цифровой акустической голографии// Акуст. журнал. 1989. т. 35. N 5. с. 784-790.
7. Бадалян В.Г., Базулин Е.Г. Улучшение качества изображения дефектов при восстановлении акустических голограмм// Дефектоскопия, 1987, N 11, с.76-80.
8. Щербинский В.Г., Белый В.Е. Эхо-зеркальный ультразвуковой метод обнаружения и распознавания дефектов сварных швов. - М.: Машиностроение, 1989, 41 с.
9. ГОСТ 23702-90. Контроль неразрушающий. Преобразователи ультразвуковые. Методы испытаний.
10. ГОСТ 26266-84. Контроль неразрушающий. Преобразователи ультразвуковые. Основные параметры и общие технические требования.
11. Бадалян В.Г., Базулин Е.Г., Тихонов Д.С. Влияние поверхности объекта на восстановленное изображение при иммерсионном контроле в акустической голографии// Дефектоскопия, 1989, N 11, с. 51-60.система ультразвукового контроля с когерентной обработкой данных "АВГУР 2.1"// Дефектоскопия, 1993, N 8.

6.8 Язык описания графических объектов АВГУР

6.8.1 Введение

В данном руководстве описана первая версия (1.0) входного языка для представления графических данных о внешних объектах в системе АВГУР. За языком мы закреплено рабочее название АВГУР, что совпадает с названием системы, для которой он предназначен.

Графическая система на базе языка АВГУР работает следующим образом. При первом обращении к графическим данным о внешнем объекте компилятор языка читает входной символьный файл с программой и перерабатывает его в двоичные данные, которые сохраняются в памяти компьютера. После этого, при необходимости появления объекта на фоне растра или трехмерного изображения дефектов запускается интерпретатор теперь уже двоичной графической информации, который строит контурное изображение объекта тем способом, который отвечает текущему контексту работы в системе АВГУР. Структура двоичных данных, вырабатываемая компилятором, такова, что она максимально облегчает последующую работу интерпретатора. Подчеркнем то нетривиальное обстоятельство, что язык АВГУР **не описывает изображения объектов, он описывает сами объекты** в физических координатах абсолютной системы координат. Какие потом получатся изображения зависит от интерпретатора, который принимает данные от компилятора, и от тех условий, в которых в данный момент времени находится интерпретатор.

Основными мотивами, которые способствовали принятию решения о создании языка высокого уровня в качестве основы системы представления графических данных, были следующие:

- Для подготовки файла с графической информацией не требуется ничего кроме любого текстового редактора.
- Достигается одновременно простота и универсальность системы: универсальность --- за счет наличия в языке средств низкого уровня, с помощью которых приложив определенные усилия можно представить любой сложный и нестандартный объект; простота --- за счет наличия библиотеки стандартных объектов, с помощью которой одной командой языка с небольшим числом параметров сразу можно представить нечто достаточно сложное (например, Х-образный сварной шов с валиками усиления).
- Компилятор языка может быть создан в полностью переносимом виде, так что в последующем с платформы системы АВГУР, для которой предназначается язык, его можно будет перенести в любую другую систему, которая, в свою очередь, может функционировать в любой операционной среде: WINDOWS, WINDOWS95, UNIX, dots - важно только, чтобы там воспринимались тексты на языке С++.
- Язык АВГУР разрабатывается таким образом, что файлы библиотек представляют из себя точно такие же символьные файлы на языке АВГУР, как и сами программы,

описывающие объекты (фактически библиотеки является макрорасширениями языка).Таким образом, пользователи своими силами смогут модифицировать, дополнять или создавать новые библиотеки без необходимости изменения системных файлов АВГУРА.

Из сказанного очевидно, что компилятор и интерпретатор языка должны быть интегрированы в систему АВГУР. Однако мы сочли разумным создать простую автономную отладочную программу (отладчик), которая может сильно упростить программирование на АВГУРЕ, особенно при создании библиотек. В состав такой отладочной системы входит собственно компилятор языка и простой графический интерпретатор откомпилированного кода. В настоящее время завершено создание первой версии языка АВГУР и отладчика.

Так как система АВГУР поддерживает двухмерные и трехмерные изображения дефектов, то и язык АВГУР должен обеспечивать описание двухмерных и трехмерных объектов. Такие возможности заложены в язык, но в настоящее время реализована только та их часть, которая относится к двухмерным объектам. Таким образом, в данном руководстве описывается двухмерное подмножество языка АВГУР и соответствующая отладочная система. Помимо самого языка приводится также и техническая информация, необходимая для интегрирования языка АВГУР в любую систему.

6.8.2 Исходные файлы и отладчик

Исполнимый файл отладчика имеет имя **drft.exe** (draft-чертёж). Одна программа на АВГУРЕ занимает ровно один исходный файл (хотя в этом файле могут содержаться инструкции по включению в компиляцию других файлов, например файлов библиотек).

Исходный файл и файлы библиотек могут иметь любое расширение, но по умолчанию предполагается расширение **.dr** для файла программы (draw - рисунок или draft) и **.hdr** для файлов библиотек (от head draw, head draft или просто header, так как файлы библиотек подключаются в начале программы. Вызов отладчика имеет следующий формат:

drft.exe [-l] <имя_файла>

Имя файла программы может содержать и полный путь к файлу. На расположение файлов накладывается следующее ограничение: все включаемые файлы должны находиться в той же директории, что и исходный файл программы. Если в вызове присутствует необязательная опция -l, то во время компиляции создается log-файл: это файл с тем же именем, что и исходный файл, но с расширением **.log**, он содержит в себе все те данные, которые препроцессор (см. ниже) передает на обработку собственно компилятору и, возможно, сообщения об ошибках. log-файл представляет собой тоже правильный файл на языке АВГУР, и он может быть исполнен как и любая другая АВГУР-программа, даже несколько быстрее чем программа, его породившая. Однако этот файл как правило имеет несколько неудобочитаемый вид, так как в нем в явном виде выполнены все включения библиотек, макроподстановки, удалены пробелы и комментарии.

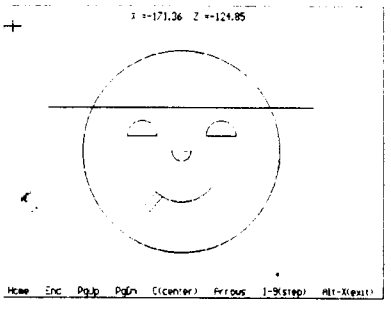

Рис. 6.6 Пример графических образов, созданных с помощью языка АВГУР, отображаемых программой drft.exe.

Если в программе была ошибка, то соответствующее сообщение будет выдано на экран. Если компиляция прошла успешно (и исходная программа содержала исполнимый код, а не только определения макрокоманд), то запускается интерпретатор откомпилированного кода и выводит изображение объекта на экран, в результате чего пользователь может увидеть что-нибудь вроде Рис. 6.6. Помимо изображения объекта на экране имеется курсор в виде креста, с помощью которого можно узнать координаты различных точек объекта. Координаты курсора выводятся в верхней части экрана. Курсор управляется клавишами со

стрелками, шаг передвижения курсора можно менять нажатием клавиш 1-9 на алфавитно-цифровой клавиатуре, другие горячие клавиши указаны в строке подсказок в нижней части экрана.

6.8.3 Язык АВГУР (версия 1.0)

Двухмерное подмножество языка АВГУР позволяет описывать объекты, составленные из отрезков прямых и дуг окружностей, при этом имеется возможность использовать два стиля линий - непрерывные и пунктир и рисовать любым из 16 цветов, присутствующих в стандартной EGA/VGA графике. При входе в программу по умолчанию установлен стиль непрерывной линии и ярко-зеленый цвет (этот цвет не используется АВГУРОМ при создании изображений дефектов).

Программа на языке АВГУР представляет из себя последовательность ряда элементов этого языка, которые мы будем называть **предложениями.** Предложения могут быть либо числовыми выражениями, либо инструкциями препроцессора (имеется всего три типа инструкций препроцессора), либо пустыми. Какова бы ни была природа предложений, все они друг от друга отделяются точкой с запятой ';'. Каждое предложение может занимать сколько угодно строк исходного кода без каких-либо специальных знаков переноса на новую строку; в то же время в одной строке можно поместить любое число предложений (в том числе и ни одного). Длина строки программы не ограничена. Таким образом, ';' является не только разделителем между предложениями, но и единственным возможным разделителем. Сам знак ';' может встретиться несколько раз подряд. Одиночный знак ';' представляет собой пустое предложение. Таким образом, ;;;;;; - правильно написанный фрагмент программы на АВГУРЕ.

Все встречающиеся пробелы и знаки табуляции игнорируются компилятором и не играют в программе никакой роли. Язык АВГУР устроен таким образом, что из программы могут быть удалены все пробельные символы, и смысл программы от этого не изменится. Разрыв строки во всех случаях кроме одного приравнивается к пробелу. Особая роль обрыва строки состоит в том, что он является ограничителем комментария. Комментарий может появиться в любом месте исходного кода, где может появиться пробел. Комментарий начинается с восклицательного знака {\tt '!'} и продолжается до конца строки. Комментарий приравнивается к пробелу, и следовательно игнорируется компилятором. Несколько необычным элементом языка являются символы '[' и ']'. Их можно рассматривать как особый вид пробелов или особый вид комментариев. Во всех случаях они обрабатываются точно так же как пробелы, т.е. игнорируются. Эта конструкция введена в язык для того, чтобы можно было подчеркнуть группирование чего-то не влияя на работу компилятора и не вставляя длинных комментариев. Наиболее типичный случай - группирование координат одной точки. Например, записи **line([10,20],[30,40])** и **line(10,20,30,40)** означают одно и то же, но в первом случае хорошо видно, что линия соединяет две точки. Такая запись особенно удобна, когда аргументов много, или аргументы представляют из себя достаточно сложные выражения, что типично для программ на АВГУРЕ.

6.8.4 Выражения

В принципе содержательная программа на АВГУРЕ может быть написана только с использованием аппарата **выражений**, не прибегая к помощи препроцессора. Выражения представляют собой самый низкий уровень языка АВГУР. Любую исходную программу препроцессор преобразует именно в последовательность выражений, и в таком виде она поступает на обработку собственно компилятора. Результат преобразования исходного кода в последовательность выражений можно найти в log-файле, если при компиляции была установлена опция **-l**. Таким образом, на самом низком уровне программа на АВГУРЕ есть последовательность выражений. Выражения представляют собой обыкновенные числовые выражения, построенные из чисел и **примитивных функций** с использованием знаков арифметических действий + - * / и круглых скобок (). Числа допускаются только

вещественные, в формате с фиксированной точкой (или вовсе без точки). Примитивные функции могут иметь самое разное назначение, но все они принимают аргументы, которые сами могут быть любыми выражениями, и возвращают результат, который является вещественным числом. Арифметические операции и скобки подчиняются обычным правилам приоритета операций. Наряду с бинарными + и - определены также соответствующие унарные операции.

Унарные операции имеют максимальный приоритет (выше чем * и /). Примеры правильно построенных выражений:

```
1.5                                  ! отдельное число - это тоже выражение
1 + 2
1 + ---2                             ! то же самое, что 1 - 2
1.7 + 2.8 * sin(45)
1 + eqz(0, 1 + 2*(3 + asin(0.5)), 1 - 2*(3 + asin(0.5)))
```

Во многих случаях требуется абсолютно точно знать, в каком порядке будут вычисляться отдельные члены выражения. Поэтому сформулируем формальное правило, которое позволяет дать однозначный ответ на этот вопрос. **Выражение** представляет собой последовательность **термов**, соединенных знаками + и -. **Термы**, входящие в выражение, вычисляются в порядке их следования слева направо, по мере их вычисления выполняются операции + и -, тем самым вычисляется значение всего выражения. **Термы** представляют собой последовательность **атомов**, соединенных знаками * и /. **Атомы**, входящие в **терм**, вычисляются слева направо в порядке их следования, по мере их вычисления выполняются операции * и /, тем самым вычисляется значение терма. Вещественные числа, примитивные функции и выражения в круглых скобках есть **атомы**. Конструкции вида **атом** и **+атом** тоже представляют собой атомы.

Компилятор языка АВГУР в действительности есть интерпретатор, работа которого состоит в том, что он одно за другим вычисляет выражения, составляющие программу, а результаты этих вычислений теряет. Эта внешне бессмысленная деятельность приобретает смысл благодаря побочному эффекту, который производится при вычислении некоторых примитивных функций, входящих в выражения. В этом отношении АВГУР напоминает язык С, где вся содержательная работа тоже является побочным продуктом вычисления выражений. Таким образом, смысловым ядром языка АВГУР являются встроенные в него примитивные функции.

6.8.5 Примитивные функции

Ниже для краткости примитивные функции будем называть просто функциями. Прежде чем перейти к описанию отдельных функций, следует сделать несколько общих замечаний. Синтаксис вызова функций обычный, как и в большинстве других языков программирования: сначала идет имя функции, которое в то же время является одним из зарезервированных имен языка (см. 6.8.12), а затем в скобках список аргументов через запятую. Каждый аргумент, функции как уже указывалось, может быть любым числовым выражением, содержащим новые вызовы функций и т.д. Во многих случаях одна и та же функция может вызываться с разным числом аргументов. В некоторых случаях недостающие аргументы заменяются значениями по умолчанию, которые в этом случае оговариваются особо, в других случаях может поменяться сама операция, выполняемая функцией. Если в вызове функции имеется несколько аргументов, то все они вычисляются слева направо, и затем подставляются. Исключением являются функции **eqz, nez, gtz, gez,** в которых аргументы могут пропускаться без вычисления (раздел 6.8.5.7).

Теперь перейдем к детальному описанию отдельных примитивных функций, объединяя при этом их в группы по смыслу. Описания функций будут сопровождаться примерами,

которые являются законченными программами на АВГУРЕ. Соответствующие файлы имеются в дистрибутиве отладчика **drft.exe**.

6.8.5.1 line() и arc()

Это основные функции, при выполнении которых создается код, описывающий образы объектов в двухмерном подмножестве языка АВГУР. При составлении программы удобно представлять себе, что непосредственно при выполнении этих функций происходит отрисовка объектов, которые они обозначают, хотя на самом деле процедура рисования может начаться только после того, как вся программа откомпилирована.

Функции **line()** и **arc()** используют значения координат точек объекта. Компилятор не понимает, как устроена используемая координатная система и не знает как называются оси координат. Будем предполагать, что двухмерные объекты описываются в абсолютной системе координат (**X,Z**), причем ось X направлена слева направо, а ось Z - сверху вниз. Отладчик **drft.exe** будет правильно обрабатывать изображения, если реальные координаты точек объекта не превышают по модулю 1000, хотя для компилятора это не является ограничением.

Функция **line()** генерирует ломаную линию, которая задается координатами своих вершин. Максимальное число вершин 255, минимальное - 2. Формат вызова функции. Скобки [и] не обязательны:

line([x0,z0], [x1,z1], ...)

Функция возвращает количество вершин ломаной. Функция **arc()** генерирует дугу окружности. Формат вызова:

arc([x,z], r, phi0, phi1)

Назначение параметров: **x** и **z** - координаты центра окружности, **r** - радиус, **phi0** - начальный угол, **phi1** - конечный угол (оба в градусах). Должно быть выполнено условие: **0 <= phi0 < phi1 <= 360** (дуга рисуется против часовой стрелки). Параметры **phi0** и **phi1** не обязательные, по умолчанию предполагается phi0 = 0, phi1 =360. Функция возвращает количество переданных аргументов (3 или 5). **Пример № 1** (Рис. 6.7):

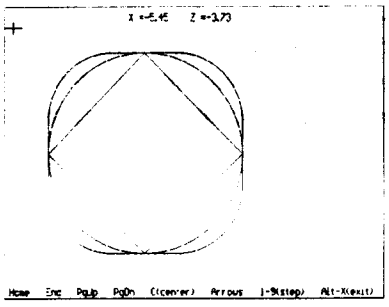

Рис. 6.7 Пример № 1. Применения функций line и arc

```
line([10,0],[20,0]) + line([30,10],[30,20]) +
line([20,30],[10,30]) + line([0,20],[0,10]) +
arc([10,10],10,90,180) + arc([20,10],10,0,90) +
arc([20,20],10,270,360) + arc([10,20],10,180,270) +
arc([15,15],15) +
line([15,0],[30,15],[15,30],[0,15],[15,0]);
```

Заметим, что вся программа примера 1 записана как единое выражение. Это следует считать хорошим стилем программирования на АВГУРЕ. Можно все знаки '+' в примере № 1 заменить на ';', результат остался бы без изменений. Запись в виде единого выражения образа целого объекта обладает двумя преимуществами. Во-первых, такая запись подчеркивает целостность объекта. Если программа описывает несколько относительно независимых объектов, то разумно представить ее в виде соответствующей последовательности выражений. Во-вторых, такая запись позволяет описание целого объекта подставить в качестве аргумента в вызов примитивной функции, ведь аргументы функций - любые выражения. Это может быть полезным при выполнении сложной программы, когда происходит выбор одной из нескольких альтернатив.

6.8.5.2 plate() и cylinder()

Это основные функции, при выполнении которых создается код, описывающий объекты в трехмерном подмножестве языка АВГУР. Они описывают соответственно кусок плоскости, ограниченный отрезками прямых, и часть цилиндрической поверхности. В версии АВГУР 1.0 функции не реализованы.

6.8.5.3 style() и color()}

Функция **style()** задает текущий тип линий: непрерывная или пунктир. По умолчанию установлен непрерывный стиль. Возможны два формата вызова. С аргументом:

style(newStyle)

Если **newStyle = 0** устанавливается непрерывный стиль, если **newStyle = 1** - пунктир. Возвращает новое значение стиля (0 или 1). Без аргумента:

style()

Только возвращает текущее значение стиля.

Функция color() задает текущий цвет линий. Цвет задается числом от 0 (черный) до 15 (белый). По умолчанию установлен цвет 10 (ярко-зеленый). Возможен вызов с аргументом и без аргумента. С аргументом:

color(newColor)

Устанавливает новый цвет **newColor** и его же возвращает. Без аргумента:

color()

Только возвращает текущий цвет. **Пример № 2** (Рис. 6.8):

```
style(1) + color(14) +
line([10,0],[20,0]) + line([30,10],[30,20]) +
line([20,30],[10,30]) + line([0,20],[0,10]) +
arc([10,10],10,90,180) + arc([20,10],10,0,90)+
arc([20,20],10,270,360) + arc([10,20],10,180,270) +
style(0) + color(10) +arc([15,15],15)+
line([15,0],[30,15],[15,30],[0,15],[15,0]);
```

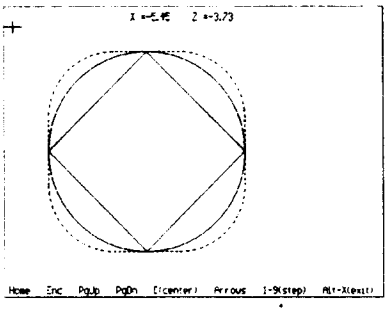

Рис. 6.8 Пример № 2. Применение функции style().

6.8.5.4 shift()

Функция **shift()** задает сдвиг объекта относительно исходной точки отсчета. Формат вызова в двухмерном подмножестве АВГУРА:

shift(deltaX,deltaZ)

в трехмерном подмножестве:

shift(deltaX,deltaY,deltaZ)

В любом случае функция возвращает текущую размерность (2 или 3).Параметры **deltaX, deltaY, deltaZ** задают сдвиг вдоль каждой из осей координат. Новые вызовы добавляют сдвиг к уже имеющемуся, таким образом сдвиг накапливается. **Пример № 3 (**Рис. 6.9**):**

```
arc([20,20],20) + arc([20,20],10);
shift(30,20) + arc([20,20],20) + arc([20,20],10);
shift(60,40) + arc([20,20],20) + arc([20,20],10);
```

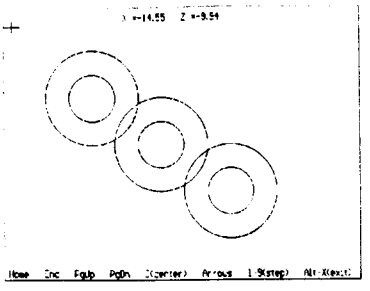

Рис. 6.9 Прмер № 3. Применение фунции shift().

6.8.5.5 scale()

Функция задает масштабирование объектов. Она действует только на параметры функций **line(), arc(), plate() {\rm и} cilinder()** и не действует на сдвиги, устанавливаемые функцией **shift()**. Может быть вызвана с одним аргументом и без аргументов. С одним аргументом:

scale(newScale)

newScale - новое значение масштабирующего множителя (вещественное число, может быть и отрицательным). Возвращает вновь установленное значение.

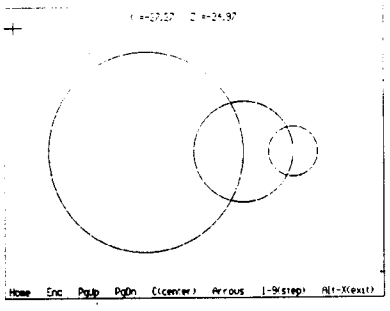

Рис. 6.10 Пример № 4. Прменение функции scale().

Без аргументов:

scale()

Возвращает текущее значение масштабирующего множителя.**Пример № 4 (**Рис. 6.10**):**

```
arc([0,0],20);
shift(20,0) + scale(0.5) + arc([0,0],20);
shift(30,0) + scale(0.25) + arc([0,0],20);
```

6.8.5.6 dim()

Функция устанавливает размерность, т.е. выбирает между двухмерным и трехмерным подмножеством АВГУРА. Может быть вызвана с одним аргументом или без аргументов. С одним аргументом:

dim(newDimension)

устанавливает размерность **newDimension = 2** или **3**. Возвращает установленную размерность. При входе в программу по умолчанию уже установлена размерность 2, поэтому в двухмерной задаче пользоваться функцией **dim()** не обязательно.

Без аргументов:

dim()

Функция возвращает текущую размерность. Размерность можно менять сколько угодно раз до тех пор, пока не будет вызвана одна из функций **line, arc, plate, cilinder** и **shift**, после этого размерность нельзя менять, но по-прежнему можно читать. Разумно устанавливать размерность (если нужно) один раз в первой строке программы, до включения библиотек.

6.8.5.7 eqz(), nez(), gtz() и gez()

Это функции условий, или условные функции. Эти функции напоминают условную операцию языка С, основное отличие состоит в том, что возможен только фиксированный набор условий. Функции принимают три обязательных аргумента и имеют форматы вызова:

eqz(condition, expr1, expr2)

nez(condition, expr1, expr2)

gtz(condition, expr1, expr2)

gez(condition, expr1, expr2)

Функции работают следующим образом. Сначала вычисляется значение выражения **condition**, и если значение этого выражения удовлетворяет некоторому условию, которое для каждой из функций свое, то вычисляется выражение **expr1** и функция возвращает значение этого выражения. В этом случае выражение **expr2** *не вычисляется*, а просто пропускается. Таким образом, если **expr2** содержало функции, определяющие объекты или режимы рисования - они будут пропущены. Если условие не выполняется, то наоборот, выражение **expr1** *пропускается*, а выражение **expr2** вычисляется и результат вычисления возвращается. Условия, о которых шла речь, следующие:

eqz: condition = 0

nez: condition # 0

gtz: condition > 0

gez: condition >= 0

Существенно, что функции не требуют соблюдения условий со всей точностью, которая характерна для 4-байтовых чисел с плавающей запятой, которые представляют вещественные числа в языке АВГУР. Требуется соблюдение условий с точностью 0.01. Таким образом, точный смысл приведенных выше условий таков:

eqz: abs(condition) < 0.01

nez: abs(condition) >= 0.01

gtz: condition >= 0.01

gez: condition > -0.01

Пример № 5 (Рис. 6.11):

```
eqz(0, line([0,0],[10,10]), line([10,0],[0,10]));
eqz(1, arc([5,5],5), arc([5,5],2));
```

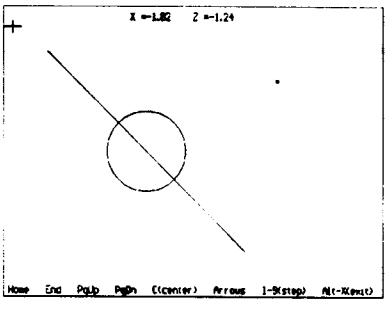

Рис. 6.11 Пример № 5. Применение функции eqz().

Заметим, что в первой функции **eqz()** отрисован только первый аргумент, а во второй - только второй.

6.8.5.8 Инвертирование маски относительно вертикальной оси - invertx()

Для инвертирования маски относительно произвольно расположенной вертикальной оси в язык АВГУР введена функция **invertx( )**. Функция может быть вызвана с двумя или с одним аргументом. Формат вызова с двумя аргументами:

invertx(activity, x0);

Первый аргумент может быть равен единице или нулю, что означает, соответственно, включение или выключение режима инвертирования. Второй аргумент задает X-координату вертикальной оси, относительно которой будет производиться инвертирование. Таким образом, в принципе в тексте программы инвертирование может включаться и выключаться несколько

раз, причем в разных местах могут вводиться разные оси инвертирования. Возвращает функция значение аргумента **activity.** Формат вызова функции с одним аргументом имеет вид

invertx(activity);

что позволяет просто включить или выключить инвертирование.

Необходимо пояснить, каким образом взаимодействует функция **invertx( )** с двумя другими функциями преобразования маски в целом: **shift( )** и **scale( ).** Эти три функции можно выстроить по приоритетам в соответствии с тем порядком, в котором они выполняются (если, конечно некоторые из этих функций одновременно активны). В первую очередь выполняется инвертирование, затем выполняется масштабирование уже инвертированного рисунка, и, наконец, результат этих преобразований сдвигается. Существенно, что на величины и направления сдвигов инвертирование не оказывает никакого влияния. Отметим также, что порядок выполнения этих операций не зависит от того, в каком порядке соответствующие функции вызывались в программе.

6.8.5.9 min() и max()

Функции принимают два обязательных аргумента, вычисляют оба и возвращают, соответственно, минимальное и максимальное значение.

6.8.6 Элементарные функции

В языке АВГУР в качестве примитивных функций определены следующие элементарные функции:

sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), sqrt(x), square(x), abs(x)

Они имеют обычный смысл и область определения. Функция **square(x)** означает возведение в квадрат. Аргументы тригонометрических функций должны выражаться в градусах, обратные тригонометрические функции тоже возвращают значение угла в градусах. Все функции принимают один обязательный аргумент.

6.8.7 Регистры и стек

В языке АВГУР отсутствует понятие числовой переменной, однако часто все-таки требуется запоминать промежуточные результаты некоторых вычислений. Для того, чтобы обеспечить эту возможность, в языке определены 20 регистров, в которые можно записывать и считывать вещественные числа, и стек на 100 позиций. Доступ к этой памяти осуществляется через соответствующие примитивные функции (см. ниже). В пользовательских программах для запоминания результатов вычислений мы рекомендуем использовать регистры, так как это значительно проще, чем использование стека. Стек лучше использовать при создании библиотечных команд, так как это позволяет инкапсулировать данные внутри самих команд (с регистрами это невозможно, так как они определены глобально) и достичь, тем самым, определенного уровня абстракции. Однако при использовании стека возникают некоторые тонкие проблемы, особенно при передачи данных стека в качестве аргументов макрокоманд.

6.8.7.1 Регистры reg()

Для доступа к регистрам определена функция **reg()**. Она может быть вызвана либо с двумя, либо с одним аргументом. Формат вызова с двумя аргументами:

reg(numb,val)

В этом формате в регистр номер **numb** (от 0 до 19) записывается значение выражения **val**. Функция возвращает только что записанное число. Формат вызова с одним аргументом:

reg(numb)

Функция возвращает число из регистра номер **numb** не изменяя содержимого этого регистра.

Допускаются сколь угодно сложные вложения обращений к регистрам. Например, присваивание $r_0 = r_0+r_1$ может быть записано как **reg(0,reg(0) + reg(1))**. **Пример № 6** (Рис. 6.12)**:**

```
reg(0,10) + reg(1,0) +
line([reg(0),reg(1)], [reg(1),reg(0)]);
\pict{examp06}
```

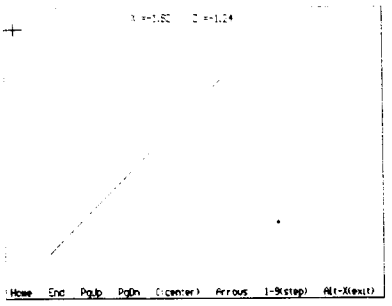

Рис. 6.12 Пример № 6. Применение регистров reg().

6.8.7.2 Стек

Стек, определенный в языке АВГУР, работает так же, как и все обычные структуры данных этого сорта: по принципу «последним вошел - первым вышел»'. Доступ к данным стека возможен двумя путями: с помощью обычных в этом случае процедур **push()** и **pop()** и прямым образом, через смещение относительно вершины стека. Вершиной стека мы называем последнее введенное в стек число. Явное положение вершины стека (как, впрочем, и дна) в программе недоступно - т.е. никак нельзя определить, сколько элементов записано в стек в данный момент. Манипуляции со стеком осуществляются с помощью следующих примитивных функций:

push()

Формат вызова: **push(x)**. Добавляет к стеку значение выражения **x** и возвращает значение этого выражения.

pop()

Формат вызова: **pop()**. Удаляет из стека последнее введенное число и возвращает значение этого числа.

get()

Формат вызова: **get(offset)**. Возвращает содержимое стека со смещением **offset** относительно вершины. Содержимое стека не меняет. Если **offset = 0**, то возвращает число из вершины стека, если **offset = 1**, то на одну позицию ниже, и т.д.

put()

Формат вызова: **put(offset,val)**. По смещению **offset** относительно вершины стека пишет значение выражения **val**. Возвращает значение **val**.

add()

Формат вызова: **add(n)**. Добавляет **n** свободных позиций к вершине стека (т.е. передвигает вершину стека на **n** позиций вверх). Содержимое новых ячеек остается неопределенным. Возвращает количество вновь зарезервированных позиций **n**.

del()

Формат вызова: **del(n)**. Удаляет **n** позиций стека считая от вершины (т.е. передвигает вершину стека на **n** позиций вниз. Возвращает значение **n**.

6.8.8 Макрокоманды

Язык АВГУР позволяет определять макрокоманды, подобные макроопределениям языка С. Однако в АВГУРЕ макрокоманды играют существенно более важную роль, чем в языке С. В то время, как в С макроопределения только сокращают и упрощают текст программы, в АВГУРЕ это единственное средство создания внешних библиотек объектов. Таким образом, в АВГУРЕ библиотеки это есть макропакеты или макрорасширения языка. Как и в языке С, в АВГУРЕ макрокоманды обрабатываются препроцессором, который динамически раскрывает текст макрокоманд, когда ссылки на них встречаются в тексте программы. Это не единственная функция препроцессора, см. также **include** и **stop**. Собственно компилятор получает входной поток данных не непосредственно из исходного файла, а от препроцессора. Компилятор не знает происхождение данных, которые он обрабатывает: берутся ли они из исходного файла, являются ли результатом раскрытия макрокоманды, или откуда-то еще. Ниже для краткости мы будем называть макрокоманды просто командами.

Команды идентифицируются по их именам. Именем команды может быть любая последовательность из латинских букв и цифр длиной не более 32 символов и начинающаяся с буквы. Другие символы в именах не допускаются, большие и малые латинские буквы различаются. Команда с одним и тем же именем может определяться сколько угодно раз (в том числе и с различным числом аргументов), при этом действительным будет последнее определение. Возможны команды двух типов: без аргументов и с аргументами. Для них определен разный синтаксис описания и вызова. Рассмотрим эти разновидности команд по порядку.

6.8.8.1 Команды без аргументов

Команда без аргументов это просто любой {ну, почти любой.} кусок текста, который получает имя. Если затем в тексте программы встретится это имя (причем - в любом месте), то вместо него чисто механически вставляется соответствующий фрагмент текста и компиляция продолжается так, как будто он с самого начала был в исходном коде. Синтаксис определения команды без аргументов такой:

имя_команды = текст_команды;

Ограничителями определения команды являются фигурные скобки {} и \. Определение команды является предложением АВГУРА, но не является выражением, это - инструкция препроцессора. Следовательно, правило размещения определений команд такое же, как и любых других предложений. Определение не может встретиться внутри числового выражения.

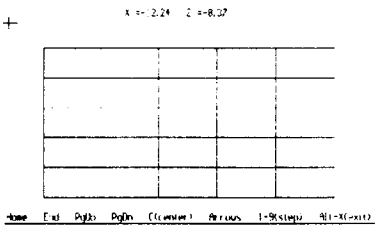

Рис. 6.13 Пример № 7. Применение определений команды.

Формальным признаком места, в котором может встретиться определение команды, является следующее: определение команды может встретиться в любом месте, где может быть оборван файл без того, чтобы привести к синтаксической ошибке. Небольшие ограничения на текст команды все же существуют. Во-первых, в тексте команды не может встретиться определение другой команды. Во-вторых, команда не может определять список аргументов другой команды, или даже его часть. Например, если команда **B** была определена как команда с тремя аргументами, то конструкция **B(1,2,3);** будет, конечно верной, но такой фрагмент: **\{A=1,2,3\}; B(A);** скорее всего приведет к ошибке. В тексте команды **B** будет произведена подстановка **1,2,3** вместо первого аргумента, а второй и третий аргументы будут определены по умолчанию, т.е. заменены символом 0. **Пример № 7** (Рис. 6.13):

```
h = line([0,0], [100,0]);
v = line([0,0], [0,50]);
```

```
v +
shift(20,0) + v +shift(20,0) + v +shift(20,0) + v +
shift(20,0) + v +shift(20,0) + v +shift(-100,0) + h +
shift(0,10) + h +shift(0,10) + h +shift(0,10) + h +
shift(0,10) + h +shift(0,10) + h;
```

6.8.8.2 Команды с аргументами

Команды с аргументами это такие макроопределения, текст которых не является полностью определенным. Этот текст содержит переменные куски, которые определяются только в момент вызова команды: соответствующие фрагменты текста передаются команде в виде аргументов. В теле определения команды эти переменные куски - аргументы обозначаются следующим образом: **#1, #2, #3, #9, #A, #B, ... ,#Z.** Здесь **#1** обозначает первый аргумент команды, **#2** - второй, **#A** - десятый и т.д. Латинские буквы в обозначениях аргументов допускаются только большие. Таким образом, всего макрокоманда может иметь до 9+26=35 аргументов. Синтаксис определения макрокоманды с аргументами таков:

{ имя_команды(количество_аргументов) = текст_команды };

В этом определении количество аргументов должно быть задано **целым числом** от 1 до 35 - не выражением! Значение выражений умеет вычислять только компилятор, а определение команды обрабатывается исключительно препроцессором, который этого делать не может. **Пример:**

Если команда **sum** определена как:

```
{ sum(5) = +#1    ! первый аргумент
          + #2    ! второй аргумент
          + #3    ! третий аргумент
          + #4    ! четвертый аргумент
          + #5 }; ! и пятый аргумент.
```

то фрагмент программы **sum(10,20,30,40,50)** приведет к подстановке и направит компилятору на обработку код **10+20+30+40+50.** Заметим так же, что препроцессор выкидывает из вашего определения команды все пробелы и вообще все, что не относится к делу.}. Вместо аргументов может быть передано и «ничего»': вызов **sum(10,,,,)** приведет к подстановке **10++++.**

Текст команды может и не содержать ссылок на некоторые аргументы. Тогда эти аргументы никуда не будут подставляться, хотя и могут быть переданы в вызове команды. Например, если sum(5) = #1 + #5 ; то фрагмент **sum(10,20,30,40,50)** приведет к подстановке **10+50**. Если команда была определена как команда с n аргументами, то не обязательно все n аргументов передавать в момент вызова этой команды. Если в вызове было передано лишь k<n аргументов, то считается, что это были первые k аргументов, а оставшиеся n-k аргументов определяются по умолчанию символом **'0'**. Например, если опять **{sum(5)=#1+#2+#3+#4+#5};** то вызов **sum(10,20)** приведет к подстановке **10+20+0+0+0**. Можно вызвать команду вообще без аргументов, тогда все аргументы будут переданы по умолчанию, но пустые круглые скобки в вызове команды все равно должны присутствовать: вызов **sum()** раскроется в **0+0+0+0+0**. Без скобок вызываются только команды, которые были определены как команды без аргументов с самого начала.

Механизм передачи аргументов по умолчанию очень важен, и играет, в частности, большую роль в библиотеке **lib.hdr**. Фактически этот механизм вместе с условными функциями обеспечивает передачу числовых аргументов с любым удобным умолчанием. Например, пусть мы хотим, чтобы некоторый параметр, например угол, по умолчанию

принимал значение 45, и пусть этот угол является (для определенности) пятым аргументом некоторой команды. Тогда в тексте команды всюду, где встречается этот угол, мы можем написать

eqz(#5, 45, #5)

Теперь если вместо **#5** оказался 0, т.е. пятый аргумент передавался по умолчанию, то **eqz** вернет значение 45. В противном случае будет предпринята попытка вычислить выражение, подставленное в вызове вместо аргумента **#5**, и если это получится - **eqz** вернет значение этого выражения. Имеется один случай подстановки аргументов по умолчанию, который требует специальной обработки. Этот случай относится к передаче по умолчанию аргумента команде, которая была определена как команда с одним аргументом. Пусть ее имя, например, **mac**. Тогда синтаксис **mac()** не позволяет понять, что мы хотим: передать команде **mac** ее единственный аргумент по умолчанию, или хотим передать пустое место в качестве аргумента. Для того, чтобы различить эти две ситуации принимается соглашение, что синтаксис **mac()** означает передачу параметра по умолчанию, а если мы действительно хотим передать пустое место, надо использовать такой синтаксис: **mac(|)**.В этом состоит то единственное отличие символов'|','|' от пробелов разного сорта. **Пример № 8 (Рис. 6.14):**

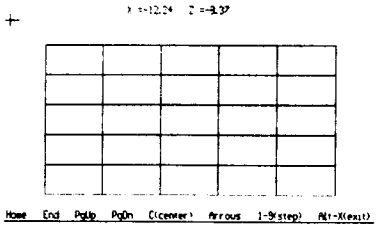

Рис. 6.14 Пример № 8.

```
{ h(1) = shift(0,#1) + line([0,0], [100,0]) };
{ v(1) = shift(#1,0) + line([0,0], [0,50]) };
v(0) + v(20) + v(40) + v(60) + v(80) + v(100) +
h(0) + h(10) + h(20) + h(30) + h(40) + h(50);
```

В качестве нетривиального примера использования команд с аргументами приведем полностью разработанную библиотечную команду **oval()**. Она описывает прямоугольник со скругленными углами. Задаются координаты центра прямоугольника, его полуоси и радиус скругления. По умолчанию и при зашкале радиуса он принимается равным меньшей из полуосей. Таким образом, эта команда не только принимает аргументы по умолчанию, но и обрабатывает ошибочные ситуации. **Пример № 9** (Рис. 6.15)**:**

```
{ oval(5) = (add(11) + put(0,#1) + put(1,#2) + put(2,#3) + put(3,#4) + put(4,#5) +
put(5, min(get(2), get(3))) +  put(6, eqz(get(4),          ! Вычисляем радиус:
get(5), gtz(get(4) - get(5),                                ! обрабатываем умолчание
get(5), get(4)))) + ! и зашкал
put(7,  get(0) - get(2) + get(6)) + put(8,  get(0) + get(2) - get(6)) +
put(9,  get(1) + get(3) - get(6)) + put(10, get(1) - get(3) + get(6)) +
line([get(7), get(1) - get(3)], [get(8), get(1) - get(3)]) +
line([get(7), get(1) + get(3)], [get(8), get(1) + get(3)]) +
line([get(0) - get(2), get(10)], [get(0) - get(2), get(9)]) +
line([get(0) + get(2), get(10)], [get(0) + get(2), get(9)]) +
arc(get(7), get(10), get(6),  90, 180) +
arc(get(8), get(10), get(6),   0,  90) +
arc(get(8), get( 9), get(6), 270, 360) +
arc(get(7), get( 9), get(6), 180, 270) +del(11)) };
```

```
oval( 0,  0, 20, 10);          ! R по умолчанию
oval( 0, 45, 10, 20);
oval(40,  0, 10, 20,  5);      ! R явно
oval(40, 45, 20, 10,  5);
oval(80,  0, 20, 10, 100);     ! R в зашкале
oval(80, 45, 10, 20, 100);
```

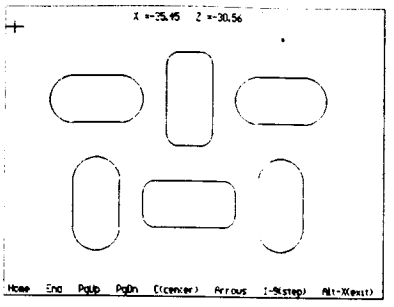

Рис. 6.15 Пример № 9,

6.8.8.2.1 Более сложные случаи макрокоманд

В тексте команды может встретиться обращение к другой команде, в свою очередь эта вложенная команда может вызвать другие и т.д. Например, такой фрагмент программы

```
{ A(1) = 1 + #1 };
{ B(1) = 2 + A(#1) };
{ C(1) = 3 + B(#1) };
{ D(1) = 4 + C(#1) };
D(5);
```

раскроется в последовательность **4+3+2+1+5;** в чем можно убедиться, посмотрев log-файл. Возможны и многократные вложения в аргументах. Фрагмент программы

```
{ A(1) = 1 + #1 };
{ B(1) = 2 + #1 };
{ C(1) = 3 + #1 };
{ D(1) = 4 + C(B(A(#1))) };
D(5);
```

раскроется в ту же последовательность **4+3+2+1+5;** . Это все понятно и не вызывает удивления. Однако могут встретиться конструкции, которые на первый взгляд могут показаться бессмысленными. Рассмотрим, например, такой фрагмент программы:

```
{ A = #1 + #2 + #3 + #4 };
{ B(5) = A + #5 };
B(10,20,30,40,50);
```

На первый взгляд определение команды **A без аргументов** кажется неверным: откуда в ее тексте берутся значки аргументов **\#1,...** , если это команда без аргументов? И действительно, изолированный вызов команды **A** в виде **A;** приведет к ошибке. Однако в контексте тела команды **B**, принимающей 5 аргументов, команда **A** становится осмысленной: вместо аргументов **\#1,...,\#4** будут подставлены те аргументы, которые получает команда **B**. Поэтому приведенный выше фрагмент раскроется в последовательность **10+20+30+40+50**. Возможны еще более запутанные случаи. Рассмотрим фрагмент:

```
{ A(2) = #1 + #2 + #3 + #4 };
{ B(5) = A(#3,#4) + #5 };
B(10,20,30,40,50);
```

Во что раскроется эта последовательность? При раскрытии команды **A**, которая встречается в теле команды **B**, препроцессор сначала нормально обработает аргументы **#1** и **#2** команды **A**. Он обнаружит, что вместо них нужно подставить 3-й и 4-й аргументы, которые фигурировали в вызове команды **B** (а не 1-й и 2-й аргументы в вызове **B**!) - это будут числа 30 и

40. Затем препроцессор увидит, что встретился аргумент **#3**, который не может иметь отношения к команде **A**, т.к. у нее всего два аргумента. Поэтому препроцессор сообразит, что этот аргумент в теле команды **A** мог появиться только из команды более низкого уровня. Он начнет искать такую команду, у которой может быть аргумент **#3** и первой ему встретится команда **B** (а могло бы быть и так, что ему пришлось бы пройти через длинную цепочку команд, прежде чем он нашел бы нужный аргумент). Тогда он возьмет третий аргумент в вызове **B** и подставит его вместо #3. Аналогично он поступит с аргументом **#4**. Таким образом, весь фрагмент раскроется в последовательность **30+40+30+40+50;**. Мы видим, что аргументы **#1, #2** и **#3, #4** команды **A** обрабатываются совершенно по-разному.

Конечно, постоянное использование таких нетривиальных вложений команд в программе нельзя считать хорошим стилем программирования. Лучше аргументы передавать в явном виде (по мере возможности). Однако в некоторых случаях неявная передача аргументов может привести к существенному сокращению текста программы, и, кроме того, логика макроопределений требует, чтобы такие конструкции были допустимы.

6.8.9 Включение файлов в компиляцию: include()

Обработка макрокоманд - это не единственная функция препроцессора. Другой обязанностью препроцессора является обработка включаемых файлов. Синтаксис включения в компиляцию нового файла такой:

include(имя_файла);

Как и определение команды, **include()** не является ни выражением, ни даже функцией, а представляет собой отдельное предложение языка и обозначает инструкцию препроцессора. Правило, определяющее где в файле может встретиться **include()**, то же, что и для определения команд. Назначение инструкции **include()** состоит в том, что в текст программы включается текст файла, заданного в аргументе, и компиляция продолжается так, будто текст включаемого файла является частью текста исходного. Включаемые файлы сами могут содержать включение новых файлов и т.д. Имеется ограничение на содержимое включаемого файла, которое состоит в том, что он должен содержать целое число предложений АВГУРА (или ни одного). Реальный смысл от использования включаемых файлов состоит, главным образом, в том, чтобы подключить к файлу исходной программы файлы библиотек. Ясно, что инструкцию **include()** разумно поместить где-то в самом начале текста программы, после определения размерности, если эта возможность используется.

6.8.10 Безусловное окончание: stop

Инструкция препроцессора **stop** вызывает безусловное окончание компиляции программы, даже если она встретилась во включаемом файле. Как и все другие инструкции препроцессора, **stop** представляет собой целое предложение АВГУРА и не может входить в выражения. Формат инструкции:

stop;

Альтернативным способом окончания программы является только достижение конца исходного файла.

6.8.11 Библиотека lib.hdr

Пока создан всего один библиотечный файл для языка АВГУР, и он содержит совсем немного определений команд. Ниже перечислены все эти команды с описаниями их назначения и форматов вызова. В формате вызова аргументы команд представлены значками **\#1,\#2,...** Во всех случаях вместо аргументов должны подставляться выражения (или просто числа). Ни одна из библиотечных команд не меняет стиля и цвета линий, и использует их текущие значения.

6.8.11.1 rect()

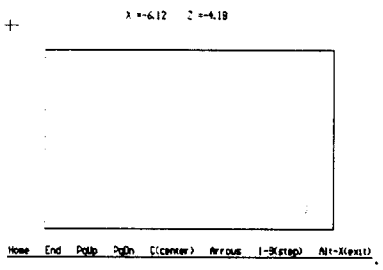

Рис. 6.16 Пример № 10. Применение функции rect() из библиотеки lib.

Формат команды:

rect([#1,#2], [#3,#4])

Команда описывает прямоугольник. **[\#1,\#2]** - координаты левого верхнего угла, **[\#3,\#4]** - координаты правого нижнего угла. **Пример № 10** (Рис. 6.16)**:**

```
include(lib);
rect([0,0], [50,30]);
```

6.8.11.2 hArc()}

Формат команды:

hArc(#1, #2, #3, #4)

Эта команда представляет альтернативную возможность описания дуги окружности (по сравнению с примитивной функцией **arc()** заданием положения ее крайних точек и стрелы прогиба. Данная команда описывает только ``горизонтальные" дуги в том смысле, что Z-координаты обеих концов должны совпадать. Назначение параметров:

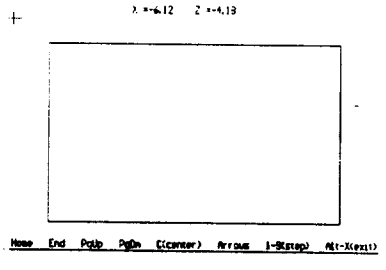

Рис. 6.17 Пример № 11. Применение функции hArc() из библиотеки lib.

#1 - X-координата левого конца дуги;

#2 - X-координата правого конца дуги;

#3 - Общая Z-координата для левого и правого края дуги;

#4 - Стрела прогиба. Положительное значение означает выпуклость вверх, отрицательное значение - выпуклость вниз, если нуль или аргумент отсутствует, дуга изобразится в виде отрезка прямой. **Пример № 11** (Рис. 6.17)**:**

```
include(lib);
hArc(0,100,0, 20);
hArc(0,100,0,-20);
hArc(0,100,0,  0);
```

6.8.11.3 vVeld()}

Формат команды:

vVeld([#1,#2], #3, #4, #5, #6, #7, #8)

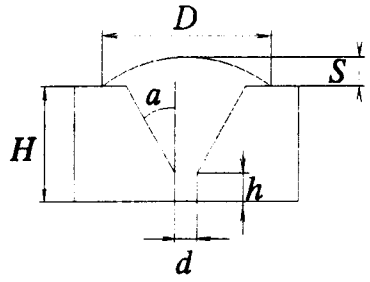

Рис. 6.18 Параметры функции vVeld() из библиотеки lib.

Команда описывает V-образный сварной шов. Назначение параметров (Рис. 6.18):

[#1,#2] - точка привязки, середина верхней грани шва на уровне поверхности свариваемых плоскостей;

#3 - полная высота шва H;

#4 - половина угла раскрыва a, в градусах;

#5 - высота нижней фаски h;

#6 - ширина зазора d;

#7 - высота валика усиления S;

#8 - ширина валика усиления D.

Если ширина валика усиления не передается, то по умолчанию устанавливается равной ширине верхней грани шва. Все остальные параметры по умолчанию устанавливаются в 0. **Пример № 12 (Рис. 6.19):**

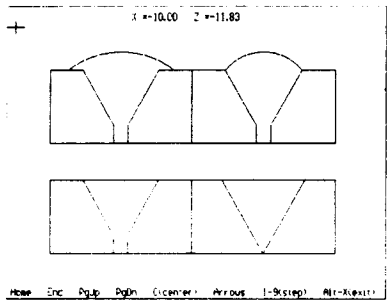

Рис. 6.19 Пример № 12. Применение функции vVeld().

```
include(lib);
{ FRAME =                    ! Команда для отрисовки рамки
        line([9.34,0],[0,0],[0,20],[40,20],[40,0],[30.73,0]) };
! Передаем все параметры явно
vVeld([20,0], 20, 30, 5, 4, 5, 30) + FRAME;
! Ширина валика усиления по умолчанию
shift(40,0); vVeld([20,0], 20, 30, 5, 4, 5) + FRAME;
! Ширина и высота валика усиления по умолчанию
shift(-40,30); vVeld([20,0], 20, 30, 5, 4) + FRAME;
! Ширина и высота валика усиления и ширина и высота зазора у основания шва по умолчанию
shift(40,0); vVeld([20,0], 20, 30) + FRAME;
```

6.8.11.4 xVeld()

Формат команды:

xVeld([#1,#2], #3, #4, #5, #6, #7, #8, #9, #A, #B, #C)

Команда описывает X-образный сварной шов:

Назначение параметров (Рис. 6.20):

[\#1,\#2] - точка привязки, середина верхней грани шва на уровне поверхности свариваемых плоскостей;

\#3 - полная высота шва H;

#4 - половина верхнего угла раскрыва a_1, в градусах;

#5 - половина нижнего угла раскрыва a_2, в градусах;

#6 - высота нижней половины шва h_1;

#7 - высота фаски h_2;

#8 - ширина зазора d (не обозначена на чертеже, аналог параметра d для V-образного шва.)

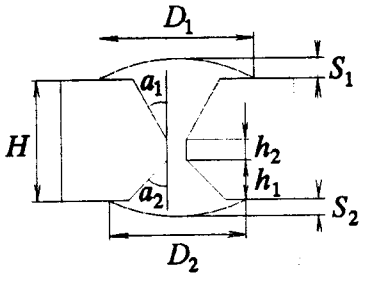

Рис. 6.20 Параметры функции xVeld() из библиотеки lib.

#9 - высота верхнего валика усиления S_2;

#A- высота нижнего валика усиления S_2;

#B - ширина верхнего валика усиления D_1;

#C - ширина нижнего валика усиления D_2;

По умолчанию ширины валиков усиления равны ширинам соответствующих оснований шва. Все остальные параметры по умолчанию устанавливаются в 0. **Пример № 13** (Рис. 6.21):

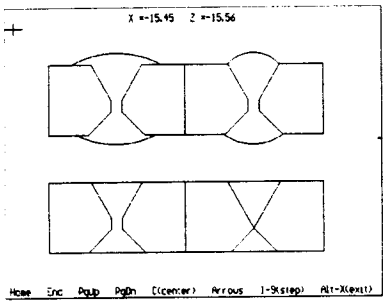

Рис. 6.21 Пример № 13. Применение функции xVeld().

```
include(lib);
! команда для отрисовки рамки
{ FRAME = line([18.86, 0], [0, 0], [0, 30], [17.51, 30]);
          line([42.52,30], [60, 30], [60, 0], [41.17, 0]) };
! Все параметры явно
xVeld(30,0,30,30,45,10,5,5,5,4,40,35) + FRAME;
! Ширины валиков усиления по умолчанию
shift(60,0); xVeld(30,0,30,30,45,10,5,5,5,4) + FRAME;
! Ширины и высоты валиков усиления по умолчанию
shift(-60,50); xVeld(30,0,30,30,45,10,5,5) + FRAME;
! Валики усиления, высота фаски и зазора по умолчанию
shift(60,0); xVeld(30,0,30,30,45,10)+FRAME;
shift(-60,-50); line([60, 80], [120, 80]);
```

6.8.11.5 oval()

Формат команды:

oval([#1,#2], #3, #4, #5)

Рисует прямоугольник со скругленными краями (овал). Назначение параметров:

[\#1,\#2] - координаты центра овала;

#3 - горизонтальная полуось овала;

#4 - вертикальная полуось овала;

#5 - радиус скругления.

По умолчанию радиус скругления равен минимальной из полуосей овала. Если радиус задан явно, и он оказался больше минимальной полуоси, то он отсекается до минимальной полуоси. **Пример № 14** (Рис. 6.22):

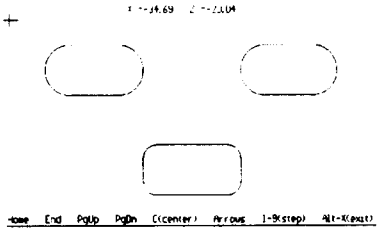

Рис. 6.22 Пример № 14. Применение функции oval().

```
include(lib);
oval([0,0], 20, 10);                    ! R по умолчанию
shift(40,40) + oval([0,0], 20, 10,  5);  ! R явно
shift(80, 0) + oval([0,0], 20, 10, 100);! R в зашкале
```

6.8.12 Ключевые слова языка АВГУР

Язык АВГУР имеет следующие ключевые слова:

abs & cylinder & function & max & put & square & acos & color & get & min & reg & stop & add & cos & gez & nez & scale & style & arc & del & gtz & plate & shift & tan & asin & dim & include & pop & sin & atan & eqz & line & push & sqrt.