МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

АВТОМАТИЗАЦИЯ ПАРАМЕТРИЧЕСКОГО ПРОЕКТИРОВАНИЯ

Методические указания
к выполнению лабораторных работ
для студентов всех форм обучения направления
12.03.01 – Приборостроение (квалификационный уровень – бакалавр)
15.03.04 – Автоматизация технологических процессов и производств
(квалификационный уровень – бакалавр)

Севастополь
СевГУ
2019

УДК 681. 3.06
А22

Рецензент:
А.О. Харченко - профессор кафедры «Технология машиностроения»
СевГУ, канд. техн. наук

Составители: Л.Е. Карташов, М.А. Худаймуратов, С.Н. Федоренко

А22 **Автоматизация параметрического проектирования:** метод. указания к выполнению лабораторных работ для студентов всех форм обучения направления 12.03.01 – Приборостроение (квалификационный уровень – бакалавр), 15.03.04 – Автоматизация технологических процессов и производств (квалификационный уровень – бакалавр) / Сост. Л.Е. Карташов, М.А. Худаймуратов, С.Н. Федоренко. – Севастополь: СевГУ, 2019. – 59 с.

Методические указания предназначаются для оказания помощи студентам при выполнении лабораторных работ по дисциплине «Системы автоматизированного проектирования средств измерений» и «Системы автоматизированного проектирования». В указаниях определены цели и задачи работ, даны методические советы по выполнению заданий.

Объектом изучения является разработка прикладных программ для автоматизации параметрического проектирования в среде AutoCAD. Прикладные программы разрабатываются с использованием файлов сценария, языка AutoLISP и языка описания диалоговых окон.

Предназначены для студентов дневной и заочной форм обучения по направлению 12.03.01 – Приборостроение (квалификационный уровень – бакалавр) и 15.03.04 – Автоматизация технологических процессов и производств (квалификационный уровень – бакалавр).

УДК 681. 3.06

Методические указания рассмотрены и утверждены на заседании кафедры АТПП, протокол № 9 от 28 февраля 2019 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Система AutoCAD – это наиболее распространенная система для ПЭВМ, ориентированная на автоматизированное проектирование в машиностроении. По данным на 1991 г. в мире имелось около 500 тыс. зарегистрированных копий AutoCAD. Можно утверждать, что на каждом предприятии, где стоят задачи автоматизации конструкторских работ, AutoCAD используется или осваивается.

Однако конструктор, освоивший графический редактор, обнаружит, что выигрыш во времени по сравнению с традиционным способом незначителен, а иногда его и вовсе нет.

Экспертные оценки дают следующие цифры: от уменьшения времени на 20% до его увеличения в 2–3 раза. В каждом конкретном случае эффект сокращения времени сильно зависит от характера чертежа.

Конструктор, освоивший графический редактор ACAD и убедившийся на практике, что эффект от автоматизации незначителен, обнаружит, что система AutoCAD предоставляет более эффектные возможности, чем графический редактор. Оказывается, чертежи можно получать без затрат времени и усилий на рисование. Чертеж, который при помощи графического редактора получается за несколько дней, может быть создан за минуты. Более того, чертеж может получить конструктор, практически ничего не знающий о системе AutoCAD.

Но чтобы получить такой эффект, кто-то должен написать программу для генерации семейства сходных чертежей. Этой программой может пользоваться любой конструктор для получения чертежа, принадлежащего данному семейству.

Для создания программ, выпускающих чертежи, в системе AutoCAD имеется мощное средство – язык AutoLISP.

Чтобы использовать богатые возможности AutoLISP, нужно научиться программировать на этом языке. Студенту, ранее не знакомому с программированием, освоить язык графического программирования AutoLISP сложнее, чем язык графического редактирования (или, как говорят, “электронный кульман”). Однако практика преподавания дисциплины показывает, что нашим студентам это вполне по силам.

Данные лабораторные работы позволят освоить использование файлов сценария и научиться программировать на языке AutoLISP для автоматизации проектных работ в среде AutoCAD.

1 ЛАБОРАТОРНАЯ РАБОТА № 1: «ИССЛЕДОВАНИЕ ВОЗМОЖНОСТЕЙ ПРИМЕНЕНИЯ ФАЙЛОВ СЦЕНАРИЯ ДЛЯ АВТОМАТИЗАЦИИ ПРОЕКТИРОВАНИЯ»

Цель работы

Научиться создавать и выполнять файлы сценария. Исследовать применимость файлов сценария для автоматизации черчения. Рассмотреть особенности выполнения команд системы AutoCAD из файлов сценария и возможность выполнения команд редактирования. Оценить время разработки файла сценария и время создания чертежа в интерактивном режиме.

Теоретический раздел

Рассмотрим, что такое программа (и, в частности, графическая программа, формирующая изображение), отталкиваясь от графического редактирования – уже знакомого способа формирования изображения, который применяется при создании чертежей в среде AutoCAD.

Этот способ заключается в следующем: конструктор определяет характер очередного действия; конструктор вводит соответствующую команду в ЭВМ; программа AutoCAD интерпретирует эту команду, т.е. разворачивает ее в последовательность команд процессора ЭВМ; затем эта последовательность команд выполняется; результат выполнения отображается на экране.

Таким образом, рассматриваемый способ характеризуется тем, что конструктор мыслит дискретными командами, а вся их последовательность заранее не определена и присутствует в его голове лишь как общий замысел, который уточняется и корректируется в ходе создания изображения.

В ACAD имеются средства создания сценариев. Сценарий – это тоже последовательность команд графического редактора, но составленная заранее и записанная в виде файла на магнитном диске. Преимущество сценария в том, что его можно использовать многократно. Однако, составить сценарий значительно сложнее, чем работать на “электронном кульмане”, так как он записывается на бумаге без мгновенного отображения на экране. Такая работа “вслепую” приведет к тому, что в сценарии могут остаться ошибки, которые после запуска сценария придется отыскивать в его тексте и исправлять.

Сценарий, написанный на языке команд ACAD, можно рассматривать как частный случай программы на графическом языке программирования. Программа – это тоже заранее составленный текст, который хранится на магнитном диске и может быть использован для генерации изображения. Технология составления и использования программы такая же, как и для сценария.

Отличие программы (в данном случае, программы на AutoLISP) от сценария состоит в том, что язык программирования обладает дополнительными возможностями по сравнению с системой команд AutoCAD и позволяет выполнять более сложные действия и записывать действия более компактно.

Можно вызывать файл сценария при запуске AutoCAD, или же можно выполнять сценарий изнутри AutoCAD, используя команду *SCRIPT*.

Файл сценария можно создать вне AutoCAD, используя любой текстовый редактор (типа Notepad Windows) или текстового процессора (типа Word) который сохраняет файл в текстовом формате. Расширение файла должно быть: *.scr.

Файлы сценария могут содержать комментарии, поясняющие его выполнение. Любая строка, которая начинается с точки с запятой, рассматривается как комментарий, и AutoCAD игнорирует эти строки при обработке файла сценария.

Когда ввод команды происходит из сценария, установки переменных системы *PICKADD* и *PICKAUTO* должны быть приняты 1 и 0 соответственно. Это сохраняет совместимость с предыдущими версиями AutoCAD и делает настройку проще.

При создании файла сценария необходимо учитывать, что пользователь может работать как с оригинальной, так и локализованной версией AutoCAD. Локализация программного обеспечения – процесс адаптации программного обеспечения к культуре какой-либо страны. В частности – перевод пользовательского интерфейса, документации и сопутствующих файлов программы с одного языка на другой.

Существует три уровня локализации:

1. Обеспечивающий поддержку языка и национальных стандартов – необходимый минимум, чтобы программа могла выполнять свои функции в другой стране.
2. Перевод текстов в интерфейсе программы на целевой язык. В сложном программном обеспечении не все части стоит переводить – например, многие не согласны с переводом имён функций Excel на русский язык. Чрезвычайно важен перевод терминологии. Например, спорным является применяемый в Windows термин “обозреватель”, обозначающий браузер.
3. Тонкая настройка под целевую страну. Сюда входит работа со словоформами. Примером будет пресловутое “Найдено 3 файлов”. К этому уровню также относятся дополнительные стандарты, не влияющие на основную функциональность программы, например, формат даты/времени.

Таким образом, локализация – это сложная и всеобъемлющая операция, и уже при разработке программного обеспечения соображения будущей интернационализации должны учитываться самым серьёзным образом. AutoCAD относится к программному обеспечению, русифицированному по первому–второму уровню. Для того чтобы файл сценария выполнялся в любой языковой версии, перед командами AutoCAD и параметрами команд ставится знак подчеркивания.

Все ссылки к длинным именам файла, которые содержат пробелы, должны быть заключены в двойные кавычки. Например, чтобы открыть рисунок my complex.dwg из сценария, необходимо использовать следующий синтаксис:

open "my complex"

Текущий сценарий завершается, когда вызывается другая команда *SCRIPT*. Для запуска созданного файла сценария из среды AutoCAD необходимо выбрать меню **Tools** пункт **Run Script...** и указать имя файла сценария.

Порядок выполнения работы

Выбрать эскиз детали согласно варианту (приложение А) по последней цифре номера зачетной книжки.

Задать размеры, определить характерные точки для построения чертежа.

Составить файл сценария, позволяющий построить чертеж.

Запустить файл сценария и получить необходимый чертеж.

Произвести оценку времени, затраченного на разработку файла сценария, создания чертежа в среде AutoCAD и построение чертежа при помощи файла сценария.

Содержание отчета

Цель работы.

Эскиз чертежа с координатами характерных точек.

Текст файла сценария с комментариями.

Описание используемых команд.

Выводы.

Контрольные вопросы

Какие способы построения отрезка в среде AutoCAD Вам известны?

Какие способы построения дуги в среде AutoCAD Вам известны?

Какие способы построения окружности в среде AutoCAD Вам известны?

Какими способами можно завершить команду *line* в среде AutoCAD?

Для чего используется знак подчеркивания перед командами AutoCAD и их параметрами?

Для каких задач автоматизации проектирования имеет смысл использовать файлы сценария?

2 ЛАБОРАТОРНАЯ РАБОТА №2: «ИЗУЧЕНИЕ РАБОТЫ ИНТЕРПРЕТАТОРА AUTOLISP ИЗ КОМАНДНОЙ СТРОКИ»

Цель работы

Научиться работать с выражениями AutoLISP из командной строки. Исследовать ситуации с неверным количеством скобок в выражениях. Рассмотреть особенности выполнения команд присвоения значений из файла.

Теоретический раздел

AutoLISP – это реализация языка программирования LISP, являющегося неотъемлемой частью пакета AutoCAD. С AutoLISP можно писать макропрограммы и функции на мощном языке высокого уровня, подходящем для графических приложений.

При вводе текста в командную строку AutoCAD интерпретирует текст, сравнивая символы по внутреннему списку допустимых имен команд. Когда AutoCAD получает некоторый код AutoLISP, он передает этот код интерпретатору AutoLISP. В ядре интерпретатора AutoLISP – вычислитель. Вычислитель читает строку программы, оценивает её и возвращает результат. Код должен быть в форме выражения AutoLISP и может читаться или из файла, или как ввод пользователя из командной строки AutoCAD. Все выражения AutoLISP имеют следующую форму:

(function параметры)

Каждое выражение начинается с левой круглой скобки и состоит из имени функции и факультативного списка параметров к этой функции, каждый из которых может быть самостоятельным выражением. Выражение заканчивается правой круглой скобкой. Каждое выражение возвращает значение, которое может использоваться окружающим выражением; если не имеется никакого окружающего выражения, AutoLISP возвращает значение в командную строку AutoCAD.

Если ввести выражение AutoLISP в командную строку AutoCAD, AutoLISP оценивает выражение и отображает результат, и командная строка появляется вновь. Следующий пример показывает использование функции * (умножение), которая принимает одно или несколько вещественных чисел как параметры.

command: (2 27)*
54

Так как этот пример кода не имеет никакого окружающего выражения, он возвращает результат в командную строку. Выражения, вложенные внутри других выражений, возвращают их результат к окружающему выражению. Следующий пример использует результат функции + (сложение) как один из параметров для функции * (умножение).

command: (2 (+ 5 10))*
30

Если Вы вводите неверное число правых круглых скобок, AutoLISP отображает следующую подсказку:

n>

где *n* – целое число, указывающее, сколько уровней левых круглых скобок остаются незакрытыми. Если появляется эта подсказка, нужно ввести n правых круглых скобок для оцениваемого выражения.

command: (2 (+ 5 10*
2>))
30

Особенности организации памяти в AutoLISP приводят к тому, что следует ясно различать два понятия: переменная и значение переменной.

Переменная – указатель на область динамической памяти, имеющий имя. Значение переменной – данные, записанные в динамической памяти начиная с адреса, записанного в переменной.

Изменить можно как переменную, так и ее значение. В первом случае та же переменная будет указывать на другой участок динамической памяти, а во втором – в том же участке динамической памяти будет помещено новое значение. Для присваивания значений переменным в AutoLISP имеются две функции – *SET* и *SETQ*.

Функция *SETQ* (set by quote) меняет значение переменной, а не саму переменную. Аргументами функции является перечень пар “переменная” – “значение”. Функция возвращает результат последнего присваивания. Например, запись *(SETQ a 10)* помещает число 10 в область памяти, на которую указывает переменная a и одновременно задает тип переменной *a* – целое число. Можно в одной функции присвоить несколько переменных:

(SETQ a 10 b "213")

Порядок выполнения нескольких присваиваний в функции *SETQ* определен слева направо. Если мы напишем:

(SETQ a 10 b (+ a 10))

то значением переменной *b* станет 20.

Изменение значения переменной на *NIL* освобождает занимаемую ее значением область памяти. *(SETQ a NIL)* сохраняет саму переменную *a*, но теперь она указывает “в никуда” (для ее значения памяти не выделено). *SETQ* – наиболее часто используемая функция присваивания. Она аналогична по-своему действия оператору присваивания.

Вторая функция присваивания – *SET* – работает не со значениями, а с самими переменными: *(SET a b)* заставляет переменную, *a* ссылаться на ту же область памяти, что и переменная *b*. Функция *QUOTE* запрещает вычисление списка, являющегося ее аргументом, и возвращает этот список. Например, написав:

(SETQ a (QUOTE (0.25 "АБВ" 46)))

и значением переменной, *a* станет список *(0.25 "АБВ" 46)*. Поскольку данные в программе встречаются весьма часто, введена сокращенная запись функции *QUOTE* в виде апострофа:

(SETQ a '(0.25 "АБВ" 46))

А что будет, если применить функцию *QUOTE* к переменной? Она вернет саму переменную, а не ее значение:

(SETQ a 5)
(SETQ b a); значением переменной b будет 5
(SETQ b 'a); значением переменной b будет переменная a.

Поэтому функцию *SET* можно использовать и для изменения значения переменной:

(SET 'a 10) *;делает то же, что и*
(SETQ a 10) *;поскольку запись 'a означает значение переменной a.*

Итак, значением одной переменной может быть другая переменная! Такой прием используется редко, но знать о его существовании следует.

Предположим теперь, что мы записали в значение переменной как данные список, который на самом деле является программой:

(SETQ a '(+ 5 10))

Как теперь заставить AutoLISP вычислить этот список? Для этого есть очень полезная функция *EVAL* (от *evaluation*). Функция *EVAL* вычисляет список, являющийся ее аргументом, и возвращает вычисленное значение. В нашем случае *(EVAL a)* вернет 15. Таким образом, функция *EVAL* обратная по отношению к функции *QUOTE*. Ее использование позволяет “на ходу”, в ходе выполнения программы, сформировать список–программу (содержащий вызовы функций) и исполнить его. В этом и заключается динамическая модификация текста программы, необходимая для решения задач искусственного интеллекта.

Теперь рассмотрим математические и логические функции AutoLISP, которые для удобства сведены в таблицу 2.1.

Таблица 2.1 – Математические и логические функции AutoLISP

Имя функции	Аргументы	Возвращаемое значение
+	*a1 a2 ... an*	*a1+a2+ ... +an*
–	*a1 a2 ... an*	*a1–a2– ... –an*
*	*a1 a2 ... an*	*a1·a2· ... ·an*
/	*a1 a2 ... an*	*a1/a2/ ... /an*
1+	*a*	*a+1*
1–	*a*	*a –1*
ABS	*a*	$\|a\|$
SQRT	*a*	Квадратный корень из *a*
EXP	*a*	exp(*a*)
EXPT	*a b*	a^b
GCD	*a b*	НОД чисел *a, b*
LOG	*a*	ln(*a*)

MIN	*a1 a2 ... an*	Минимальное из чисел *a1 a2 ... an*
MAX	*a1 a2 ... an*	Максимальное из чисел *a1 a2 ... an*
REM	*a b*	Остаток от деления a/b
SIN	*a*	sin(*a*), *a* – в радианах
COS	*a*	cos(*a*), *a* – в радианах
ATAN	*a*	arctg(*a*), *a* – в радианах

В языке также предусмотрена встроенная переменная *PI*, со значением, равным числу “пи”.

Пример: вычислить площадь треугольника S со сторонами a, b, c по формуле Герона. Запишем программу вычислений:

```
(SETQ p (/ (+ a b c) 2))
(SETQ s (SQRT (* p (– p a ) (– p b)
 (– p c))))
```

Для того, чтобы эту программу выполнить, ее можно либо построчно набирать в ответ на приглашение AutoCAD “c*ommand*:”, либо записать в текстовый файл с расширением *.lsp и загрузить его. Функции будут выполнены в процессе загрузки файла.

Внимание! AutoCAD не загружает автоматически файл с программой после того, как в него внесены изменения. Это надо делать принудительно при помощи меню или функции *LOAD*. В противном случае можно долго искать ошибку: программа исправлена, а работает, как и прежде.

Чтобы просмотреть значение переменной, надо в командной строке набрать восклицательный знак, а за ним – имя переменной:

```
command: !a
```

выведет на экран значение переменной *a*. При наборе программ на AutoLISP легко ошибиться с парностью скобок, поэтому рекомендуется использовать текстовый редактор, отслеживающий скобки.

Порядок выполнения работы

Выбрать задание из таблицы. Вариант задания определяется согласно последней цифре номера зачетной книжки.

Вариант	Задание
0	Перевод градусов в радианы
1	Перевод радиан в градусы
2	Вычисление тангенса числа
3	Вычисление котангенса числа
4	Перевод дюймов в милиметры
5	Перевод миллиметров в дюймы
6	Перевод секунд в градусы
7	Перевод градусов в секуны
8	Перевод пунктов в миллиметры
9	Перевод об/мин в рад/с

Составить формулы для решения задачи.
Решить задачу из командной строки.
Создать файл с выражениями AutoLISP и выполнить его из AutoCAD.

Содержание отчета

Цель работы.
Формулы для решения поставленной задачи.
Текст файла с выражениями AutoLISP.
Описание используемых команд.
Выводы.

Контрольные вопросы

Какую форму должно иметь любое выражение?
В чем различие между функциями *SET* и *SETQ*?
Как загрузить программу AutoLISP в AutoCAD?
О чем говорит появление сообщения *n*> в командной строке?
Как вывести значение переменной на экран?

3 ЛАБОРАТОРНАЯ РАБОТА №3: «ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ ПОЛЬЗОВАТЕЛЯ ДЛЯ ПОСТРОЕНИЯ ЧЕРТЕЖА»

Цель работы

Научиться создавать и использовать функции пользователя в AutoLISP. Исследовать работу функции *COMMAND* для реализации построения чертежа в среде AutoCAD.

Теоретический раздел

По сути дела, программирование на AutoLISP сводится к написанию набора пользовательских функций, выполняющих определенные действия. Головная функция должна явно вызываться тем или иным способом, которые рассмотрены ниже. Стандартный (но не единственный!) способ создания пользовательской функции заключается в использовании функции *DEFUN* (DEFine FUNction). В общем виде она записывается следующим образом:

(DEFUN name (a1 a2 ... an / v1 v2 ... vm)
(выражение 1)
(выражение 2)
....
(выражение N))

где *name* – имя функции; *ai* – i-й аргумент функции; *vi* – i-я локальная переменная.

Функция *DEFUN* создает в памяти пользовательскую функцию с именем name и списком аргументов *a1 a2 ... an*. При этом сама функция name еще не выполняется, а только помещается в память – для выполнения она должна быть явно вызвана.

Пользовательские функции не могут иметь произвольное число аргументов, как встроенные функции типа “+”, “–“ и др. Вызов пользовательской функции с неверным числом аргументов приведет к возникновению ошибки “Неверное число аргументов”.

Поскольку пользовательская функция, как и любая другая, обязательно возвращает значение, возникает вопрос: как определить, что же она вернет? Ведь внутри функции, как правило, большое количество списков–программ. Воспользуемся следующим правилом нахождения возвращаемого пользовательской функцией значения:

- найти закрывающую скобку, парную открывающей перед словом *DEFUN* (т.е. последнюю);
- найти предпоследнюю закрывающую скобку;
- найти парную ей открывающую скобку;
- значение, возвращаемое функцией, стоящей после этой открывающей скобки, и будет возвращаемым значением всей пользовательской функции.

Иначе говоря, возвращаемое значение есть результат вычисления последнего списка, находящего внутри *DEFUN*.

AutoLISPу все равно, в какой последовательности в тексте lsp–файла идут описания функций *DEFUN* – ведь перед выполнением программы все они сначала загружаются в память. Поэтому при написании программы последовательность описания функций не важна (в отличие от Паскаля, где каждая вызываемая процедура или функция должна быть уже определена).

Это совершенно корректная ситуация, не вызывающая у AutoLISP вопросов. Паскаль бы в такой ситуации стал сильно ругаться.

А теперь попробуем наконец-то создать собственную полезную функцию.

Пример: в AutoLISP нет функции, вычисляющей тангенс угла. Введем функцию с именем *tan* следующего вида:

```
(DEFUN tan( a )
 ( / ( SIN a ) ( COS a ) ) )
```

Текст пользовательской функции можно вывести на экран при помощи "!", поскольку он также является значением переменной. Закономерный вопрос: а куда делись косая черта и загадочные "локальные переменные" *v1, v2, ... vm*?

Все переменные в AutoLISP делятся на два вида: глобальные и локальные. Глобальные переменные постоянно находятся в оперативной памяти и, следовательно, доступны из любой функции. Глобальные переменные создаются автоматически при присваивании им значения. Например, функция *(SETQ a 5)* создает глобальную переменную *a*.

Локальные переменные явно описываются в заголовке функции *DEFUN* и видны только внутри соответствующей пользовательской функции. В начале работы пользовательской функции локальные переменные имеют значение *NIL*. По окончании работы этой функции они автоматически удаляются из памяти.

Например, опишем функцию с именем *s* следующего вида:

```
(DEFUN s ( a / p )
    ( SETQ p 5 ) )
( SETQ p "ABC" )
```

Здесь *p*, которой присваивается значение 5 – локальная переменная. Она не будет доступна из других функций. Поэтому функция *(SETQ p "ABC")* создает глобальную переменную с именем *p*, никаким образом не связанную с локальной переменной *p*.

Аргументы пользовательских функций также являются локальными переменными. В начале работы пользовательской функции аргументы принимают значения, переданные функции при ее вызове.

Использование глобальных переменных крайне нежелательно. Они занимают память; вызывают необходимость отслеживания имен переменных (а локальных переменных с одинаковыми именами в разных пользовательских функциях может быть сколько угодно). В любом языке программирования использование глобальных переменных – признак низкой квалификации программиста. Данные следует хранить в локальных переменных и передавать через параметры вызываемых функций.

Если глобальные переменные все же используются, как только необходимость в них отпадает, следует освобождать занимаемую ими память, присваивая им значение *NIL* (вопрос нехватки памяти в AutoLISP в его реализации для AutoCAD стоит очень остро).

Итак, пользовательская функция может не иметь как локальных переменных, так и аргументов. Поэтому возможны записи:

(DEFUN s (a b / cd)
(DEFUN s (a b)
(DEFUN s (/ c d)
(DEFUN s ())

В любом случае пара скобок после имени функции сохраняется. Попробуем теперь написать более сложную функцию.

Пример: написать функцию вычисления площади треугольника со сторонами a, b, c.

Очевидно, снова нужно воспользоваться формулой Герона. Назовем нашу функцию *trisqu* (triangle square). Ее аргументами, очевидно, будут стороны треугольника. Введем также локальную переменную p для записи полупериметра:

```
(DEFUN trisqu( a b c / p)
    (SETQ p( / (+ a b c) 2)
       (SQRT (* p ( - p a) (- p b ) (- p c))))
```

Обратите внимание, что нет необходимости записывать площадь в переменную – мы сразу сделали ее возвращаемым значением. При вызове функции *trisqu* ей обязательно должны быть переданы три аргумента, значения которых запишутся в локальные переменные *a, b, c*, т.е. мы должны написать:

(trisque 3 4 5) или *(trisque a 4.5 (/ 2 b))*

Рассмотрим теперь способы занесения пользовательских функций в память и их вызова. Как мы уже знаем, наиболее удобный путь – записать пользовательские функции в текстовый файл с расширением *.lsp и загрузить его функцией LOAD, которую можно набрать в командной строке AutoCAD:

(LOAD "имя_файла")

Функция *LOAD* при ошибке загрузки возвращает "имя_файла", при нормальной загрузке – ссылку на последнюю определенную в загруженном файле пользовательскую функцию (имя этой функции выводится на экран).

Задание имени файла сопряжено с тремя "подводными камнями":

Имя файла либо является переменной типа "текст', либо выражением, вычисление которого дает результат текстового типа. Если имя задается явно, как константа, оно заключается в кавычки.

Расширение имени файла и точка перед ним не указываются. AutoLISP сам добавляет расширение *.lsp. Поэтому программу можно записывать в файл только с таким расширением, иначе ее будет невозможно загрузить.

Если указывается полный путь доступа к файлу, то символ "\" нужно заменить символом "/" и вместо "Y:\STUDENT\3.LSP" написать "Y:/

STUDENT/3.LSP". Это связано с тем, что AutoLISP воспринимает символ "\" как управляющий код.

А как же теперь вызвать нашу функцию на исполнение? Принципиально возможны три способа:

- вызов функции с командной строки;
- вызов функции из меню AutoCAD (вызов функции из меню требует знания правил его модификации и не может быть рекомендован неопытным пользователям);
- введение новой команды в AutoCAD.

Первый способ очевиден: после загрузки файла, содержащего, к примеру, функцию вычисления тангенса *tan* мы просто наберем в командной строке *(tan 0.25)*, чтобы узнать тангенс угла в 0.25 радиан. Однако скоро набирать скобки надоест.

Если имя пользовательской функции начинается с *"C:"*, то после ее загрузки (и до окончания сеанса работы или до явного удаления функции из памяти) в AutoCAD появляется новая команда с именем, совпадающим с именем функции. При этом пользовательская функция не может иметь аргументов.

Главная цель применения языка AutoLISP – работа с графическим экраном, что позволяет автоматизировать построение изображений. Но для этого нужно узнать, каким образом AutoLISP связан с командами AutoCAD.

Обычно пользователи AutoCAD, даже достаточно опытные, умеют работать только с использованием верхнего (падающего) или бокового (экранного) меню. Предложение нарисовать чертеж на компьютере без мыши поставит их в тупик. Между тем первые версии AutoCAD появились еще в эпоху машин типа IBM PC и IBM PC/XT, когда никаких мышей и в помине не было. Как же работать с AutoCAD с клавиатуры?

AutoCAD имеет встроенный набор команд, которые вводятся с клавиатуры в ответ на приглашение "*command*:" и в большинстве запрашивают определенные параметры. Последовательность ввода этих параметров определяется форматом команды. Набор команд в пределах одной версии AutoCAD постоянен и одинаков, а меню можно создавать самому, поэтому в разных организациях меню обычно сильно отличаются.

В качестве примера рассмотрим формат команды *pline* ("*ПЛИНИЯ*") как наиболее широко используемой в построениях на графическом экране (см. таблицу 3.1).

Таблица 3.1 – Формат команды pline при интерактивном создании чертежа в среде AutoCAD

Что вводим	Что это такое мы ввели
Command: pline	Имя команды. С этого начинается выполнение любой команды AutoCAD
From point:	Координаты точки в виде: 1. x,y например, 100,250 – абсолютные координаты точки; 2. @dx,dy например, @–10,45 – приращения по осям координат от текущей точки; 3. @r<a например, @10<90 – перемещение от текущей точки под

	углом a на расстояние r.
Current line-width is 0.0000 Arc/Close/Halfwidth/ Length/Undo/Width/:	Здесь вводится или следующая точка, или ключевое слово – одно из слов в приводимом списке, разделенных символом "/". Можно вводить не слово целиком, а только ту его часть, которая набрана заглавными буквами
Command:	Для явного завершения рисования полилинии команду необходимо прервать нажатием клавиш Esc, Enter или нажатием правой кнопки мыши

Итак, в AutoCAD возможен ввод следующих типов параметров команд с клавиатуры:

- текстовые строки (например, название команды или имя файла);
- координаты точек;
- выбор объекта;
- численные значения (например, номер цвета);
- ключевые слова;
- прерывание выполнения команды.

Большинство команд AutoCAD могут быть выполнены из программы на AutoLISP при помощи функции *COMMAND*:

(COMMAND t1 t2 .. tn)

где *t1* – имя вызываемой команды; *t2 ... tn* – параметры вызываемой команды.

Чтобы не пытаться создавать принципиально невозможные программы, сразу следует отметить: из программы на AutoLISP нельзя вызвать следующие команды: *DTEXT*, *SKETCH*, *PRINT*, *PLOT*, *SCRIPT*, а также команды, определенные пользователем при помощи *(RUN C:)*.

Особенно печально то, что невозможно автоматизировать вывод чертежа на бумагу (команды *PRINT* и *PLOT*), что делает любые системы автоматизации конструкторского труда, написанные на AutoLISP, несколько неполноценными.

Есть два особых вида выражений, которые могут быть аргументами функции *COMMAND*: *PAUSE* – позволяет пользователю ввести соответствующий параметр вручную; “ ” (две кавычки) или отсутствие параметров вообще равносильно прерыванию команды.

Пример: нарисуем из программы на AutoLISP квадрат с левым нижним углом в точке (10,10) и стороной 25мм.

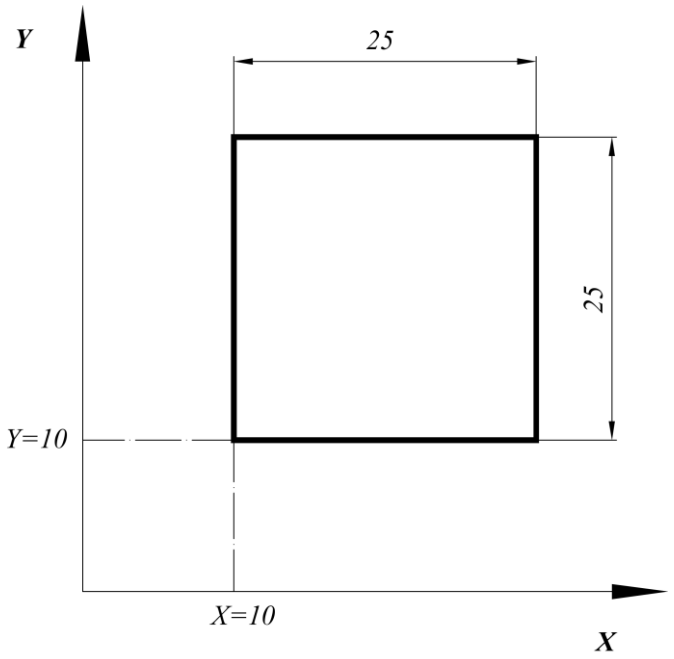


Рисунок 3.1 – Эскиз квадрата с базовой точкой

Если бы мы пользовались только клавиатурой, то диалог выглядел бы следующим образом:

Command: pline
From point: 10,10
Current line-width is 0.0000
Arc/Close/Halfwidth/Length/Undo/Width/: @25,0
Arc/Close/Halfwidth/Length/Undo/Width/: @0,25
Arc/Close/Halfwidth/Length/Undo/Width/: @-25,0
Arc/Close/Halfwidth/Length/Undo/Width/: C

На AutoLISP это будет выглядеть так:

(COMMAND "pline" "10,10" "@25,0" "@0,25" "@-25,0" "C")

Все константы, являющиеся параметрами функции *COMMAND*, задаются как текстовые строки, даже если они являются числами или координатами точек. Однако главное свойство функции *COMMAND* – возможность подстановки в качестве параметров результатов выполнения программ. Любой параметр функции *COMMAND* можно заменить именем переменной или выражением AutoLISP. Данный параметр примет значение, равное значению переменной или результату вычисления выражения.

Порядок выполнения работы

Выбрать эскиз детали из приложения А по последней цифре номера зачетной книжки.

Задать размеры детали, определить характерные точки для построения чертежа.

Создать две функции пользователя. Первая должна задавать ряд размеров детали, вторая – производить построение чертежа.

Созданный файл загрузить в AutoCAD

Вызвать первую функцию пользователя с заданием параметров детали.

Вызвать вторую функцию пользователя для построения чертежа.

Содержание отчета

Цель работы.

Эскиз детали с характерными точками.

Формулы для определения всех точек, необходимых для построения.

Текст программы с комментариями.

Описание используемых команд.

Выводы.

Контрольные вопросы

Какой формат имеет функция пользователя?

В чем различие между глобальными и локальными переменными?

Как задать функцию пользователя с локальными переменными?

Как вызвать функцию пользователя с параметрами?

Если имя пользовательской функции начинается с “*C:*”, какие возможности получает проетировщик?

Какие команды AutoCAD невозможно вызвать из функции пользователя?

Как вызвать команду AutoCAD из файла AutoLISP?

4 ЛАБОРАТОРНАЯ РАБОТА №4: «ИССЛЕДОВАНИЕ СПОСОБОВ ВВОДА И КОНТРОЛЯ ВВОДИМОЙ ИНФОРМАЦИИ ДАННЫХ В ДИАЛОГОВОМ РЕЖИМЕ ПРИ ПАРАМЕТРИЧЕСКОМ ПРОЕКТИРОВАНИИ»

Цель работы

Научиться создавать и использовать диалог с пользователем в функциях AutoLISP. Исследовать работу функций ввода данных для реализации параметрического построения чертежа в среде AutoCAD. Изучить и исследовать работу функций контроля вводимой информации.

Теоретический раздел

Как говорилось в предыдущей работе, если имя пользовательской функции начинается с “*C:*”, (что позволяет вызывать эту функцию как команду AutoCAD), пользовательская функция не может иметь аргументов.

Как же тогда передавать нашей функции информацию? Интерактивно: запрашивать ее у пользователя. Для этого в AutoLISP есть набор функций, обеспечивающих ввод–вывод информации.

Начнем с вывода. Экран AutoCAD может работать в двух режимах: текстовом и графическом (в версии 14 для вызова текстового окна нужно нажать клавишу F2). Эти режимы переключаются функциями AutoCAD *TEXTSCR* и *GRAPHSCR*.

Сначала поговорим о выводе на текстовый экран. Для этого предназначены следующие функции, представленные в таблице.

Таблица 4.1 – Функции вывода информации на текстовый экран

Наименование	Аргументы	Описание
PRINC	Любое выражение	Вывод выражения на экран без учета управляющих кодов
PRIN1	Любое выражение	Вывод выражения на экран с учетом управляющих кодов
PRINT	Любое выражение	Вывод выражения на экран с учетом управляющих кодов, с новой строки и с пробелом в конце
PROMPT	Текст	Вывод текста на экран с учетом управляющих кодов
TERPRI	Нет	Вывод пустой строки

AutoLISP “понимает” следующие управляющие коды в выводимых на экран текстовых строках (см. таблицу 4.2):

Таблица 4.2 – Управляющие коды строк

Код	Значение
\e	Символ с кодом 27 (ESC)
\n	Переход на новую строку
\r	Переход в начало той же строки
\t	Переход на следующую позиции табуляции (8 пробелов)
\nnn	Ввод символа с восьмеричным кодом nnn

Если нужно просто вывести на печать символ “\”, его нужно удвоить, т.е. написать “\\”.

Пример: для вывода классической фразы “Hello, world!” с новой строки в AutoLISP следует написать:

(PROMPT “\nHello, world!”)

Функция *PRIN1* не возвращает значения и удобна для “тихого” завершения работы головной программы. Для вывода на экран цветного текста, управления положением курсора и т.д. можно использовать драйвер экрана и клавиатуры ANSI.SYS, который должен быть установлен на компьютере и которым можно управлять при помощи ESC–последовательностей (для этого и нужен управляющий код “\e”). Ввод данных осуществляется семейством *GET*–функций (см. таблицу 4.3).

Таблица 4.2 – *GET*–функции для ввода данных

Наименование	Аргументы	Описание
GETINT	Текст подсказки	Ввод целого числа
GETREAL	Текст подсказки	Ввод вещественного числа
GETSTRING	Флаг пробела (*T* или *NIL*) Текст подсказки	Ввод текста. Если флаг пробела = *T*, в тексте могут быть пробелы, иначе (по умолчанию) пробел воспринимается как окончание ввода
GETPOINT	Текст подсказки	Ввод координат точки с клавиатуры или указанием точки мышью

Все *GET*–функции возвращают введенное с клавиатуры значение или *NIL*, если имел место пустой ввод (сразу нажата клавиша Enter). Для сохранения этого значения его следует записать в переменную, например:

(SETQ (GETINT “\nВведите X:”))
(SETQ m (GETSTRING T “\nВведите фамилию и имя:”))

Ввод данных всегда чреват возникновением ошибок. Как известно, в большинстве случаев на подсказку “Введите целое число от 1 до 3” пользователь пишет “Вася” и злорадно ожидает реакции компьютера. Для автоматического исключения наиболее очевидных ошибочных ситуаций предназначена функция *INITGET*:

(INITGET сумма_кодов)

действие которой распространяется только на одну следующую за ней *GET*–функцию. Возможны следующие значения кодов (см. таблицу):

Таблица 4.3 – Значения кодов функции *INITGET*

Код	Значение
1	Запретить пустой ввод
2	Запретить ввод нуля
4	Запретить ввод отрицательных чисел

Пример: пусть с клавиатуры требуется ввести номер позиции детали в спецификации и записать его в переменную *p*. Очевидно, что номер позиции – целое положительное число. Делается это так:

(INITGET 7)

(SETQ p (GETINT "\nВведите номер позиции детали:"))

Пример: написать интерактивную программу вычисления площади треугольника.

```
(DEFUN C:trisque ( / a b c p s)
      (TEXTSCR)
      (INITGET 7)
      (SETQ a (GETREAL "\nДлина стороны A:"))
      (INITGET 7)
      (SETQ b (GETREAL "\nДлина стороны B:"))
      (INITGET 7)
      (SETQ c (GETREAL "\nДлина стороны C:"))
      (SETQ p (/ (+ a b c) 2)
      (PROMPT "\nПлощадь треугольника равна ")
      (PRIN1 (SQRT (* p (- p a) (- p b) (- p c))))
   (PRIN1)
)
```

Итак, в AutoCAD возможен ввод следующих типов параметров команд с клавиатуры:

- текстовые строки (например, название команды или имя файла);
- координаты точек;
- выбор объекта;
- численные значения (например, номер цвета);
- ключевые слова;
- прерывание выполнения команды.

Как получить средствами AutoLISP текстовую строку, в общем, понятно. Однако возникает естественный вопрос: как представить координату точки?

Координаты точек являются списками из двух или трех вещественных чисел – координат по осям X, Y и Z соответственно.

Таким образом, точка с координатами 10,10 может быть задана как текстовой строкой "10,10", так и списком: *(LIST 10 10)*. Второй способ позволяет использовать переменные и выражения AutoLISP для указания координат.

Пример: если координата X точки записана в переменной *A*, а координата Y равна $(A+20)^2/4$, то следует записать:

(LIST A (/ ((+ A 20) (+ A 20)) 4))*

В принципе имеющихся в AutoLISP математических функций достаточно, чтобы выполнять геометрические построения. Однако для удобства язык имеет ряд специальных встроенных функций для вычисления координат точек.

Основная геометрическая функция – *POLAR*:

(POLAR a angle dist)

где *a* – список из двух элементов (координаты точки); *angle* – угол в радианах; *dist* – расстояние в текущих единицах измерения.

POLAR возвращает в виде списка координаты точки, отстоящей от точки *a* на расстояние *dist* под углом *angle*.

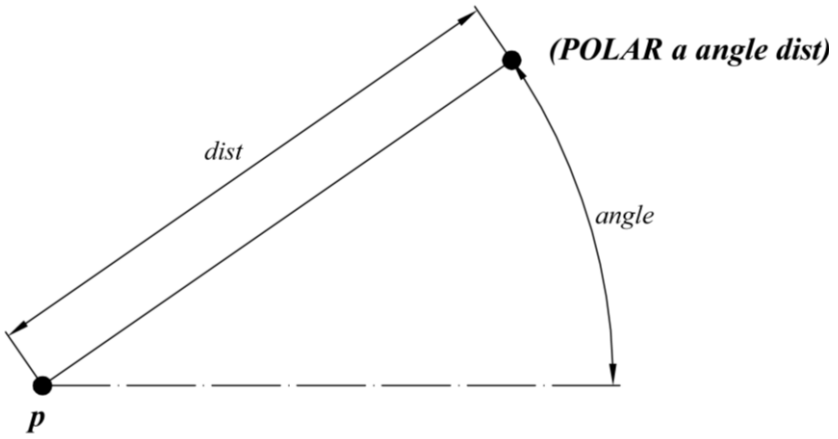


Рисунок 4.1 – Результат применения функции *POLAR*

Следует вспомнить, что положительное направление отсчета углов в системе AutoCAD – против часовой стрелки.

Единицы измерения, как и многие другие параметры, определяются значениями системных переменных AutoCAD. Системная переменная – это ячейка памяти, содержащая определенное значение и имеющая неизменное имя. Значения системных переменных задают различные режимы работы команд AutoCAD.

Не путайте переменные AutoLISP и системные переменные AutoCAD – это абсолютно разные вещи! К системным переменным нельзя обращаться напрямую, как к обычным переменным AutoLISP. Для доступа к системным переменным в AutoLISP имеются две функции:

(GETVAR “имя”)
(SETVAR “имя” значение)

Функция *GETVAR* возвращает значение системной переменной с именем “имя”, заданным как текстовая строка. Например, системная переменная “*LASTPOINT*” содержит координаты текущей точки. Для их использования в программе следует использовать функцию *GETVAR* в виде:

(GETVAR “LASTPOINT”)

Если в ходе отрисовки полилинии следующую точку удобнее рассчитать от предыдущей при помощи функции *POLAR*, необязательно записывать все промежуточные точки в переменные. Можно использовать функцию *GETVAR*, например:

(COMMAND "PLINE" (LIST (+ A 10) (– B 20))
(POLAR (GETVAR "LASTPOINT") 0 40 “”)

В приведенном примере координаты начальной точки рассчитываются. Чтобы не записывать эту точку в отдельную переменную (вопрос нехватки памяти в AutoLISP стоит очень остро), следующая точка, координаты которой рассчитывается при помощи функции *POLAR*, использует в качестве опорной

координаты текущей (т.е. начальной) точки, всегда записываемые в виде списка в системную переменную "*LASTPOINT*".

Функция *SETVAR* меняет значение соответствующей системной переменной.

Осторожно! Хорошенько подумайте, прежде чем менять значение системной переменной. Эти значения записываются в файл чертежа. Часть системных переменных (например, переменная, содержащая номер версии AutoCAD) доступна только для чтения и их значения нельзя изменить.

При геометрических расчетах используются также следующие функции:

(INTERS t1 t2 t3 t4 признак)

возвращает точку пересечения двух отрезков, проходящих через точки *t1, t2, t3* и *t4* соответственно. Признак показывает, следует ли находить точку пересечения бесконечных прямых, проходящих через точки *t1, t2, t3* и *t4* (если признак=*NIL*) или же только отрезков (если признак не равен *NIL*). Если точка пересечения отсутствует, функция возвращает *NIL*.

Функция *(ANGLE t1 t2)* возвращает угол в радианах между положительным направлением оси X и прямой, проходящей через точки *t1* и *t2*.

Функция *(DISTANCE t1 t2)* возвращает расстояние от точки *t1* до точки *t2* в текущих единицах измерения расстояний.

Поскольку список – главное действующее лицо языка LISP, следует тщательно рассмотреть набор функций по работе с ними.

До некоторой степени список аналогичен массиву в языках типа Pascal. Там доступ к конкретному элементу списка решается просто – указывается его номер: a[5]. В LISP все несколько сложнее.

Функция *(CAR mylist)* возвращает первый элемент списка *mylist*. Например, *(CAR (LIST 10 20))* вернет 10.

Если список *mylist* является описанием координат точки, то *(CAR mylist*) возвращает координату X. Название функции идет от первой, 1958 года, реализации языка LISP на древнем компьютере, в работе которого немаловажную роль играл адресный регистр памяти (Contents of Address Register), сокращенно CAR. Создатель LISP Дж. Маккарти назвал в честь этого регистра одну из функций языка LISP.

Функция *(CDR mylist)* возвращает все элементы списка mylist, кроме первого. Иначе говоря, у списка отрывается "голова", а возвращается остающийся "хвост", причем, даже если этот "хвост" длиной в один атом, он все равно будет списком:

(CDR (LIST 10 20)) ; возвращает (20)

Названа функция также в честь одного из регистров древнего компьютера – Contents of Decrement Register.

Функция *CDR* не годится для получения координаты Y точки – она возвращает список, а координата, конечно же, должна быть выражена атомом - вещественным числом. По-хорошему надо из списка, возвращаемого функцией *CDR*, выделить первый элемент, написав

(SETQ p (LIST 10 20)) ; координаты точки p
(SETQ y (CAR (CDR p)))

Но такая запись выглядит весьма громоздко. Поэтому в AutoLISP предусмотрена возможность использования вложенных функций *CAR* и *CDR*, которые будут называться соответственно *CADR*, *CDAR*, *CAAR*, *CDDR* и так далее (до четырех уровней вложенности). При этом *(CADR mylist)* эквивалента *(CAR (CDR mylist))*.

Последний элемент списка как атом возвращает функция *(LAST mylist)*. В принципе ее можно использовать для получения координаты Y, но где гарантия, что в один прекрасный день пользователь не включит режим использования трехмерных точек? Тогда *LAST* станет возвращать уже координату Z и ваша программа тут же потеряет работоспособность.

И, наконец, самая общая функция выделения элементов из списка: *(NTH n mylisy)*, которая возвращает *n*–й элемент списка *mylist*. Название функции происходит от английского окончания порядковых числительных -th.

Нумерация элементов списка в функции *NTH* начинается с нуля!

Хуже того, в AutoLISP нет единообразия в этом вопросе: часть функций все же уверена, что нумерация начинается с единицы. Такая путаница – один из наиболее крупных “проколов” AutoLISP.

Итак, поскольку наиболее часто нам нужно выделить отдельные координаты точки, подытожим, как это легче всего сделать (см. таблицу 4.4).

Таблица 4.4 – Функции получения координат точек

Точка p	
Координата X	Координата Y
(CAR p)	(CADR p)
(NTH 0 p)	(NTH 1 p)

Разбор точек на координаты используется, если необходимо рассчитать положение точки через приращения. Например, при отрисовке фаски нужно узнать координаты точки P2, зная точку P1:

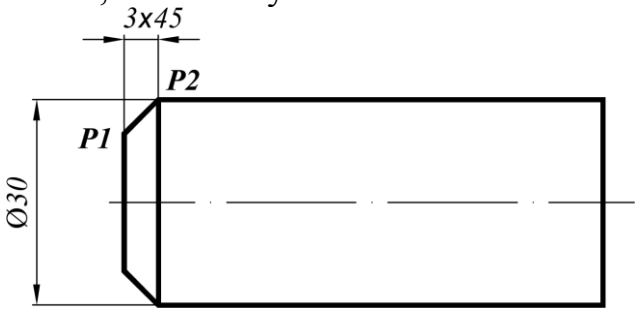


Рисунок 4.2 – Вал с фаской

Хотя угол здесь задан в явном виде, использование функции *POLAR* будет некорректным: мы не знаем расстояние *P1P2*.

Если его высчитывать как гипотенузу прямоугольного треугольника, точность вычисления квадратного корня окажется ограниченной, и мы не попадем в точку *P2*. Если бы мы рисовали фаску вручную, то мы бы ввели прира-

щения по осям в виде @3,3 (т.е. переместиться от точки *P1* вправо на 3мм и верх на 3 мм). Сделаем то же самое в AutoLISP:

(SETQ p2 (LIST (+ (CAR p1) 3)
(+ (CADR p1) 3)))

Такая запись является стандартным способом расчета координат точек в приращениях. При использовании списков, особенно списков-данных, часто необходимо добавлять в имеющийся список новые и новые значения, как бы "приклеивая" их к его хвосту. Для этого предназначена функция *(APPEND list1 list2)*, которая добавляет в конец списка *list1* список *list2* и возвращает новый, удлиненный список. Список *list1* от этого автоматически не меняется, нужно сохранять результат выполнения функции *APPEND.*

Обратите внимание: для добавления в список атома его сначала нужно превратить список из одного элемента!

Пример: в переменной *a* хранится список вида (19 49). К нему нужно добавить число 20. Делается это так:

(SETQ a (APPEND a (LIST 20)))

Порядок выполнения работы

Выбрать эскиз детали из предыдущей работы.

Создать четыре функции пользователя. Первая должна организовать диалог для ввода параметров детали, вторая – произвести вычисления для определения всех необходимых точек и размеров детали, третья – построить чертеж.

Вызов этих функций последовательно осуществить из главной функции. Имя главной функции должно начинаться с "*C:*".

Созданный файл загрузить в AutoCAD. Задать недопустимые параметры (отрицательные, нулевые и т.п.). Убедиться, что программа требует ввести корректные значения.

Содержание отчета

Цель работы.

Эскиз детали с характерными точками.

Формулы определения точек и параметров, необходимых для построения чертежа.

Текст программы с комментариями.

Описание используемых команд.

Выводы.

Контрольные вопросы

Как запретить пользователю ввод отрицательных чисел и нуля?

Какие способы получения координат из точки Вам известны?

Как получить координату Z из трехмерной точки?

Какую точку следует выбирать в качестве базовой?

5 ЛАБОРАТОРНАЯ РАБОТА № 5: «ИСПОЛЬЗОВАНИЕ КОМАНД ВЕТВЛЕНИЯ ДЛЯ ПРОВЕРКИ КОРРЕКТНОСТИ ВВЕДЕННЫХ ПАРАМЕТРОВ»

Цель работы

Изучить команды ветвления и их использование в функциях пользователя в AutoLISP. Исследовать работу функций ветвления для проверки введенных данных при реализации параметрического построения чертежа в среде AutoCAD.

Теоретический раздел

Работа со списками нужна нам не сама по себе, а для решения той или иной задачи. А любая мало-мальски сложная задача, в свою очередь, требует для своего решения применения ветвлений и циклов. Поэтому сейчас мы начнем рассматривать управляющие конструкции AutoLISP – ветвления и циклы.

И ветвление, и цикл в обязательном порядке содержат проверку условия. В качестве условий в LISP используются логические функции, возвращающие *T* (true – истина) или *NIL* (ложь): “<”, “>”, “<=”, “>=”, “=”, “/=”. Обратите внимание: операция “не равно” записывается не так, как в большинстве других языков программирования!

Функции сравнения могут применяться к целым и вещественным числам, а также текстовым строкам, но не к спискам. Примеры:

```
(= 2 2)            ; возвращает T
(= 2 5)            ; возвращает NIL
(= "ABC" "AB")     ; возвращает NIL
```

Если мы сравниваем вещественные числа, то следует помнить об ограниченной точности вычислений с ними. Особенно это относится к тригонометрическим функциям. Поэтому крайне нежелательно сравнивать результат тригонометрической функции с константой, иначе возникают трудноуловимые казусы следующего вида:

```
(SETQ a1 (SIN 0.0))
(SETQ a2 (SIN (* 2.0 PI)))
```

Тогда *(= a1 a2)* возвращает *NIL*, т.е. с точки зрения AutoLISP, поскольку не равен точно нулю.

Что же делать, если нужно сравнить два вещественных числа, когда хотя бы одно из них – результат вычисления тригонометрической или иной функции? (заметим, что аналогичная проблема в других языках программирования просто обходится молчанием). Создатели LISP ввели в язык специальную функцию сравнения с заданной точностью:

```
(EQUAL e1 e2 точность)
```

Здесь точность – число (0.1, 0.01..), указывающее, сколько знаков поле запятой принимается во внимание при сравнении выражений *e1* и *e2*. Поэтому функция

(EQUAL (SIN 0) (SIN (PI 2)) 0.1)*

вернет *T*.

Функции сравнения могут объединяться при помощи логических функций, образуя сложные условия. В AutoLISP имеется богатый набор логических функций (заметно больший, чем в других языках), из которых реально достаточно знать следующие четыре:

Таблица 5.1 – Логические функции AutoLISP

X	Y	X *AND* Y	X *OR* Y	X *XOR* Y	*NOT* X
T	*NIL*	*NIL*	*T*	*T*	*NIL*
T	*T*	*T*	*T*	*NIL*	*NIL*
NIL	*T*	*NIL*	*T*	*T*	*T*
NIL	*NIL*	*NIL*	*NIL*	*NIL*	*T*

Запомнить эти функции легко: *AND* – и X, и Y истинны, тогда X *AND* Y будет истинно; *OR* – или X, или Y, или X и Y сразу истинны, тогда X *OR* Y будет истинно; *XOR* – или X, или Y истинны по отдельности, но не оба сразу, тогда X *XOR* Y будет истинно; *NOT* – просто “переворачивает”, инвертирует значение, превращая *T* в *NIL*, а *NIL* в *T*.

Пример: если в переменных *a, b, c* хранятся длины сторон треугольника, то весьма желательно, чтобы они все были больше нуля. Тогда получим сложное условие:

(AND (> a 0) (> b 0) (> c 0))

Теперь мы готовы к рассмотрению функции IF, обеспечивающей ветвление в программе. Ее общий вид:

```
(IF c
    f1
       [f2] )
```

Здесь *c* – условие (простое или сложное), *f1* – функция, выполняемая, если условие истинно (часть “то”), а *f2* – функция, выполняемая, когда условие ложно (часть “иначе”), причем квадратные скобки говорят о том, что часть “иначе” может отсутствовать.

Простейший пример:

```
(IF (< a 0)
    (PROMPT "\nПеременная a меньше нуля")
       (PROMPT "\n"Переменная a больше или рана нулю"))
```

А как быть, если нужно в случае выполнения (или невыполнения) условия выполнить не одну, а сразу несколько функций? Ведь синтаксис функции *IF* разрешает записать только одну. Проблема решается так же, как в языке Паскаль: там используются операторные скобки begin ... end, а в AutoLISP – функция *(PROGN f1 f2 ... fn)*. Она всего лишь объединяет функции *f1 f2 ... fn* в один блок, который можно подставить в функцию *IF*. Например, мы решаем

квадратное уравнение и в переменную *d* записали дискриминант. Теперь нужно посчитать действительные корни и вывести их на экран:

```
(IF (>= d 0)
    (PROGN
        (SETQ x1 (/ + (* b -1) (SQRT d))
        (* 2 a))
        (SETQ x2 (/ (- (* b -1) (SQRT d))
        (* 2 a))
    (PROMPT "\nX1=")
    (PRINT x1)
    (PROMPT "\nX2=")
    (PRINT x2))
       (PROMPT "\nДействительных корней нет"))
```

Продолжая рассматривать параллели с Паскалем, обратимся к функции *COND*, обеспечивающей множественное ветвление аналогично паскалевскому оператору CASE. Ее общий вид:

```
(COND
    (c1 f11 f12 ... f1n1)
    (c2 f21 f22 ... f2n1)
   ...
       (cm fm1 fm2 ... fmnm))
```

Здесь *c1…cm* – логические условия, *fnm* – функции, выполняемые при выполнении того или иного условия.

Внимание! Условия проверяются последовательно до первого истинного. Если истинно сразу несколько условий, то выполняются только функции, относящиеся к первому из них, а остальные условия даже не проверяются.

Основное назначение функции *COND* – обработка ввода пользователя, например, так:

```
(SETQ a (GETINT "\n1 - фаска, 2 - галтель, 3 - выточка"))
(COND
    ((= a 1) .... )   ; фаска
    ((= a 2) .... )   ; галтель
      ((= a 3) .... ) ; выточка
)
```

А вот пример неправильного применения функции *COND*. Пусть мы хотим присвоить переменной *flag* значение *NIL*, если хотя бы одна из сторон треугольника *a*, *b*, *c* оказалась отрицательной. Неопытному программисту вместо очевидного

```
(SETQ flag (NOT
(OR (<= a 0) (<= b 0) (<= c 0))))
```

может прийти в голову следующая пагубная идея:

```
(SETQ flag NIL)
 (COND
        ((> a 0) (SETQ flag T))
        ((> b 0) (SETQ flag T))
        ((> c 0) (SETQ flag T)))
```

Программа выглядит работоспособной, но представим себе, что будет происходить при следующих исходных данных: a=10, b=–5, c=2. Функция *COND* проверит первое условие $a>0$, убедится в его истинности, установит переменную *flag* в *T* и остальные условия вообще проверять не будет! Поэтому тот факт, что сторона b меньше нуля, останется незамеченным.

Порядок выполнения работы

Необходимо модифицировать программу из предыдущей лабораторной работы. После ввода параметров чертежа необходимо осуществить проверку корректности введенных данных. Например, может оказаться, что диаметр окружности превышает размеры самой детали.

В случае некорректности введенных данных необходимо вывести сообщение об ошибке и завершить выполнение программы.

Созданный файл загрузить в AutoCAD и выполнить его для корекктно и некоррекно введенных размеров детали.

Содержание отчета

Цель работы.
Текст программы с комментариями.
Описание используемых команд.
Выводы.

Контрольные вопросы

Как логические функции AutoLISP Вам известны?
Как организовать ветвление программы?
Для чего нужен оператор *PROGN*?
Каким образом можно сообщить пользователя об ошибке ввода?

6 ЛАБОРАТОРНАЯ РАБОТА № 6: «ИЗУЧЕНИЕ ОРГАНИЗАЦИИ ДИАЛОГА С ПОЛЬЗОВАТЕЛЕМ НА ОСНОВЕ ФУНКЦИЙ ЦИКЛОВ»

Цель работы

Изучить функции циклов в AutoLISP. Использовать функции циклов для организации ввода размеров детали. Исследовать работу функций циклов для построения повторяющихся объектов чертежа и ввода корректных размеров детали при параметрическом проектировании в среде AutoCAD.

Теоретический раздел

Одной из трех базовых структур в любом языке программирования является цикл. Нужно ясно понимать, что цикл выполняет двоякую функцию:

- выполнение одного и того же участка программы более одного раза;
- передача управления (под управлением понимается выполняемый в данный момент элемент программы) “назад” или “вверх” (т.е. ближе к началу программы).

Вторая функция цикла менее очевидна, чем первая, хотя в структурных языках и в AutoLISP цикл – единственный способ передать управление назад. Кстати, если необходимость многократного выполнения участка программы более-менее понятна (например, нам нужно нарисовать сотню окружностей), то необходимость возвратов в программе менее очевидна. Поэтому приведем хотя бы один простейший пример: пользователь вводит данные, программа что-то рисует, а затем пользователь смотрит на результат, ужасается увиденному, и очень хочет повторить ввод. Вот здесь и придется возвращаться назад – к тому куску программы, который вводит данные.

Циклы в AutoLISP, разумеется, обеспечиваются функциями. Простейшая из них – функция *REPEAT* вида

(REPEAT n f1 f2 ... fm)

Здесь *n* – число повторений, а *f1…fm* – те функции, которые будут выполняться *n* раз.

В цикле *REPEAT*, в отличие от, например, цикла FOR...TO в Паскале, нет переменной, значение которой изменяется от 1 до *n*. Поэтому таким циклом не стоит решать задачи перебора элементов списка. Для этого существуют специальные функции.

Назначение цикла *REPEAT* – в основном решение задач отрисовки. Например, если деталь имеет повторяющиеся элементы, можно запросить у пользователя их количество, затем вычленить повторяющийся участок и отрисовать его нужное число раз.

Пример: нарисовать деталь, показанную на рисунке. Число отверстий, идущих с заданным шагом, может варьироваться от 1 до 5.

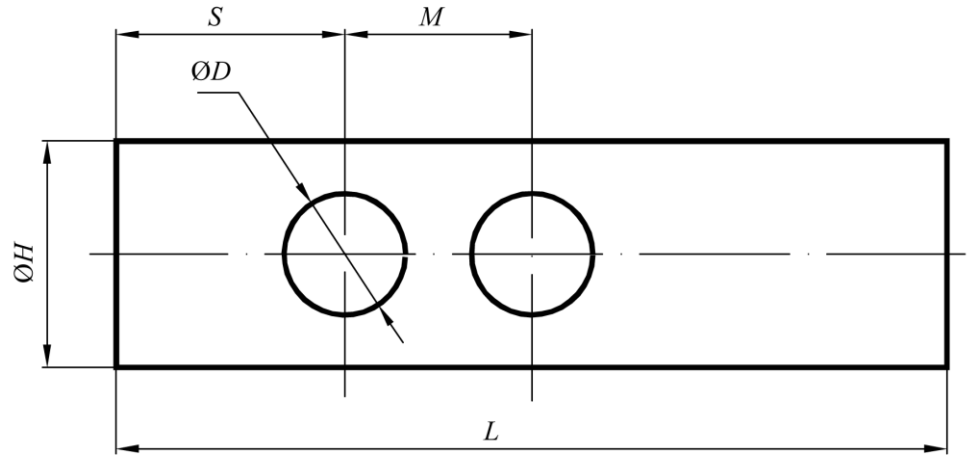


Рисунок 6.1 – Параметры для применения команды REPEAT

Предположим, что все данные, кроме числа отверстий, введены, сам контур детали нарисован и левый нижний угол находится в точке с координатами (0,0). Делаем следующее:

```
. . .
(INITGET 7)
(SETQ n (GETINT "\nВведите число отверстий:")
x (LIST s (/ h 2.0)))
(REPEAT n
    (COMMAND "CIRCLE" x (/ D 2.0))
      (SETQ x (POLAR x 0 m)))
```

В этом примере в переменную *x* записывается текущая координата центра окружности.

Однако одного такого цикла нам явно недостаточно. Как быть, если мы не знаем заранее, сколько раз нам нужно выполнять то или иное действие? Мы просто знаем, что нужно остановиться при выполнении определенного условия. Например, пусть в той же детали (см. рисунок) сверлятся отверстия, причем центр первого из них отстоит от левого края на величину *S*, а край последнего должен отстоять от правого края на величину *B*.

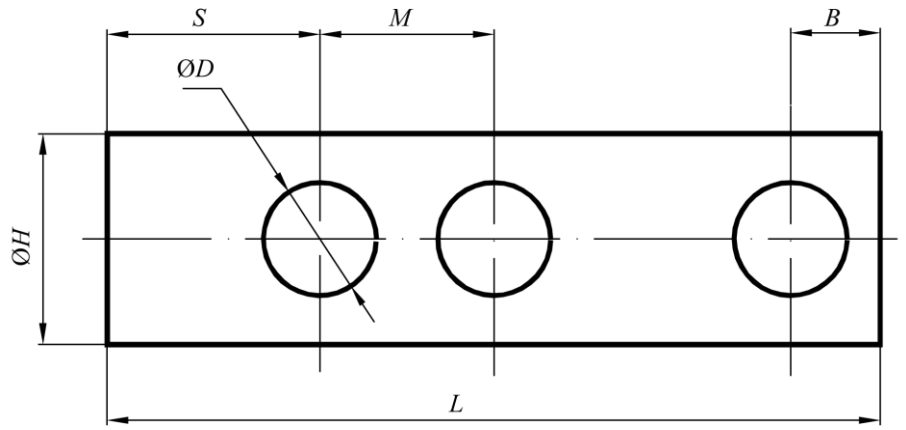


Рисунок 6.2 – Параметры для применения команды WHILE

Конечно, можно рассчитать число отверстий. А можно просто рисовать их до тех пор, пока соблюдается условие:

$$xr - x + \frac{D}{2} \le B$$

где *x* – текущая координата центра отверстия; *xr* – координата правой границы детали. В AutoLISP цикл по условию задается функцией *WHILE*:

(WHILE c f1 f2 ... fm)

Здесь *c* – логическое условие. Цикл выполняется, пока это условие истинно. В нашем случае (*p0* – координаты левого нижнего угла детали):

```
(SETQ pr (+ (CAR p0) L)) ; координата x правого конца детали
(SETQ x (+ (CAR p0) S)) ; Координата центра первого отверстия
(WHILE (< (– xr (+ x (/ D 2))) B)
    (COMMAND "CIRCLE"
    (LIST x (+ (CADR p0) H)) D)
      (SETQ x (+ x m)))
```

Обратите внимание: при использовании цикла *WHILE* нужно позаботиться о задании начальных условий (в рассматриваемом примере – о присваивании переменной *x* координаты первого отверстия), поскольку условие цикла проверяется до входа в него и, если оно не соблюдается, цикл *WHILE* не выполняется ни разу.

Цикл *WHILE* можно по ошибке сделать “вечным”. Чтобы этого не происходило, в теле цикла должны обязательно изменяться переменные, входящие в условие цикла. При зацикливании иногда не удается прервать цикл нажатием клавиши Ctrl+C или Esc. Тогда придется перезагружать компьютер с потерей чертежа.

Еще одно частое применение цикла *WHILE* – тот самый возврат к началу программы. Например, нужно что-то нарисовать и спросить у пользователя “Повторить? Да/Нет”. Самый простой способ сделать это – использовать переменную логического типа, в которую можно записать результат проверки условия и которую можно использовать в качестве условия цикла *WHILE*. Например:

```
(SETQ flag T)    ;Задание начального условия
(WHILE flag
    ....
    (SETQ ans (GETSTRING "\nПовторить?"))
    (SETQ flag (= (STRCASE ans ) "Y"))
)      ; конец цикла
```

Работает этот фрагмент кода следующим образом: в качестве условия цикла задана переменная *flag*. Это значит, что цикл выполняется, если в эту переменную записано логическое значение “истина”, т.е. T. Внутри цикла переменную *flag* нужно обязательно изменять, иначе произойдет зацикливание. Мы это делаем простейшим способом: вводим с клавиатуры букву (“*Y*” или “*N*”),

переводим ее на верхний регистр (для удобства пользователя – он сможет вводить и заглавные, и строчные буквы; разумеется, с русскими буквами функция *STRCASE* не работает) и затем сравниваем со значением "*Y*". Результат любого сравнения – либо *T* (действительно введена буква "*Y*"), либо *NIL* (введена другая буква). Этот результат мы и записываем в переменную *flag*. Затем управление передается снова на начало цикла и делается проверка: если *flag=T* – цикл выполняется снова, а если *flag=NIL* – цикл прекращается.

Как уже отмечалось выше, в AutoLISP нет цикла с переменной, которая меняется от min до max с заданным шагом. Между тем такие циклы есть практически во всех других языках программирования. В чем же дело? В том, что список – особая структура данных. Наиболее часто цикл с переменной нужен для прохода по всем элементам массива и их обработки (например, суммирования для нахождения среднего). В AutoLISP для обработки каждого элемента списка предусмотрены два особых цикла: *FOREACH* и *MAPCAR*.

Нет необходимости искусственно (при помощи цикла *WHILE*) создавать цикл с переменной для прохода по списку!

Рассмотрим цикл *FOREACH*:

```
(FOREACH name list exp)
```

Здесь *name* – имя переменной, в которую последовательно записываются элементы списка *list*, а *exp* – выражение AutoLISP, выполняемое столько раз, сколько элементов есть в списке *list*.

Рассмотрим работу цикла на примере. Пусть нам нужно просуммировать все элементы списка *data* и записать сумму в переменную *s*. Делается это так:

```
(SETQ s 0) ; Обнуление суммы
(FOREACH cur data (SETQ s (+ s cur)))
```

При выполнении этого фрагмента происходит следующее: берется список *data*, по очереди каждый его элемент, начиная с первого, записывается в переменную *cur*, после чего выполняется выражение

```
(SETQ s (+ s cur))
```

Функция *FOREACH* имеет одним из своих параметров не число или строку, а выражение AutoLISP, поскольку оно является просто списком – еще одна возможность, отсутствующая в большинстве других языков.

Если нужно выполнить сложное действие над элементами списка, есть два пути: написать для этого отдельную функцию и вызвать ее из *FOREACH*, передавая ей текущий элемент списка как параметр, или объединить несколько выражений при помощи функции *PROGN*. Например, если нужно и сложить, и перемножить элементы списка, это можно сделать в одном цикле:

```
(SETQ s 0 p 1) ; Инициализация суммы и произведения
((FOREACH cur data
    (PROGN (SETQ s (+ s cur))
       (SETQ P (* p cur))))
```

В AutoLISP нет возможности изменить значение отдельного элемента списка, хотя такая задача возникает довольно часто. Например, при масштабировании изображения нужно все элементы списка размеров *size* умножить на масштабный коэффициент *k*.

Существует элегантный способ сделать это при помощи функции *MAPCAR*. К сожалению, в большинстве книг по AutoLISP назначение функции *MAPCAR* или не разъясняется вовсе, или разъясняется неверно. Ее общий вид:

(MAPCAR 'f l1 l2 ... ln)

MAPCAR выполняет функцию *f* поочередно над первыми, вторыми, третьими ... элементами списков *l1 ... ln*. При этом n должно быть равно числу аргументов функции *f*, а все списки *l1 ... ln* должны содержать одинаковое число элементов. Самое главное – функция возвращает список результатов выполнения функции *f*.

Перед функцией нужно ставить апостроф, потому что *MAPCAR* нужна просто ссылка на то место в памяти, где эта функция находится, а не результат выполнения этой функции.

Рассмотрим работу функции на примере. Итак, предположим, что все элементы списка размеров *size* нужно умножить на масштабный коэффициент *k*. Функция умножения требует, как минимум два аргумента, в данном случае – текущий элемент списка и коэффициент *k*.

Первым делом нам придется сформировать вспомогательный список, состоящий из стольких значений *k*, сколько есть размеров в списке *size*: ведь функции умножения на вход будут даваться пары значений и для каждого элемента списка *size* нужно явно указать второй сомножитель. Делается это так:

```
(SETQ coeff NIL)
(REPEAT (LENGTH size)
        (SETQ coeff (APPEND coeff (LIST k))))
```

После этого можно применить функцию *MAPCAR*:

(SETQ size (MAPCAR ' size coeff))*

Такое решение работоспособно, но неэлегантно: приходится создавать лишний список *coeff*. Чтобы этого избежать, можно было бы написать отдельную функцию умножения на коэффициент *k*. Но снова возникает проблема: как передать в нее значение *k*?

Для решения подобных казусов было разработано так называемое *l*-исчисление Черча. Само по себе оно представляет математический аппарат для описания функций. В AutoLISP же *l*-исчисление заключается в наличии возможности создать “одноразовую” функцию, даже не имеющую своего имени. Причем внутри этой функции будут видны все текущие локальные переменные, т.е. для нашего примера снимается вопрос передачи значения *k* – оно будет видно в такой непоименованной функции, останется только подавать ей на вход по одному значения элементов списка *size*.

“Одноразовая” функция создается функцией AutoLISP *LAMBDA*:

(LAMBDA (arg1 ... argn) exp1 ... expm)

Здесь *arg1 ... argn* – список аргументов "одноразовой" функции, а *exp1 ... expm* – выражения AutoLISP, выполняющие какие-то операции над аргументами.

Как и при определении функции при помощи *RUN*, “одноразовая” функция возвращает результат вычисления последнего выражения, т.е. *expm*.

Сама по себе функция *LAMBDA* возвращает создаваемую ей временную функцию. Поэтому ее можно прямо записать как аргумент функции *MAPCAR*, не забыв поставить апостроф для подавления ее выполнения:

```
 (MAPCAR
'(LAMBDA(x) (* x k))
      size)
```

Здесь определена функция с одним аргументом *x*, которая вычисляется для каждого элемента списка *size*. Полученные произведения “слепливаются” в список, являющийся возвращаемым значением функции *MAPCAR*.

Порядок выполнения работы

Необходимо модифицировать программу из предыдущей лабораторной работы. При вводе данных, используя операторы цикла, необходимо осуществить проверку корректности введенных данных до тех пор, пока не введены размеры, позволяющих осуществить построение детали.

После ввода корректных размеров детали должен следовать запрос только о базовой точке для построения чертежа детали.

Созданный файл загрузить в AutoCAD и проверить его работу при различных вариантах вводимых размеров.

Содержание отчета

Цель работы.
Текст программы с комментариями.
Описание используемых команд.
Выводы.

Контрольные вопросы

Какие функции циклов AutoLISP Вам известны?
Чем отличается цикл *REPEAT* от цикла *WHILE*?
Для чего используется цикл *FOREACH*?
Приведите пример использованиея фукции *MAPCAR*.
Для чего используется фукция *LAMBDA*?
Какие функции цикла можно использовать для организации ввода корректных размеров детали?

7 ЛАБОРАТОРНАЯ РАБОТА № 7: «ИССЛЕДОВАНИЕ ВЗАИМОДЕЙСТВИЯ ДИАЛОГОВЫХ ОКОН И ПРОГРАММ AUTOLISP ПРИ ПАРАМЕТРИЧЕСКОМ ПРОЕКТИРОВАНИИ»

Цель работы

Изучить основные операторы языка построения диалоговых окон DCL. Исследовать механизм связи диалогового окна с программой AutoLISP при вводе размеров детали и построении чертежа в среде AutoCAD. Разработать окно диалага для ввода размеров детали, обеспечить передачу данных из диалогового окна в программу на AutoLISP.

Теоретический раздел

Итак, в предыдущих работах мы научились использовать язык AutoLISP для автоматизации параметрического проектирования в среде AutoCAD. Однако, после того как первые трудности с освоением языка и элементарных приемов работы остаются позади, возникают вопросы другого плана.

Сначала смутно, а затем все яснее разработчик начинает понимать, что созданные им десятки (или даже сотни) программ надо приводить в какую-то систему. У него возникает желание распространять свои разработки и получать за это деньги. На этом этапе приходит понимание разницы между программой (даже очень хорошей) и программным продуктом. И чаще всего выясняется, что с самого начала многое надо было делать не так, многое требуется переписать заново. Обнаруживается, что программы “почему-то” работают только в присутствии автора, на его компьютере, приходится расставаться с мечтами о заработке на продажах.

Следует отметить, что разработкой прикладных программ на базе AutoCAD занимаются, как правило, не “настоящие программисты”, а обычные инженеры (по крайней мере, на постсоветском пространстве). Вызвано это, по-видимому, тем, что проще инженеру освоить не очень сложные средства разработки для AutoCAD (и получать при этом великолепные результаты), чем заставить профессионального программиста заниматься этой “грязной” работой. Действительно, задачи, решаемые “под AutoCAD” обычно не требуют ни сложных математических методов, ни хитроумной обработки больших объемов данных, ни умелого обращения с ресурсами операционной системы. Зато решающее значение имеют, казалось бы, мелкие нюансы интерфейса пользователя с программой, знание предметной области, глубокое знание самого AutoCAD и, обязательно, любовь к пользователю. Все это приходит только в результате опыта практической работы по проектированию (как “бумажного”, так и “компьютерного”).

Естественно, что первым шагом к разработке программного продукта под AutoCAD является создание диалоговых окон. Конечному пользователю гораздо удобнее вводить данные в поля диалогового окна, чем работать с командной строкой, пусть даже программа выводит при этом самые подробнейшие подсказки.

Начиная с версии 12 AutoCAD предусматривает возможность самостоятельного написания диалоговых окон, отличных от определенных в системе. Для этой цели был даже разработан специальный язык – DCL (Dialogue Control Language, или другими словами – язык управления диалоговыми окнами).

Следует отметить, что Autodesk свято, как правило, свято придерживается принципов совместимости своих программных стандартов, и при переходе на Windows–версию AutoCAD, язык DCL не притерпел никаких серьезных изменений, и приложения, написанные для AutoCAD R12 (DOS) работают и под AutoCAD R14 и выше для Windows.

Кроме массы достоинств, которые влечет использование диалоговых окон в пользовательских приложениях под AutoCAD, существуют и определенные недостатки. К ним можно, прежде всего, отнести отсутствие встроенных средств отладки – в качестве отладчика используется сам AutoCAD, который пишет сообщения об ошибках в файл acad.dce, а также трудности с осуществлением дизайна окон (в сложных окнах эстетическое оформление расположения элементов довольно затруднено).

Диалоговые окна определяются текстовыми файлами ASCII, написанными на языке управления диалогов DCL с соответствующим расширением *.dcl. Элементы диалогового окна – кнопки, окна редактирования, известны как tile-элементы. Размер и функциональные возможности каждого элемента управляется атрибутами. Размер диалогового окна и размещение частей установлены автоматически.

Диалоговое окно состоит из поля и tile-элементов внутри этого поля. Основные типы тайлов предопределены специальным программным обеспечением – PDB (programmable dialog box). Можно создавать комплексные tile-элементы, называемые подсистемами, группируя tile-элементы в строки и столбцы, с или без охватывающего поля или границы.

Строка или столбец tile-элементов рассматривается как кластер. Подсистемы определяют группы tile-элементов или кластеров, которые могут использоваться в других диалоговых окнах.

Например, кнопки OK, Cancel, и Help сгруппированные в подсистему, определены как кластер из трех tile-элементов.

Подсистемы обрабатываются как единичные tile-элементы. Tile-элементы в пределах подсистемы называются дочерними. Файлы DCL организованы в древовидную структуру. Наверху дерева – tile-элемент *dialog*, который непосредственно определяет диалоговое окно. Приведенная ниже диаграмма демонстрирует пример структуры DCL-файла.

Размещение, вид, и поведение tile-элемента или подсистемы определяются атрибутами. Например, сам *dialog*, и наиболее предопределенные типы tile-элементов, имеют атрибут label, который определяет текст, связанный с tile-элементом. Атрибут *label* диалогового окна определяет заголовок сверху диалогового окна, атрибутам *label* кнопки определяет текст внутри кнопки, и так далее.

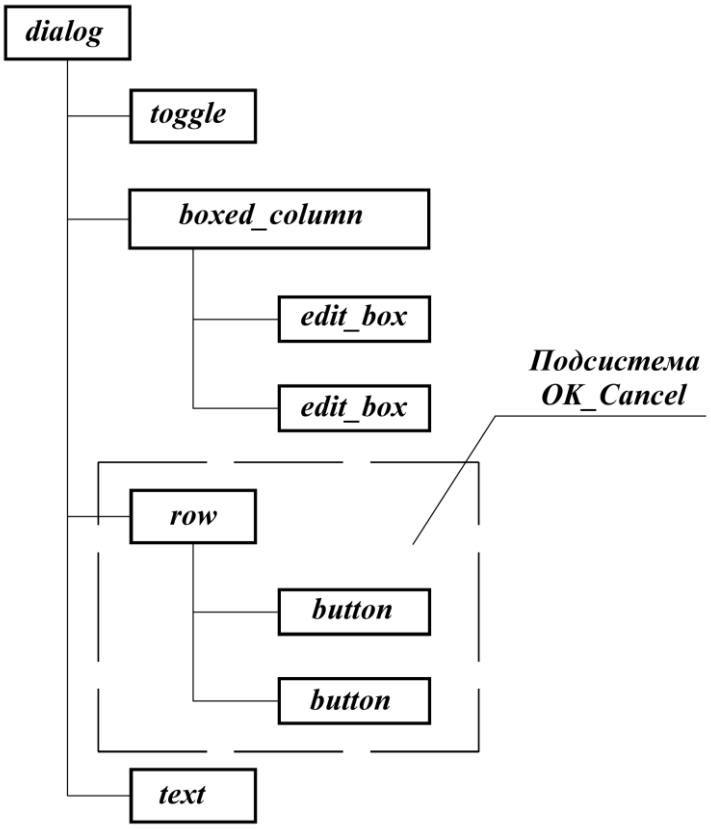


Рисунок 7.1 – Диаграмма структуры DCL-файла

DCL дает возможность определить новые tile-элементы, называемые прототипами, которые не обязательно связаны с каким либо диалоговым окном. Они полезны, когда необходимо использовать один компонент в нескольких диалоговых окнах, поскольку можно ссылаться на прототип tile-элементов из других файлов DCL, и изменяя их атрибуты, тем самым изменять предопределенные tile-элементы.

Новые tile-элементы создаются путем их определения. Определение tile–элемента имеет следующую форму:

name: item1 [: item2 : item3 ...] {
attribute = value;
... }

Здесь каждый *item* – предварительно определенный tile-элемент. Новый tile-элемент name наследует атрибуты всех указанных tile-элементов (*item1*, *item2*, *item3*, ...). Определения атрибута в фигурных скобках ({}) – это или добавления, или, если имя атрибута идентично, замена унаследованных определений. Когда определение имеет множественных родителей, атрибуты имеют приоритет слева направо. Другими словами, если один атрибут определяют несколько *items*, используется первый.

Обратите внимание, что имена tile-элементов учитывают регистр. Например, *bigbutton* – не то же самое, что *BigButton* или *BIGBUTTON*. Будьте внимательным при использовании преобразования букв в прописные.

Пример определение кнопки:

```
hello : dialog {
    label = "Sample Dialog Box";
    : text {
        label = "Hello, world";
        }
    : button {
        key = "accept";
        label = "OK";
        is_default = true;}
    }
```

Если запустить этот простейший код, то в окне AutoCAD при запуске мы увидим результат (см. рисунок 7.2)

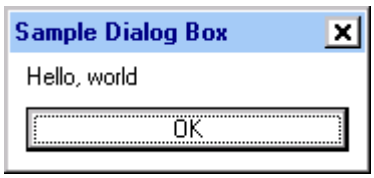


Рисунок 7.2 – Простейшее диалоговое окно

Обратите внимание, что кнопка OK занимает почти полную ширину диалогового окна. Чтобы улучшить вид этого диалогового окна, надо отредактировать файл DCL и добавить два атрибута к элементу *button*. Чтобы ограничить ширину кнопки, добавим атрибут *fixed_width*, и установим его равным *true*. Кнопка сожмется и станет немного шире, чем текст внутри. Чтобы выровнять кнопку по центру, добавим атрибут *alignment*. Элементы в столбце по умолчанию выравнивается по левому краю. Результат не заставит себя ждать:

Рисунок 7.3 – Диалоговое окно с атрибутами выравнивания

А фрагмет кода для этого окна будет выглядеть так:

```
hello : dialog {
label = "Sample Dialog Box";
: text {
 label = "Hello, world"; }
: button {
 key = "accept";
 label = "OK";
 is_default = true;
 fixed_width = true;
 alignment = centered; }
    }
```

Теперь можно рассмотреть основные tile-элементы. Существуют tile-элементы, используемые при построении диалоговых окон. Кроме этого, существуют активные группы tile-элементов. И наконец, имеются декоративные и иформационные tile-элементы. Описание всех этих элементов приведены в приложении Б.

Ниже приведены примеры различного расположения tile-элемента текст: код программы и результаты ее выполнения.

```
// DCL-файл (вариант а)
hello : dialog {
    label = "Sample Dialog Box";
    : text {label = "text1";}
    : text {label = "text2";}
    : text {label = "text3";}
    : text {label = "text4";}
    : button {
        key = "accept";
        label = "OK";
        is_default = true;
        fixed_width = true;
        alignment = centered; }
    }
```

```
// DCL-файл (вариант б)
hello : dialog {
    label = "Sample Dialog Box";
    : row {
     : text {label = "text1";}
     : text {label = "text2";}
     : text {label = "text3";} }
    : text {label = "text4";}
    : button {
        key = "accept";
        label = "OK";
        is_default = true;
        fixed_width = true;
        alignment = centered; }
    }
```

```
// DCL-файл (вариант в)
hello : dialog {
    label = "Sample Dialog Box";
    : row {
    : text {label = "text1";}
     : text {label = "text2";} }
    : row {
     : text {label = "text3";}
     : text {label = "text4";} }
    : button {
        key = "accept";
        label = "OK";
        is_default = true;
        fixed_width = true;
        alignment = centered; }
    }
```

```
// DCL-файл (вариант г)
hello : dialog {
    label = "Sample Dialog Box";
    : row {
        :column {
      : text {label = "text1";}
      : text {label = "text2";} }
     : text {label = "text3";}
     : text {label = "text4";} }
    : button {
        key = "accept";
        label = "OK";
        is_default = true;
        fixed_width = true;
        alignment = centered; }
    }
```

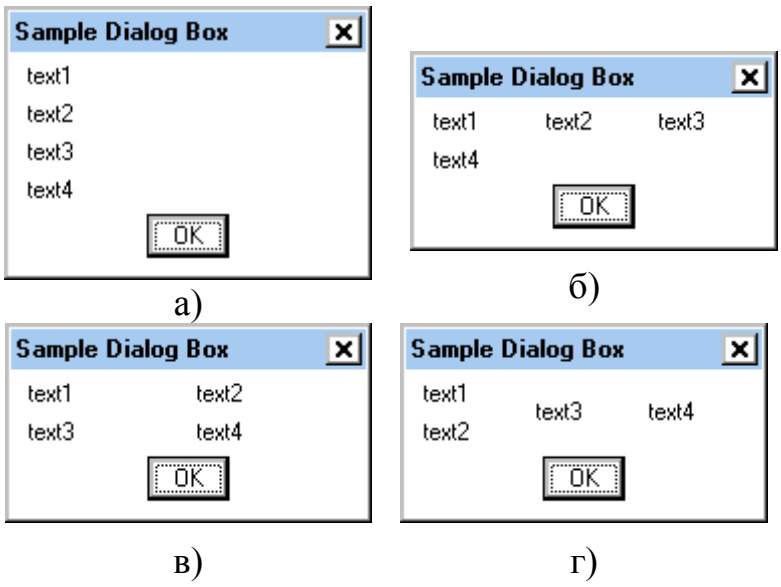


Рисунок 7.3 – Диалоговые окна с различными вариантами расположения текстовых элементов

Как использовать диалоги в программе на AutoLISP? Сначала DCL-файл надо найти и открыть. Обычно он располагается в том же каталоге, где и вызывающая программа.

(load_dialod <имя DCL-файла>)

Возвращаемое значение выражения: *DCL_ID* – целое число. Функция

(unload_dialog DCL_ID)

выгружает диалоговый файл. Она записывается после обработки всех полей диалогового окна и выхода из диалога. В DCL-файле может быть описано несколько диалоговых окон. Следующая функция вызывает диалоговое окно (по имени) и начинает управление этим окном:

(new_dialog <имя окна> DCL_ID)

После вызова этой функции происходит установка всех полей, присвоение значений, создание изображений и списков. Возвращаемое значение: *t* – при успешном открытии диалогового окна.

Итак, определяем порядок действий по созданию окна диалога. Создаем функцию. Внутри функции определяем идентификатор диалога – *load_dialog*, и проверяем, есть ли такой диалог – *new_dialog*. После этого определяем действия, связанные с ключами *action_tile*, и осуществляем запуск диалога функцией *start_dialog*. С помощью оператора *cond* определяем выбранный пользователем вариант, выгружаем диалог из памяти функцией *unload_dialog*, и все! Самый простой пример – обработка диалога с кнопками. Рассмотрим более подробно пример программы:

```
// DCL-файл. Описание диалога для панели с кнопками
dd_button : dialog {
    label = "Тестирование кнопок";
    fixed_height = true;
```

```
    : button {
        key = "btn1";
        value = "0";
        fixed_height = true;
        alignment = center;
        label = "Кнопка 1"; }
     : button {
        key = "btn2";
        value = "0";
        fixed_height = true;
        alignment = center;
        label = "Кнопка 2"; }
    : row {
        cancel_button;}
}
```

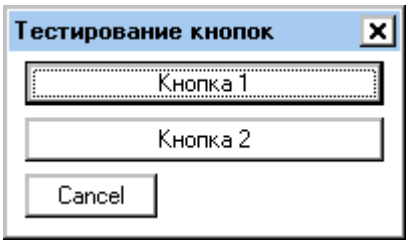


Рисунок 7.4 – Вид окна диалога

```
; AutoLISP-файл
; Запуск функции работы с диалоговым окном.

(defun dd_btn ( / dcl_id what_next)

; Загрузка диалога. Функция позволяет загрузить
; диалоги из указанного файла в память

(setq dcl_id (load_dialog "ot_tab.dcl"))

; Инициализация диалога – проверка существования.
; Если среди новых диалогов нет того, который нам
; необходим, то программа завершает работу.

(if (not (new_dialog "dd_button" dcl_id))
    (exit))


; Две кнопки = два действия, по первой кнопке результат – 1.
; Оператор action_tile описывает действия, которые должны
; происходить при выборе того или иного элемента управления.
; Элемент управления определяется по "key". Действия
; записываются в кавычках.

(action_tile "btn1" "(done_dialog 1)")
(action_tile "btn2" "(done_dialog 2)") ; По второй кнопке результат – 2.
```

```
; В данном случае диалог возвращает 1 или 2 в зависимости от нажатой
; кнопки. Следующая строка программы запускает диалог,
; результат которого будет записываться в переменную what_next.
; Запуск диалога – результат запишется в what_next.

 (setq what_next (start_dialog))
(cond
    ((= what_next 2) (alert "Запуск по кнопке 2!"))
    ((= what_next 1) (alert "Запуск по кнопке 1!")))
(unload_dialog dcl_id)) ; Выгружаем из памяти диалоги.
 (prompt "dd_btn ")
```

Следующий пример отличается тем, что используется цикл. Для того чтобы дать возможность пользователю кликать/щелкать переключателями сколь угодно раз до нажатия ОК, необходимо процесс обработки данных заключить в цикл. Пока не будет *what_next* = 1, цикл будет крутиться. Функция *ok_tab* формирует данные для выхода.

```
//DCL-файл. Переключатели
dd_radio : dialog {
    label = "Тестирование переключателей";
    fixed_height = true;
    : boxed_column {
        label = "Выберите вариант";
        : radio_column {
            : radio_button {
            key = "radio1"; label = "Radio1";
            value = 1; }
        : radio_button {
            key = "radio2";
            label = "Radio2"; }
        : radio_button {
            key = "radio3";
            label = "Radio3"; }
        }
    }
    : row {
        ok_button;
        cancel_button;  }
      }
```

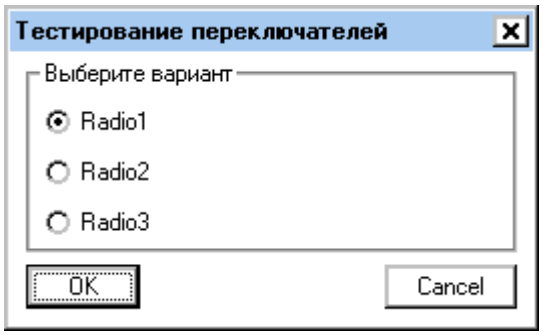


Рисунок 7.5 – Вид окна диалога

```
; AutoLISP-файл
(defun dd_radio ( / ret_value1 dcl_id what_next on_rad)
(defun ok_tab ( / ) ; функция, вызываемая по OK
    ; формирование списка данных на выход
    (setq ret_value1 (list "radio" on_rad)))
    (set_tile "radio1" "1"); Загрузка диалога
    (setq on_rad "radio1")
    (setq dcl_id (load_dialog "ot_tab.dcl"))
    (if (not (new_dialog "dd_radio" dcl_id)) ; Инициализация диалога
        (exit) ; Выход, если это не работает.
    )
 (setq what_next 8)
 (while (< 2 what_next)
    (action_tile "radio1" "(setq on_rad $key)")
    (action_tile "radio2" "(setq on_rad $key)")
    (action_tile "radio3" "(setq on_rad $key)")
    (action_tile "accept" "(done_dialog 1) (ok_tab)")
    (setq what_next (start_dialog)))
 (unload_dialog dcl_id) ; Выгрузить DCL-файл.
 (setq relst ret_value1))
(prompt "dd_radio")
```

А вот примере с полями, где особенностью является задание значений полей заранее – в тексте программы.

```
// DCL-файл. Диалог с полями
dd_edit : dialog{
    label = "Тестирование полей";
    fixed_height = true;
        : edit_box {
            key = "edit1";
            value = "0";
            fixed_height = true;
            alignment = center;
            label = "edit1"; }
        : edit_box {
            key = "edit2";
```

```
            value = "0";
            fixed_height = true;
            alignment = center;
            label = "edit2"; }
        : row {
            ok_button;
            cancel_button;}
    }
```

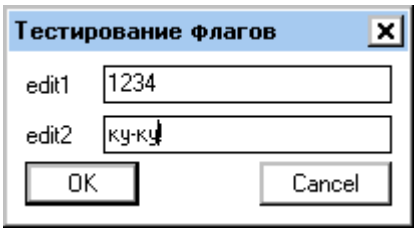


Рисунок 7.6 – Вид окна диалога

```
; AutoLISP-файл
(defun dd_edit ( / ret_value1 dcl_id what_next on_ed2 on_ed1)
(defun ok_tab ( / ) ; функция, вызываемая по OK
    ; формирование списка данных на выход
    (setq ret_value1 (list (list "ed1" on_ed1)
     (list "ed2" on_ed2))) )
    (setq dcl_id (load_dialog "ot_tab.dcl")); загрузка диалога
    (if (not (new_dialog "dd_edit" dcl_id)) ; инициализация диалога
        (exit)) ; Выход, если это не работает.
    (set_tile "edit2" "ку-ку!")
    (set_tile "edit1" "1234")
    (setq on_ed1 "1234" on_ed2 "ку-ку!")
    (setq what_next 8)
    (while (< 2 what_next)
        (action_tile "edit1" "(setq on_ed1 $value)")
        (action_tile "edit2" "(setq on_ed2 $value)")
        (action_tile "accept" "(done_dialog 1) (ok_tab)")
        (setq what_next (start_dialog))
    )
    (unload_dialog dcl_id) ; Выгрузить DCL-файл.
     (setq relst ret_value1)
)
 (prompt "dd_edit ")
```

Рассматривая пример с флагами, необходимо отметить, что в данном случае мы заранее инициализируем значения. Функция *on_type_os* изменяет элементы окна на заданные в параметрах функции. В зависимости от параметров, с помощью оператора *set_tile* устанавливаем значения флагов.

```
// DCL-файл. Диалога с флагами
dd_toggle : dialog {
```

```
    label = "Тестирование флагов";
    fixed_height = true;
    : toggle {
        key = "tog1";
        value = "0";
        fixed_height = true;
        alignment = center;
        label = "Флаг1"; }
    : toggle {
        key = "tog2";
        value = "0";
        fixed_height = true;
        alignment = center;
        label = "Флаг2"; }
    : row {
        ok_button;
        cancel_button;}
      }
```

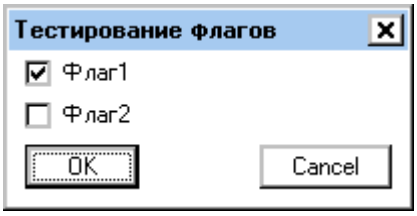


Рисунок 7.7 – Вид окна диалога

```
; AutoLISP-файл
(defun dd_toggle (type_os / ret_value1 dcl_id what_next on_tog1 on_tog2)
(defun ok_tab ( / ) ; Формирование списка данных на выход
     (setq ret_value1 (list (list "tog1" on_tog1)
     (list "tog2" on_tog2)))
 )
; Выбор включенных-выключенных ключей в
; зависимости от введенных параметров
(defun on_type_os( / )
    (cond
        ((= type_os "A")
        (setq on_tog1 "1" on_tog2 "0")
        (set_tile "tog1" "1")
        (set_tile "tog2" "0"))
        ((= type_os "B")
        (setq on_tog1 "1" on_tog2 "1")
        (set_tile "tog1" "1")
        (set_tile "tog2" "1"))
        (t))
    )
```

```
    (on_type_os)
    (setq dcl_id (load_dialog "ot_tab.dcl"))
    (if (not (new_dialog "dd_toggle" dcl_id))
    (exit)
)
; Исходная инициализация ключей в зависимости от
; введенныхпараметров
 (on_type_os)
 (setq what_next 8)
(while (< 2 what_next)
     (action_tile "accept" "(done_dialog 1)(ok_tab)")
     (action_tile "tog1" "(setq on_tog1 $value)")
     (action_tile "tog2" "(setq on_tog2 $value)")
     (setq what_next (start_dialog))
 )
(unload_dialog dcl_id)
(setq relst ret_value1))
 (prompt "dd_toggle")
```

В примере с кнопками ситуация еще интереснее. Необходимо создать слайды, которые будут использоваться на кнопке. Слайды создаются так: рисуете на поле чертежа что-нибудь такое. Введите команду *mslide* – и введите имя файла–слайда. Все! Слайд готов. Для отображения слайда в окне AutoCAD необходимо ввести команду *vslide*. Тогда слайд временно выведется на рабочее поле.

Размеры изображения задаются в DCL-файле, и еще изображение надо инициализировать в AutoLISP-файле – посмотрите пример! А обработка слайда производится так же, как и обработка кнопок.

```
// DCL-файл. Диалога с картинками-кнопками
dd_imgbutton : dialog {
    label = "Тестирование кнопок";
    : boxed_row {
        label ="Sizes";
        fixed_height = true;
        :image_button{
            color = 0;
            height = 1.5;
            width = 6;
            aspect_ratio = 0.36;
            fixed_height = true;
            fixed_width = true;
            alignment = center;
            key = "btn1";}
    :image_button {
        color = 0;
```

```
            height = 1.5;
            width = 6;
            aspect_ratio = 0.36;
            fixed_height = true;
            fixed_width = true;
            alignment = center;
            key = "btn2";}
        }
        : row {
            cancel_button;}
          }
```

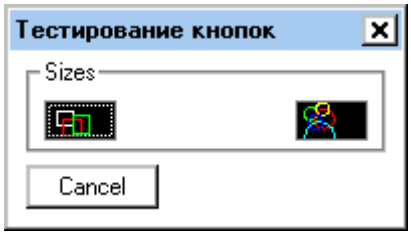


Рисунок 7.8 – Вид окна диалога

```
; AutoLISP-файл
(defun dd_imgbtn ( / dcl_id what_next)
    (setq dcl_id (load_dialog "ot_tab.dcl")); Загрузка диалога
    (if (not (new_dialog "dd_imgbutton" dcl_id)) ; Инициализация диалога
        (exit))
    (start_image "btn1")
    (slide_image 0 0 (dimx_tile "btn1")
    (dimy_tile "btn1") "btn1" )
    (end_image)
    (start_image "btn2")
    (slide_image 0 0 (dimx_tile "btn2")
    (dimy_tile "btn2") "btn2" )
    (end_image)
    (setq what_next 8)
    (action_tile "btn1" "(done_dialog 1)")
    (action_tile "btn2" "(done_dialog 2)")
    (setq what_next (start_dialog))
    (cond
        ((= what_next 2)
        (alert "Запуск по кнопке 2!"))
        ((= what_next 1)
        (alert "Запуск по кнопке 1!"))
    )
    (unload_dialog dcl_id)
)
(prompt "dd_imgbtn ")
```

В результате Вы должны получить на экране примерно следующее:

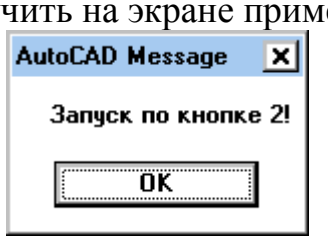


Рисунок 7.9 – Окно вывода результата выбора

В работе со списками основная сложность – составление самого списка. Он составляется как символьная строка, через знак “*n*” – перевод каретки.

```
// DCL-файл. Диалог со списком
dd_list : dialog {
    label = "Тестирование списков";
    :popup_list{
        label = "1s&t: ";
        key = "lst1";
        list = "None \nDatum Triangle Filled \nDatum Triangle \nIntegral
\nUser Arrow...";
        edit_width = 20;}
    :popup_list{
        label = "2n&d:";
        key = "lst2";
        list = "None \nDatum Triangle Filled \nDatum Triangle \nIntegral
\nUser Arrow...";
        edit_width = 20;}
    : row {
        ok_button;
        cancel_button;}
      }
```

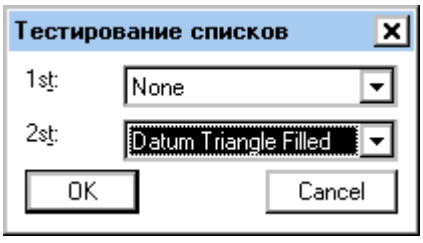


Рисунок 7.8 – Вид окна диалога

```
;AutoLISP-файл
(defun dd_list ( / ret_value1 dcl_id what_next on_list1 on_list2)
(defun ok_tab ( / ) ; функция, вызываемая по OK
    ; формирование списка данных на выход
     (setq ret_value1 (list (list "list1" on_list1)
    (list "list2" on_list2)))
    )
    (setq on_list1 0 on_list2 0)
```

```
(setq dcl_id (load_dialog "ot_tab.dcl"))
(if (not (new_dialog "dd_list" dcl_id))
      (exit))
(setq what_next 8)
(while (< 2 what_next)
      (action_tile "lst1" "(setq on_list1 $value)")
      (action_tile "lst2" "(setq on_list2 $value)")
      (action_tile "accept" "(done_dialog 1) (ok_tab)")
      (setq what_next (start_dialog))
)
 (unload_dialog dcl_id)
(setq relst ret_value1))
(prompt "dd_list")
```

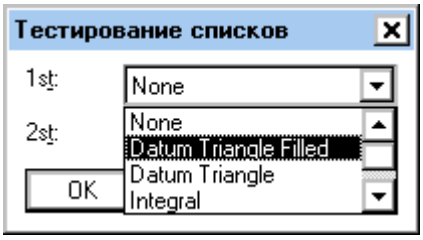


Рисунок 7.8 – Тестирование окна диалога

Последнее замечание – не забудьте преобразовать данные, введенные в *edit_box* из текста в число. Например, это можно сделать при помощи функции *atof*:

```
(setq D (atof D_txt))
```

Итак, мы рассмотрели основные типы диалоговых окон. Теперь можно создать свой собственный диалог, сделав первый шаг к разработке программного продукта.

Порядок выполнения работы

Необходимо разработать диалоговое окно для ввода размеров детали из предыдущих лабораторных работ.

Разработать передачу введенных параметров в программу AutoLISP.

Воспользоваться разработанным в предыдущих работах кодом для организации отрисовки детали.

Созданные файлы загрузить в AutoCAD и выполнить.

Содержание отчета

Цель работы.

Текст DCL-файла с комментариями.

Текст программы с комментариями.

Вид диалогового окна.

Выводы.

Контрольные вопросы

Как логические функции AutoLISP Вам известны?
Как организовать ветвление программы?
Для чего нужен оператор *PROGN*?
Каким образом можно сообщить пользователя об ошибке ввода?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Грувер М. САПР и автоматизация производства / М. Грувер, Э. Зиммерс. – М.: Мир, 1987. – 528 с.

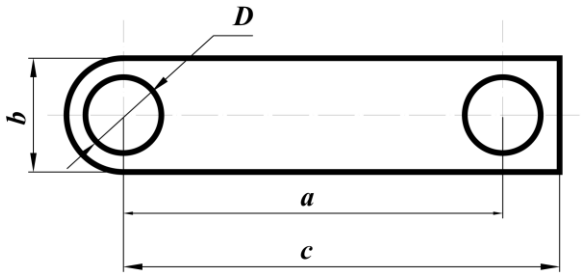
2. Шпур Г. Автоматизированное проектирование в машиностроении / Г. Шпур, Ф.-Л. Краузе. – М.: Машиностроение, 1988. – 648 с.

3. Полещук Н. AutoLISP и Visual LISP в среде AutoCAD / Н.Н. Полещук, П.В. Лоскутов. – СПб.: БХВ – Петербург, 2006. – 960 с.

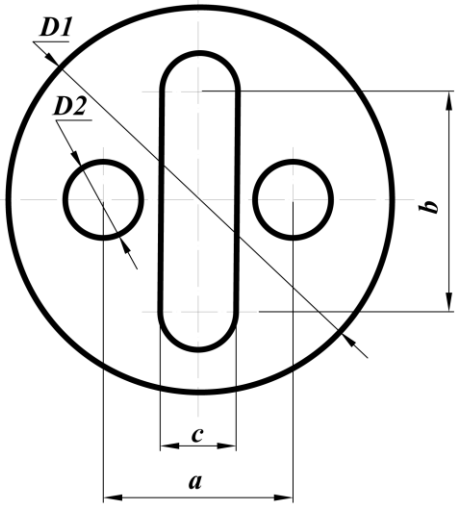
4. Ли К. Основы САПР (CAD/CAM/CAE) / К. Ли – СПб.: Питер, 2004. – 560 с.

5. Зуев С. САПР на базе AutoCAD – как это делается / С.А. Зуев, Н.Н. Полищук. – СПб.: БХВ – Петербург, 2004. – 1168 с.

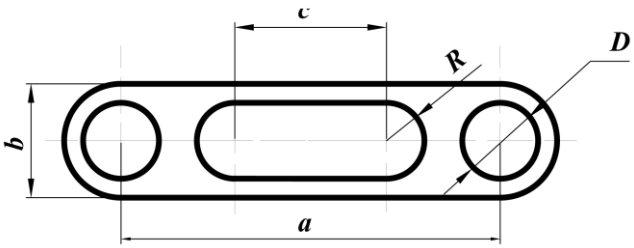
ПРИЛОЖЕНИЕ А - (обязательное). Варианты заданий



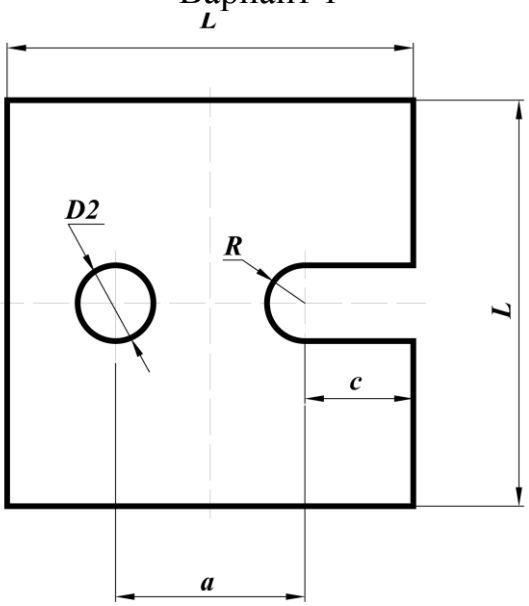
Вариант 0



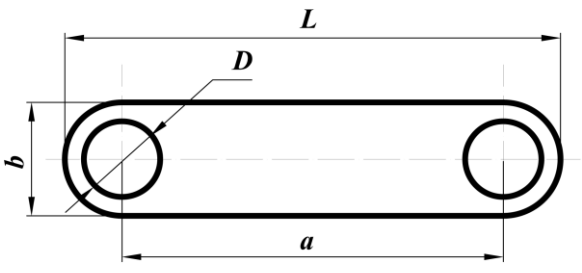
Вариант 1



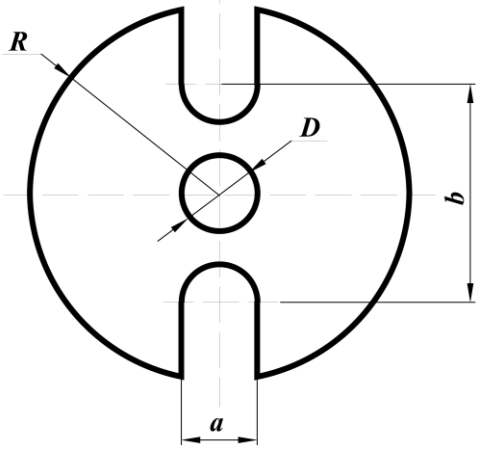
Вариант 2



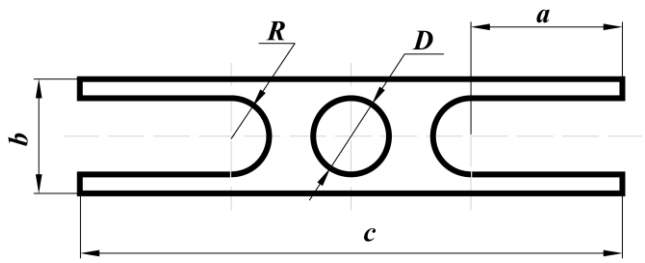
Вариант 3



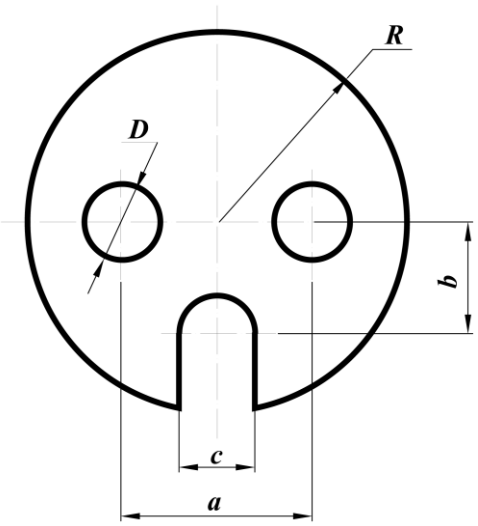
Вариант 4



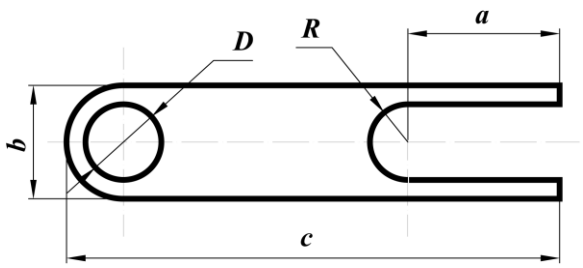
Вариант 5



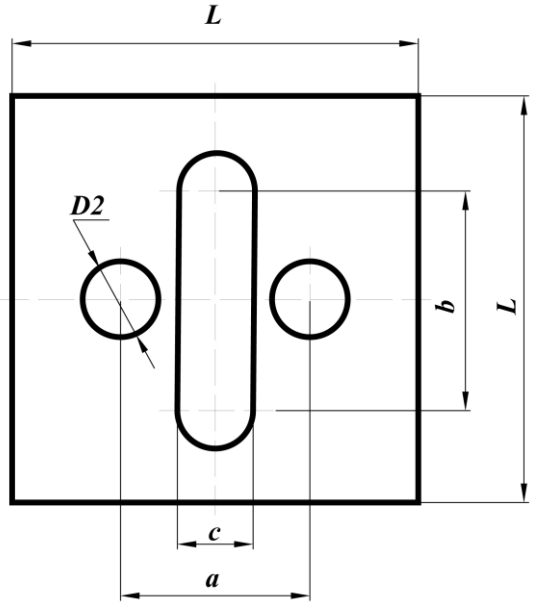
Вариант 6



Вариант 7



Вариант 8



Вариант 9

ПРИЛОЖЕНИЕ Б - (справочное). Tile-элементы диалоговых окон

Таблица Б.1 – Tile-элементы построения диалоговых окон

tile-элемент	Атрибуты	Описание
button		Обычная клавиша Windows, кнопка.
	label:	Строка в кавычках, определяет надпись, которая будет на клавише.
	is_default:	Действие по умолчанию. Если значение равно *FALSE* или отсутствует, кнопка не подсвечивается выбором. При значении *TRUE* (допускается только для одной кнопки окна) клавиша выделена и при нажатии клавиши ENTER происходит действие, сопряженное с этой кнопкой.
	is_cancel:	Тоже самое, что и *is_default*, но при нажатии клавиши CANCEL.
edit_box		Поле для ввода или редактирования текста.
	label:	Текстовая строка в кавычках. Выводится слева от поля.
	edit_width:	Целое число – максимальное число символов, ограниченное рамкой, если значение не указано, поле маштабируется по окну.
	edit_limits:	То же самое с точностью до наоборот.
	value:	Строка в кавычках – исходное значение.
list_box		Поле с текстом в несколько строк. Обычно список. При количестве строк большем, чем высота окна, справа автоматически появляется шкала прокрутки.
	label:	Строка в кавычках, выводимая над полем списка – типа заголовока.
	multiple_select:	Возможность множественного выбора с помощью “SHIFT + левая кнопка мыши” при значении *TRUE*. *FALSE* – выбор только одного пункта.
	list:	Строка в кавычках – начальный выбор строки в поле списка.
	value:	Строка из чисел (начиная с нуля), заключенных в кавычки – первоначальный выбор пунктов списка. Соответственно *multiple_select* может содержать несколько чисел *TRUE*, или одно *FALSE*.

Продолжение таблицы Б.1

tile-элемент	Атрибуты	Описание
image_button		Поле с графическим изображением. Возвращаемое значение – координаты точки, в которой произошел выбор.
	color:	Цвет фона (номер или символьное имя AutoCAD): *dialog_line* – цвет диалогового окна, *dialog_forground* – цвет символа, *dialog_background* – цвет фона, *graphic_background* – цвет фона графического экрана, *graphic_forground* – цвет символа (черным по белому).
popup_box		Поле с текстом в несколько строк, аналогичное *list_box*, но выглядит как поле со стрелкой справа, открывающей поле списка для выбора.
	label:	Строка в кавычках, выводимая слева списка – значения по умолчанию нет.
	edit_widtht:	Число – ширина поля в единицах ширины символа. При умолчании или нуле – выравнивается, насколько возможно, по окну.
	list:	Строка в кавычках – начальный выбор строки в поле списка.
	value:	Строка из чисел (начиная с нуля), заключенных в кавычки, – первоначальный выбор пунктов списка. Этот пункт будет выводиться в закрытом окне на экран.
rado_button		Группа или одна кнопочка ввиде кружка с текстом справа, объединенных в колонку или, реже, в ряд. Возможен выбор только одной кнопки. При выборе другой кнопки из группы – первая отключится.
	label:	Строка в кавычках, выводимая справа от кнопки – значения по умолчанию “нет”.
	value:	Строка в кавычках с атрибутом включения кнопки при открытии окна. “1” – кнопка включена, “0” и все другие значаения – кнопка выбора отключена.

Продолжение таблицы Б.1

tile-элемент	Атрибуты	Описание
slider		Перемещаемый пользователем движок, премещение соответствует изменению числового значения, передающегося в программу.
	min_value: *max_value:*	Целые числа, определяющие граничные значения.
	small_increment: *big_increment:*	Целые числа, определяющие значения приращения. По умолчанию первое – сотая доля диапозона, второе – десятая. Атрибуты не обязательны.
	layout:	Горизонтальная (по умолчанию) или вертикальная шкала.
	label:	Строка в кавычках, выводимая справа от кнопки – значения по умолчанию нет.
	value:	Строка в кавычках (целое число) с текущим значением шкалы по умолчанию – *min_value.*
toggle		Перключатель “0” или “1” – небольшой прямоугольник с меткой “X”, включен/выключен.
	label:	Строка в кавычках, выводимая у кнопки – значения по умолчанию нет.
	value:	Строка в кавычках (целое число “0” или “1”) включен/выключен. При значении “1” выводится “X”.

Таблица Б.2 – Декоративные и иформационные tile-элементы

tile-элемент	Атрибуты	Описание
image		Прямоугольник с векторным изображением.
	color:	Цвет фона (номер или символьное имя цвета в AutoCAD), *dialog_line*, *dialog_forground*, *dialog_background*, *graphic_background*, *graphic_forground* аналогичны *image_button*.
	acpect_ratio:	Отношение ширины изображения к высоте.
text		Текстовая строка.
	label:	Строка в кавычках, выводимая в поле надписи.
	value:	Аналогична *label*, но не влияет на компоновку полей. Если сообщение является неименяемым, следует определить атрибут *label* без определения *width* и *value* .
spacer		Разделитель – пустое поле. Атрибутов нет.

Таблица Б.3 – Tile-элементы построения активных групп

tile-элемент	Атрибуты	Описание
column		Вертикальная колонка элементов. Колонка не имеет дополнительных атрибутов.
row		Горизонтальный ряд элементов. Ряд не имеет дополнительных атрибутов.
boxed_column		Вертикальная колонка элементов с рамкой по периметру.
	label:	Строка в кавычках, выводимая в верхнем левом углу колонки, значение по умолчанию “ “.
boxed_row		Горизонтальный ряд элементов с рамкой по периметру.
	label:	Строка в кавычках, выводимая в верхнем левом углу ряда, значение по умолчанию “ “.
radio_column		Колонка, содержащая поля кнопок выбора, из которых можно выбрать только одну кнопку.
	value:	Строка в кавычках, содержащая значение ключа “key” выбранной текущей кнопки выбора.
radio_row		Ряд, содержащий поля кнопок выбора, из которых можно выбрать только одну кнопку.
	value:	Строка в кавычках, содержащая значение ключа “key” выбранной текущей кнопки выбора.

АВТОМАТИЗАЦИЯ ПАРАМЕТРИЧЕСКОГО ПРОЕКТИРОВАНИЯ

Методические указания

Составители: **Карташов** Леонид Евгеньевич,
Худаймуратов Мурат Абдуллаевич, **Федоренко** Сергей Николаевич

Издается в авторской редакции

Изд. № 226/19. Объем 3,75 п.л.
РИИЦМ ФГАОУ ВО «Севастопольский государственный университет»