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 352-369
224. *Stiller G.*, ALGOL 68-Begriffe und Ausdrucksmittel. München-Wien, R. Oldenbourg Verl., 1974, 164S.
225. *Strawn G. O.*, Heap Data, Virtual memory and APL. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 253-264
226. *Tanenbaum A. S.*, A tutorial on ALGOL 68. Comput. Surv., 1976 (PJKMar, 1977, 4B712), 8, № 2, 155-190
227. *Taupin D.*, The ALGOL 68 Compiler of Paris-Sud University. Proc. 1975, Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 16-22
228. *Terrenoire M.*, *Simonet M.*, An evaluation of ALGOL 68 for interrogation process algorithms. Int. J. Comput. and Inform. Sci., 1972, 1, № 1, 67-73 (PJKMar, 1973, 1B895)
229. *Tseytin G. S.*, Letter to P. Branquart, J. Lewi, M. Sintzoff and P. Wodon, November 1968
230. *Turski W. M.*, Some remarks on chapter from a document. ALGOL Bull., 1968, № 29, 9-10

231. *Uzgalis R.*, Page K., Nunes G., Fenchel R., Burgess H., The lessons of Michelle. Proc. 1975 Internat. Conf. on ALGOL 68. Oklahoma State Univ., Stillwater, 1975, 220-232
232. *Valentine S. H.*, Comparative notes on ALGOL 68 and PL/I. Comput. J., 1974, 17, № 4, 325-331 (PЖMar, 1975, 5B906)
233. *Vliet J. C. van*, Towards an implementation — oriented definition of the ALGOL 68 transput, Preprint. «Math. Cent. Afd Inform.», 1977, 1W, 90/77 Amsterdam, 1977
234. *Wegbreit B.*, A generalised compactifying garbage collector. Comput. J., 1972, 15, № 3, 204-208
235. *Wegner E.*, A generalized completer for ALGOL 68. ALGOL Bull., 1973, № 35, 29-31. (PЖMar, 1974, 9B1028).
236. *Wichmann B. A.*, Timing ALGOL statements. Algol Bull., 1968, № 27, 24-26
237. *Wijngaarden A. van*, Orthogonal design and description of a formal language., MR76, 1965, Math. Cent. Amsterdam
238. —, *Mailloux B. J.*, A draft proposal for the algorithmic language ALGOL X., 1966, WG 2.1 working paper
239. —, —, *Peck J. E. I.*, *Koster C. H. A.*, Report on the algorithmic language ALGOL 68. Numer. Math., 1969, 14, № 2, 79-218
240. —, —, —, —, *Sintzoff M.*, *Lindsey C. H.*, *Meertens L. G. L. T.*, *Fisker R. G.*, Revised report on the algorithmic language ALGOL-68. Acta Inform., 1975, 5, № 1-3, 1-236 (PЖMar, 1975, 7B939) (а также приложение к ALGOL Bull., 1974, № 36; а также перепечатка Berlin, Springer, 1976, 236 pp. (PЖMar, 1977, 1B793K))
241. *Wirth N.*, A proposal for a report on a successor of ALGOL 60 1965, Math. Cent. Amsterdam
242. —, Closing word at Zürich colloquium. ALGOL Bull., 1968, № 29, 16-19
243. *Wood D.*, Sets and types. ALGOL Bull., 1968, № 29, 7-8
244. *Woodward P. M.*, Practical experience with ALGOL 68. Software-practice and experience, 1972, 2, № 1, 7-19
245. —, *Bond S. G.*, Users' guide to ALGOL 68R. 1970, 121 pp.
246. *Wössner Hans*, Operatoridentifizierung in ALGOL 68. Diss., Dokt. rer. nat. Fak. Allg. Wiss. Techn. Hochschule München, 1970, III, 76 S. (PЖMar, 1972, 1B990)
247. *Zosel M. E.*, A formal grammar for the representation of modes and its application to ALGOL 68. Diss. Dokt. Phil. Univ. Wash., 1971. Ann. Arbor. Mich Xeros Univ. Microfilms, 1974, 91 pp. (PЖMar, 1975, 9B924K)