ICAD course

ICAD COURSE.

Mathieu A.J. de Vries
Johan M.M. Zentjens

Eindhoven, Prinsenbeek
27 januari 1992

WPA 1235

WPA reportnr.:
1235

Version:
1.1

Supervisors:
dr.ir. F.L.M. Delbressine
ir. R. de Groot

Bijzondere Onderwerpen Produktie-
en Meetmiddelen (4U041)

CONTENTS.

CHAPTER 1. PHILOSOPHY.
This chapter explains something about 'Knowledge-based Engineering'.

CHAPTER 2. GNU EMACS EDITOR AND SYSTEM COMMANDS.
Chapter 2 tells you how to handle with the SUN computer and how to use the GNU EMACS Editor.

CHAPTER 3. ICAD DESIGN LANGUAGE AND COMMON LISP.
This chapter deals with some aspects about Object Oriented Programming in Common LISP and about how to use the ICAD Design Language.

CHAPTER 4. DEFPART ELEMENTS.
One of the most important things about ICAD are 'defparts'. This chapter will explain more about these.

CHAPTER 5. THE ICAD BROWSER.
The ICAD Browser is the tool to show you your defined object. In this chapter the most important aspects will be explained.

CHAPTER 6. POSITIONING AND ORIENTATION.
In the real world objects are placed and oriented. The same is done in ICAD. This chapter will show some examples of how to position and orientate objects.

CHAPTER 7. QUANTIFICATION.
Quantification is a means of replicating using a single object. In this way it is not necessary to specify more than this single object. Chapter 7 will tell you something about this.

CHAPTER 8. TRICKS SPECIAL.
Because every beginner has to deal with some programming problems, this chapter is written to help you with some of these problems.

CHAPTER 9. EXERCISE.
This chapter contains some exercises to test your knowlegde about ICAD.

CONTENTS.

CHAPTER 1. PHILOSOPHY. 1.1

1.1. Introduction 1.1
1.2. What is Knowledge-based Engineering? 1.2
1.3. What benefits do knowledge-based CAD systems give us? 1.3
1.4. What disadvantages are there to knowledge-based CAD systems? 1.3
1.5. What kind of applications knowledge based CAD systems can be used for? . 1.4

CHAPTER 2. GNU EMACS EDITOR AND SYSTEM COMMANDS. 2.1

2.1. What is the purpose of this document? 2.1
2.2. What you have to know before you even touch the SUN system? 2.1
2.3. How to enter the SUN system? 2.1
2.4. How to leave the SUN system? 2.2
2.5. Can Sunview be left temporary? 2.2
2.6. What do you have to know about GNU EMACS Editor? 2.3
2.7. For what is GNU EMACS editor used and how is it working? 2.3
2.8. How to get descriptions of defparts in the EMACS Editor? 2.3
2.9. How to save the buffer? 2.4
2.10. How to edit a text in the Text area? 2.4
2.11. Where to find a summary of all editing commands? 2.5
2.12. What to do after the text has been edited? 2.5

CHAPTER 3. ICAD DESIGN LANGUAGE AND COMMON LISP. 3.1

3.1. What is IDL? 3.1
3.2. What kind of language is Common LISP? 3.1
3.3. What do you have to know about Object Oriented Languages? 3.1
3.4. What rules are to be considered in Common LISP? 3.2
3.5. What are expressions in Common LISP? 3.2
3.6. What about conditional expressions? 3.3
3.7. What predicate functions are available for conditionals? 3.3
3.8. Where you do and where you do not use conditionals? 3.4
3.9. What additional LISP functions are used? 3.4
3.10. What does IDL consist of? 3.6
3.11. What is a defpart? 3.6
3.12. What can you do with the ICAD Browser? 3.6
3.13. What should you have learned until now? 3.7

CHAPTER 4. DEFPART ELEMENTS. 4.1

4.1. How to construct an IDL-description? 4.1
4.2. What about :attributes? 4.1
4.3. Which specialities of :attributes are known? 4.2
4.4. What are :parts? 4.2
4.5. What about generic parts? 4.3
4.6. What about :inputs? 4.3
4.7. What about :optional-inputs? 4.4
4.8. What other ways are possible to ask for input? 4.4
4.9. What about the tree structure? 4.5
4.10. What about inheritance? 4.8
4.11. What about reference chains? 4.8

CHAPTER 5. THE ICAD BROWSER. 5.1

5.1. What do you have to know about the ICAD Browser? 5.1
5.2. What is the ICAD Browser used for and how does it work? 5.1
5.3. How to switch to the ICAD Browser? 5.2
5.4. How to quit and leave the ICAD Browser? 5.2
5.5. How to work with the ICAD Browser? 5.2
5.5.1. What to do with the menubars? 5.2
5.5.2. How to get an instance of a defpart on the Browser screen? 5.3
5.5.3. What are the leaves and nodes used for? 5.5
5.5.4. What is the ModifyGraphics command used for? 5.5
5.5.5. What is the EditView command used for? 5.5
5.6. Where to find more information about the Browser commands? 5.6

CHAPTER 6. POSITIONING AND ORIENTATION. 6.1

6.1. What is needed for building objects? 6.1
6.2. What are faces and axis of a defpart and a part? 6.1
6.3. What about :position ? 6.2
6.4. What keywords are used for face-to-face positioning? 6.2
6.5. What keywords are used for center-to-center positioning? 6.3
6.6. What if you do not wish to position in respect to the defpart? 6.3
6.7. What about :orientation ? 6.4
6.8. Can I specify more than one rotation and translation? 6.6

CHAPTER 7. QUANTIFICATION. 7.1

7.1. What is Quantification? 7.1
7.2. What if I don't want the quantified objects evenly spaced? 7.2
7.3. What about referencing quantified objects? 7.3
7.4. How to distinguish a certain quantified object? 7.6

CHAPTER 8. TRICKS SPECIAL. 8.1

8.1. What can you use this chapter for? 8.1
8.2. What compile-time and edit-time errors can be made? 8.1
8.2.1. Unbalanced parentheses. 8.1
8.2.2. Misbalanced parentheses. 8.1
8.2.3. Function undefined. 8.1
8.2.4. Redefining old version. 8.1
8.3. What run-time errors can be made? 8.2
8.3.1. No parent could handle the XXX message. 8.2
8.3.2. The function XXX is undefined. 8.2
8.3.3. The control (or binding) stack overflowed. 8.2
8.3.4. The variable XXX is unbound. 8.2
8.3.5. The argument given to the SYS: -INTERNAL instruction, XXX, was not a number. 8.2
8.4. How can you generate descriptions without introducing failures? 8.2
8.5. How to minimize the number of bugs? 8.2
8.6. How to debug? 8.3

CHAPTER 9. EXERCISE. 9.1

9.1. What is the meaning of this final exercise chapter? 9.1

APPENDIX A. ANSWERS TO EXERCISES. A.1

CHAPTER 1. PHILOSOPHY.

1.1. Introduction

The most important factors for succes in today's competitive environment are reduced time to market and attention to customer-specific needs. These factors are realised through responsiveness adaptability and flexibility. Especially the flexibility is of great importance to a company because it results in short production runs, reduction in throughput time and a greater product range leading to a higher efficiency and higher profits.

In mechanical design, there are several problems which need to be adressed in order for companies to bring new products to market quicker while being responsive to customer-specific needs, which include:

- Repetitive designs. Even though similar products are designed from similar components, analyzed and produced in similar ways, each individual design is treated as a new event.

- Expertise is scattered. In most companies, knowledge about designing and manufacturing products is scattered throughout the organization. This situation can lead to high engineering and manufacturing costs, slow response to customer requests, bottlenecks caused by the unavailability of a few key people, and loss of knowledge when key people retire.

- Non-concurrent engineering. Manufacturing and engineering departments are usually physically separated and the design process proceeds sequentially through each group. The knowledge and experience of the manufacturing engineers and process planners are often not considered until the design is essentially complete. Because of this, the design process can be lengthy when changes are needed to accomodate manufacturing criteria.

- Improving quality is difficult. Because the optimization phase of the design is a time consuming effort, improvements in quality are often left incomplete in order to bring products to market quickly.

Knowledge-based engineering is an advanced technology which addresses the need for companies to bring new high quality products to market quickly while adapting to customer-specific design needs. Knowledge-based engineering systems automate the design and engineering process by providing a means of storing product design information as a set of engineering rules for generating product and process designs automatically.

1.2. What is Knowledge-based Engineering?

Knowledge-based engineering technology provides a means of storing product or process information as a set of engineering rules and requirements for generating designs or process plans automatically. With conventional CAD systems, the user interactively creates and manipulates the geometric design data. The result is a geometric representation of a part which can be viewed or altered by interactively modifying the geometric data directly. The engineering which precedes the design is performed off-line, and when changes are made to design specifications, the engineering must be reevaluated manually before the designs are again manually changed.

Unlike traditional CAD systems that capture geometric design information only, knowledge-based systems like ICAD can capture the intent behind the product design. With knowledge-based engineering systems, engineers build a model of a product or process which is used to create the geometric design. When product specifications are changed or new versions of the product are desired, the rules in the model evaluate the new specifications and a new geometry (instance) is created automatically. A representation of a product model is shown in the figure below.

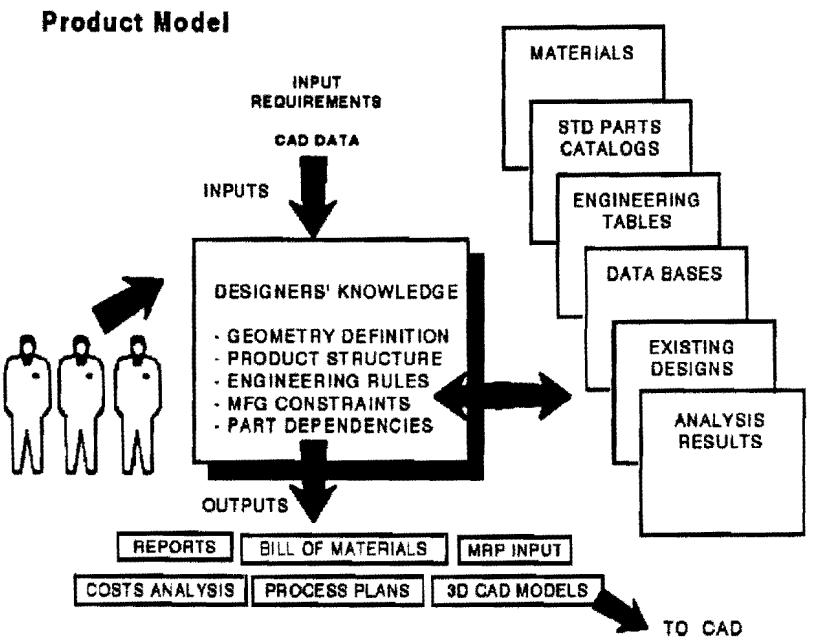

Figure 1: The Knowledge-based Engineering Process.

The product model is a means of integrating the knowledge from many organizations, departments and people which contribute to the creation of the company's products. Using product models, engineers can evaluate design alternatives quickly and generate new designs automatically. Since design engineers' knowledge is captured in the model, the manual intervention of the designer is eliminated, and designs are recreated automatically given a new set of input requirements.

The knowledge-based engineering product model must include:

- The product structure.

The knowledge-based engineering product model can include:

- Rules for altering the product structure given a new set of input requirements.
- Dependencies between parts of the product, such that if one part or an attribute of a part changes, other parts or attributes which depend on it change automatically. Not that the dependencies are created by the designer or engineer.
- Engineering design rules from multiple engineering groups which contain the knowledge for creating the part geometry or process information automatically from a given set input specifications.
- Engineering design rules for optimizing the product or process for cost, performance, and quality.
- Decision criteria for extracting information from external data bases, as with selection of standard parts from catalogs, material properties from material tables, or parts from feature-based design libraries.
- Rules for interfacing with and analyzing results of engineering analysis programs.
- Rules for using geometric design data which is imported from a CAD system as a geometric design constraint. In addition, the resulting geometric data which is generated from the product model can be sent to a CAD system for further completion of the detailed design, generation of NC programs, and documentation.

1.3. What benefits do knowledge-based CAD systems give us?

The benefits of Knowledge-based engineering to a company are: reduced time to market, capturing of the engineering knowledge and the ability of concurrent engineering. All these benefits are caused by the capturing of all the needed engineering knowledge into one integrated product model.

1.4. What disadvantages are there to knowledge-based CAD systems?

The disadvantages of Knowledge-based engineering to a company are:

- The high efforts of time and money that have to be made to make a product model. All the knowledge which is scattered throughout the company has to be gattered; this will take a lot of time and highly educated people.
- It is hard to predict the break-even point for knowledge-based systems. You can not point exactly where the benefits will be. In this way it is hard to convince managers of the need of these systems.

1.5. What kind of applications knowledge based CAD systems can be used for?

There are different classes of applications suited for knowledge-based engineering solutions. One type of application is the design and engineering of a family of products. The products need not have a constant topology, since rules can be built into the model to vary the product's topology from a given set of input specifications. Characteristics of these applications may include: many design rules, dependencies between parts of the product, redundant designs, catalog look-up, input requirements from existing CAD systems, or complex iterations to optimize the product for cost, performance or quality. An example of this type of applications is family of motorbikes or cars.

Another type of application is the design of complex, one-of-a-kind products which have long design cycles due to factors such as many design iterations and multiple engineering disciplines involved in the design process. A good example that falls into this category is a jet engine turbine blade. Prior to using the ICAD system it was not uncommon to the number of design iterations to approach 50 or more, with process taking about 6 months to complete. When one integrated model was used for all engineering groups to work with, the iteration process was shortened from weeks to minutes. In addition the engineers get immediate feedback on the result of a design change, and can more easily see its effect on other parts of the product.

CHAPTER 2. GNU EMACS EDITOR AND SYSTEM COMMANDS.

2.1. What is the purpose of this document?

The purpose of this document is not to present you a fully described user manual, because there exists one already, see 'Introduction to GNU EMACS on the Sun'. Then why is there a need for some additional paper? We believe, that this document helps you to find your way quickly throughout the ICAD-system.

2.2. What you have to know before you even touch the SUN system?

The system you work on is a SUN Sparc workstation. This computer is different from computers like personal computers. An important difference is, that you never turn on or shut off the SUN computer and SUN monitor. Only the system operator is allowed to do this.

Another issue is that of PATIENCE. Though this computer is very fast, it can happen that after you have given a command, it takes some time before something will happen. Especially in the beginning you should learn to wait what is going to happen. Never start pushing randomly on all the keys you can find, because you think something must happen. It might happen, that you have given a wrong command or that you believe you have given a command. In these cases there is always a way to turn back properly. In worst case you have given a command that causes a 'deadlock', which results in a 'deaf and blind' computer that ignores everything you tell him. When this occures, don't pray but get some help of the system manager.

2.3. How to enter the SUN system?

SUN computers belong to computers that deal with the UNIX Operating System. For this reason you have to log in with a log in name and a password. Be sure to obtain one from your instructor. When you first sit down you see a "login:" prompt on the screen. Now the following actions have to be taken:

1> **Type your log in name, followed by [RETURN]-key.**

2> **Type your password, followed by [RETURN]-key. Notice that your password is not shown on the screen.**

What happens then? The screen turns grey, showing you that the system is starting SunView (the Sun windowing system), and the ICAD system. This takes several minutes, because the ICAD system is being loaded.

EXERCISE 2.1:
Try to log in.

2.4. How to leave the SUN system?

The actions to be taken for leaving the system depend on the window in which you are working. This can be the GNU EMACS window, the ICAD window, the CMDTool window or the Console window. For this moment don't mind the CMDTool and the Console window. In the other cases you must first quit the window. What is meant by 'quit the window'? This mean that you must disable the window.

For quiting the GNU EMACS you place the Mouse Cursor at the top of the window (in the black header). Then you press and hold the right Mouse Button. A popup menu will be shown in which you can select the option 'quit window'. Select this option by moving the Mouse Cursor and releave the button. You are now asked to confirm the action. Move the Mouse Cursor to this window and use the left Mouse Button to confirm.

For quiting the ICAD Browser the Mouse Cursor has to be placed on the ICAD symbol, which can be found in the top of the window, on the left of title 'The ICAD Browser' and above the '?'-symbol. Then push the key 'stop'; this key can be found on the top left of the keyboard. When you quit this window, it is likely that the GNU EMACS window is still active. Quit this window as described above.

Now you have to place the Mouse Button on the grey area around the windows. Press and hold the right Mouse Button. This pops up a menu of commands.
With the right Mouse Button still depressed, drag the mouse downward until the command Exit SunView is highlighted. Then release.
You are asked to confirm this action. Click on the left Mouse Button. Then the screen shows the 'login:' prompt again.

EXERCISE 2.2:

\# Try to log out, when you are already logged in. Otherwise, experiment with the log in and log out routines.

2.5. Can Sunview be left temporary?

Yes, this is possible. The meaning of this option is, that you can walk away from the system to get a cup of coffee, without the possibility of anyone damaging your files. When you log in again, you will find all the windows as they were before you left the system.
How can you leave the system temporary?

Well, you have to place the Mouse Cursor on the grey area as described before and turn on the popup window. Now you do not select the 'Exit Sunview' command, but select the 'Services' command. A new pop up menu will be shown from which the 'Lock Screen' command can be chosen. SunView will show you now the Lock Screen. When you want to continue with the SUN system, you only have to enter the password. This option prevents anyone else from using the Sun system.

EXERCISE 2.3:

Log in if you are not logged in, and execute the 'Lock Screen' command.

2.6. What do you have to know about GNU EMACS Editor?

Everything about GNU EMACS Editor can be found in the manual 'Introduction to GNU EMACS on the Sun'. A lot of the subjects described in the mentioned manual are not needed to make your first ICAD program. Here the most important aspects are presented. These aspects are the ones that you probably will use most often.

2.7. For what is GNU EMACS editor used and how is it working?

This editor is used for creating defparts. Defparts are objects, which can be presented in ICAD Browser. Examples are houses, tables, cars, machines. More about defparts will be presented in the next chapters. The description is done by using the ICAD Design Language IDL, which is based on the LISP-language. Don't be frightened, we will not put you on a LISP-course. In next chapter you will learn more about defparts, LISP and IDL, most of them from examples. After you have described defparts you have to check the description. This means that you have to compile your description. When this is OK to the computer, you can switch to ICAD Browser, from where you can draw your objects.

2.8. How to get descriptions of defparts in the EMACS Editor?

The description of defparts can be found in existing files or can be created as new ones in the editor. First, let us assume that there are already existing file on disk. Then these first have to be put in the editor. This is done by the command:

Ctrl-x Ctrl-f
(This is: push and hold the 'Ctrl'-key, and push sequentially 'x'-key en 'f'-key. You will see at the bottom of the EMACS window, that a prompt has appeared that asks for a filename.).

Type the name of the file followed by the [RETURN]-key.

See also page 1-12 and page 1-15 of the GNU EMACS Editor introduction manual. Also, when you want to make a new file (with a defpart) then you need this command.

Another way of loading an **existing** file is done by using the Mouse. Put the Mouse Cursor on the Text area. Then press the right Mouse button. A popup menu will be shown. Move the cursor to the command '**Dired**' and select it by pressing the left Mouse cursor. A list with files is shown. Move the Mouse cursor to and hold it on the file name which you want to load. Select this file by pressing the **left Mouse button**. Then press the **'f'-key**. Your file will be loaded.

What has happened after your text is shown? The computer has put the existing file in a buffer (temporary file) and this buffer is shown in the Text area. The changes to this buffer will not affect

the existing file on disk. Remember therefore, that you always **save** the buffer at regular times.

2.9. How to save the buffer?

Saving the buffer is done by moving the Mouse Cursor to the Text area, and pressing and holding the right Mouse Button. A popup menu will be shown with commands. One of them is 'save buffer'. Select this one. The question 'Save buffer as ... [name]...: Y/N (yes/no)'is shown in the mini buffer, the area beneath the Text are in the same EMACS window. Answer this question by pressing the 'y' or 'n' key.
You can also save the buffer with the commands:

Ctrl-x Ctrl-s
Saving it to the file with the same name.

Ctrl-x Ctrl-w
Saving it to the file with a new name.

See also, page 1-14 of the introduction manual.

The way in which you describe the defparts can be found in plenty examples around you. Let us view the most basic commands of editing the text.

2.10. How to edit a text in the Text area?

To edit the text you have to put the Mouse Cursor in the Text area. Click on the left Mouse Button to make sure of that you are really working in the Text area. You will see a black text cursor. At the place where this cursor stands, you can delete or add text. The easiest way to move around the Text area with this cursor is by moving the Mouse Cursor and clicking the left Mouse Button at the place where you want to edit.

Adding and deleting text is not the same as in most other Text editors. You have to get used to it. We recommend you to experiment with it.

See page 1-13, page 1-21 to 1-23 of the introduction manual for editing commands. The most important commands are:

- for **adding** text : just type characters at the desired place in the area
- for **deleting** text: use the [delete]-key on the keyboard.

When you add text you will probably use the [RETURN]-key at the end of a sentence to go to the next line. What happens then, is that the text cursor goes to the beginning of the next line. In stead of using the [RETURN]-key, you can use the [LINEFEED]-key.

EXERCISE 2.4:

Load the file 'tryout1.lisp' in the buffer.
Change the first line from 'file tryout1' into 'file tryout2'.
Save the file with the name 'tryout2.lisp'.
Load the file 'tryout1.lisp' again to see if this text has changed.
Load the file 'tryout2.lisp' to see if you have created a new file.
Clear the buffer.

2.11. Where to find a summary of all editing commands?

For more information of the editor and a summary of all the editing commands we refer to GNU EMACS Quick Reference, page 1-26 to 1-36.

2.12. What to do after the text has been edited?

After editing the text you are of course curious of what the result is of your created defparts. The first thing to do is to compile your buffer text. After you do so, do not forget to save the buffer: it is not a bad habbit to do this always before of just after you have compiled the text. When you compile the text, it is essential that your text cursor is in the text area, where your defpart is described (this applies too in case of saving the buffer). Now you use the command:

META-SHIFT-B
press all these keys at the same time; remember that de [META]-key is the diamond-key on the left of the spacebar.

What is going to happen next, is that your text is being compiled. When syntax errors exist, these will be presented in the Compilation area. This is the area beneath the Text area. Remember, that when no errors are presented, this does not mean that your text (code) is without errors.

If errors exist you will find out that about 75% of these are created by false or forgotten brackets. See our chapter 'Tricks special'. In case that there are no errors, you can switch to the ICAD Browser.

CHAPTER 3. ICAD DESIGN LANGUAGE AND COMMON LISP.

3.1. What is IDL?

The programming language you use is called IDL, ICAD Design Language. This language is based on the Common LISP programming language. Therefore, before we can tell you something about IDL, we have to tell something about Common LISP. It is not our intention to learn you everything about this language, but there are still a few things you should know.

3.2. What kind of language is Common LISP?

Common LISP is a Object Oriented Programming language. The purpose of this language is, that programming is made more comfortable, so that even Mechanical Engineers are able to construct programs. How must you imagine this. Compared with other languages like Fortran, Basic and Pascal, the way of programming is different. The way you would describing things in the real world, is the way to describe these things in LISP: you see objects in the world around you, so you describe objects and their functionality in LISP. You describe things in a natural way, but this does not mean you can describe these things in English or Dutch. For this reason, you have to know some things about object oriented languages.

3.3. What do you have to know about Object Oriented Languages?

All the things you describe are called objects, for example a 'BMW bike' is an object. These objects are part of classes, for example the 'BMW bike' is an object belonging to the class 'conveyance'. Classes represent generic characteristics of objects. Each object contains all knowledge about itself. The object can have interactions with other objects, for example with a man, who is riding on this bike.

Now there is coming a difficult one. We talk about object instances which define actual objects for which specific values have been supplied as inputs, for example we talk about object 'BMW bike', but the instances (=actual object) of these objects could be a BMW R80-type or BMW R100-type. The process of creating an object instance from a class definition is called instantiation.
Classes of objects are defined; this is the same as 'talking about things'. With instantiation object instances are made, and this is the same as 'the being there of the things'.

Let us consider ICAD. In de EMACS Editor we 'talk about things', in other words we describe **defpart elements** (see further on). When we move on to ICAD Browser and represent our defparts then we create actual objects or instances with a Makepart-function.

The things you describe have characteristics. For example the BMW R80 has two cilinders, each of them contains 400 cc, a R100 has two cilinders, each of them contains 500 cc. The number and the contents of cilinders we can call attributes. Atrributes provides a means of attaching information to a part. In LISP we can define attributes in many ways: in terms of other attributes, information contained in a catalog, arithmetic expressions or conditional expressions. Attribute values may be

given as inputs or depend on the value of an input creating "generic parts".

With this above in mind, we are now able to describe mechanical systems in a hierarchical way, in the same way as we would describe parts of subassemblies of subassemblies of ..... and so on. Until sofar we described Object Oriented Languages, but we did not tell how to use and program in Common LISP.

3.4. What rules are to be considered in Common LISP?

Common LISP defines several kinds of elements, three of which are described here:

- A series of letters and digits defines a symbol. There may be any number of letters and digits in its name. "Words" in symbol names are usually seperated by hyphens. A **keyword** is a symbol whose name begins with a **colon**. Here are some symbols:
  abcd x15 123q :length

- A series of digits, with an optional decimal point, defines a **number**. Fractions (by themselves) are legal numbers too. Examples:
  800 3.26 1/3

- A balanced set of parantheses defines a **list**. The other objects in the list are called **elements**. An element of a list may be a list itself. This is called a **nested list**. Examples are:
  (a b c) (:length 6)

3.5. What are expressions in Common LISP?

An **expression** is the basic result of computation in Common LISP. All LISP programming consists of writing expressions. Another word for expression is **form**.

An expression can be:

- A **constant**, which is a number, a keyword, a quoted symbol, or a quoted list.
- A **variable**, which is a unquoted symbol.
- A list, whose first element is a symbol naming a function, and whos remaining elements are expressions. Examples of functions are: + - / * cos tan. This is the most common expression.
- A list, whose first element is a symbol naming a macro, and whose remaining elements are determined by the documentation of the macro. This usage is less frequent, except for the macro named '**defpart**'.

An expression ALWAYS RETURNS A RESULT, which is the answer from the expression. They are used for computation in many aspects of IDL. Parentheses organize expressions.

Examples of expressions are:

- (+ 36 57 89)
- (+ (the :height) 6)

An important class of expressions are the conditional expressions, which are not shown above.

3.6. What about conditional expressions?

Conditional expressions are the means by which rules are incorporated into the definition of an object. The term 'conditional' means that part of the expression is evaluated only under certain conditions. An IF-THEN statement is a conditional expression. IF is a LISP conditional operator, which is used to choose between two alternatives.

Structure of conditional expression:

```
(if (test expression, which returns t or nil)
    (then-expression, which executes if the test expression is true)
    (else-expression, which executes if the test expression is nil)
)
```

More conditional expressions are known in IDL, like '**ecase**'. Look up in the ICAD manual's what other conditionals are known. Some of these conditionals you will need for the exercises in chapter 'EXERCISES'.
All LISP conditionals depend upon a test expression succeeding or failing. Any expression may be used as a test expression. An expression which returns the symbol NIL is FALSE. An expression which returns anything except NIL is TRUE. A convential non-NIL result used when there is no other meaningful result is the symbol T. An expression which returns TRUE or FALSE is called a '**predicate**'.

Remember that a conditional cannot be used to choose between alternate keywords: it cannot choose between :front and :rear. It may be used everywhere in IDL where an expression is required. For example, a conditional can be used to specify the value of an attribute.

3.7. What predicate functions are available for conditionals?

There are predicates used to test for equality.
- 'Eql' is used to determine whether two symbols are identical. The function 'eq' is identical, but 'eql' is preferred.
- '=' is used to determine whether two numbers are exactly equal. Both arguments must be numbers.

Many other predicates are available. The names of these functions are often named ending with a 'p' or with a '?'. The ones that end in '?' are usually ICAD-defined. These predicates are described in Chapter 5 of the User's Manual. Here are some of the Common LISP and IDL predicates.

LISP predicates:

<	**plusp**	**string-equal**
>	**oddp**	**string-lessp**
<=	**evenp**	**string-greaterp**
>=	**zerop**	**between?**
/=	**minusp**	

You can combine test expressions with 'and' and 'or'. You can invert the meaning of a test expression with 'not'.

3.8. Where you do and where you do not use conditionals?

Normally you use these expressions at places of values of attributes, default values of optional-inputs and defaulted-inputs, supplied values in part specifications, and amount of positioning offsets. Places where these expressions less frequently used, are: orientation keyword values and types of parts. Places where these expressions are **never allowed**, are: orientation keywords, positioning keywords, mixin list and defpart keywords, because they are for values and not for keywords.

Examples of expressions with their results are:

- *(eql :abc :abc)* *result: TRUE*
- *(not (eql :abc :abc))* *result: FALSE*
- *(= 3 (+ 1 2))* *result: TRUE*
- *(string-lessp "Abott, M." "Zane, B.")* *result: TRUE*

Example of the usage of the IF-statement:

```
(defpart office-chair (box)
    :inputs
        (:owners-job-title
        )
    :attributes
        (upholstery-material
            (if (or (eql (the :owners-job-title) :president)
                    (eql (the :owners-job-title) :vice-president))
                :leather
                :vinyl)
        )
 )
)
```

3.9. What additional LISP functions are used?

Next additional LISP functions are used.

- **max,min** Returns the maximum or minimum of the given numbers, for exmaple: (max (the :length) (the :width)).
- **half** Divides the given number by 2.
- **div** Divides two numbers, yielding a floating point result.
- **twice** Multiplies the given by 2.
- **round** **truncate** **ceiling** These four functions perform integer division when given two numbers, and convert a floating point number to an integer when given one number. The only

	floor	difference is in their 'rounding' behavior.
-	**degree**	Returns radians, given degrees: (degree 30) → 0.5235995
-	**sin,cos,tan**	Returns the sin, cos, or tan of the given angle; where the angle is given in radians. Example: (sin (degree 30)) → ½
-	**expt**	Raises a number to a power: (expt 3 2) → 9
-	**sqrt**	Returns the square root of the given number.

EXERCISE 3.1:

\# What type of elements are next elements of?

a. -32
b. BMW-with-sidecar
c. 15 x 15
d. (-32)
e. (+ 4 5 6)

EXERCISE 3.2:

\# What are the LEGAL and ILLEGAL expressions?

a. ((+ 3 4))
b. (3) (4))
c. (3 + 4)
d. brick
e. (+ ((+ 36 57) 89))
f. (sin 30)
g. :brick
h. (* :height 6)
i. (* (:height) 6)
j. (* (the: height) 6)
k. (* the :height 6)
l. (+ 5 (* 2 2 2) (+ 7 3))
m. sin(30)
n. (+ (5 (* 2 2 2) (+ 7 3)))

EXERCISE 3.3:

\# What are the results of next expressions:

a. (eql :three 3)
b. (between? 2 5 5.1)
c. (>= (* 2 9) 8)
d. (oddp 8)
e. (or (not (eql :abc :abc)) (oddp 8))

EXERCISE 3.4:

What does the defpart description from the example of the usage of the IF-statement on page 3.4?

3.10. What does IDL consist of?

'Programming' in IDL consists of:

- Defining descriptions of objects using defpart. This is done using the EMACS Editor window. Defparts are stored in files.

- Creating objects from their description using the Make Part command. This is done using the ICAD Browser window. They are used to get results: drawings, reports, data files, etc. Objects can be stored.

3.11. What is a defpart?

A defpart consists of a name, mixins, attributes (and other features). A name is always a symbol. Attributes define information about these parts. Mixins define the 'kind-of' relationship, this is also called '**inheritance**'. These mixins may be defined als defparts, though often only 'box' is used. Each mixed-in defpart require values for certain attributes. For instance a box requires a length, width and heigth. Let us see the next example (pay attention to the Lots of Insane Silly Parentheses).

Example of a defpart:

```
(defpart house (box)
      :attributes (:length 30
                   :width  25
                   :heigth (the :width)
                  )
)
```

house is a name;
(box) is a mixin list;
:attributes is a defpart keyword;
:heigth, *:length* and *:width* are attribute names;
(the :width) is a attribute value (expression).

3.12. What can you do with the ICAD Browser?

The ICAD Browser is the tool you use to show your instantiated defparts. Every time you start up the Browser your first action is to use the 'Makepart'-function. With this function you instantiate a object of your defpart definition. How you should use the remaining functions will be explained in one of the next chapters. You can imagine, that after the object has been presented you do not

like this version of the object and do want to change the definition. To do so, you have to stop and leave the ICAD Browser and recall the EMACS Editor. This can be done several times of course.

3.13. What should you have learned until now?

Until sofar you should have been become a little familiar with object oriented languages, the language Common LISP and IDL. You should know what the characteristics are of this knowledge based engineering tool. Also should you know what has to be done to create defparts, instantiate the objects and the possibility to change the defpart definitions. Next chapters will explain in more detail how to create defparts and how to use the ICAD Browser.

CHAPTER 4. DEFPART ELEMENTS.

4.1. How to construct an IDL-description?

Most of your programs will have the next structure:

Structure:

```
(in-package 'IDL-User)

(defpart defpartname1 (primitivename1)
...
)

(defpart defpartname2 (primitivename2)
...
)
```

For this course you always start with the definition: **(in-package 'IDL-User)**. The computer has to know that you are an ICAD-user.
In every IDL description the name **defpart** is used to tell ICAD that a new defpart will be defined. **<defpartname1>** and **<defpartname2>** are free to be chosen. **<primitivename1>** and **<primitievename2>** are primitives like <box>, <cylinder>. These names are system defined. Other then these primitives are user defined primitives. At the place of <...> several expressions are defined, which will be mentioned in the rest of this chapter.

4.2. What about :attributes?

An **attribute** is a way to associate any piece of information or knowledge with a particular defpart. It has two parts:
- Its name, which is always a keyword.
- Its value, which is always defined by an expression. the actual values of the attribute is the result of the expression.

Example of attributes:

```
:attributes
(:length 20
 :width  (* 2 40)
)
```

4.3. Which specialities of :attributes are known?

There are some specialities.

1. The primitives (box, cylinder, etc.) have system-defined attributes. For box the attributes **:length, :width, :height** are defined.

2. Other attributes names then the defined ones are allowed. Notice that these names are constants.

3. There is no specific ordering necessary with respect to the way in which the attributes are defined.

4. Instead of (* 2 40) other expressions are allowed. One of the used expressions will be shown at '**What about reference chains?**'.

4.4. What are :parts?

When we look at a house we could distinguish several parts like the first-floor and the roof. We could describe this house in IDL-code:

Example 'House':

```
(in-package 'IDL-user)

(defpart house (box)
      :attributes
      (:length 30
       :width  25
       :height 20
      )
      :parts
      ((first-floor  :type box)
       (roof         :type roof)
      )
)

(defpart roof (box)
      :attributes
      (:height 3
      )
)
```

We see these parts create a **tree structure** of a house. All these parts have a name, which is a non keyword that will appear in the tree structure, and a type, which is a symbol naming a defined defpart. It is obvious that a house could have more parts then just these two. Like this house, every defpart represents an assembly, and the parts are sub-assemblies. Each defpart therefore specifies one level of the product structure tree. So a two-level tree requires at least one user-defined defpart. An **N-level** tree requires **N-1** user defined defparts. You will be writing lots and lots of defparts to

build your product's hierarchical structure.

A few remarks have to be made:

1. Instead of '(roof :type roof)' we could write (roof).

2. :Type is pronounced 'is a', as in 'The first-floor is a box.' or 'The roof is a roof.)

3. Notice that however you use the name :Parts this not means, that you have to describe more than one part.

4.5. What about generic parts?

A fully generic part is a defpart all of whose abstract parameters are either inputs or attributes whose **values depend** on those inputs, or are constants. These parts are easy to reuse, duplicate and move around in the tree. 'Genericness' is a goal to strive for. A given defpart may be generic in certain respects, and not in others. Generic parts define a class of objects that represent all possible configurations for that part. By giving inputs to the generic part, you define a particular instance of the class.

4.6. What about :inputs?

Inputs are used to tell the defpart that there has to come some information from outside. Without this information you can not draw an instance of your defpart on the screen. So where does this information come from? There are a number of possibilities. One of them is the next one. For example your defpart definition is something like this:

Example of inputs:

```
(in-package 'IDL-User)

(defpart dice (box)
      :inputs
      (:dice-dimension)
      :attributes
            (:height (the :dice-dimension)
             :length (the :dice-dimension)
             :width  (the :dice-dimension)
            )
)
```

(Do not pay attention to the 'the' expression, this will be explained later at the "**What about reference chains?**").

In the ICAD Browser you choose now the '**Makepart**'-function. What will happen, is that ICAD presents you a window, called 'inputs'. ICAD expects from you, that you enter the :dice-height. If you do not type in a number, then ICAD can not present you your nice dice. Do not forget to use

the '**accept**'-function in the window, to accept your input.

Inputs are the mechanism which make classes of parts interesting. The defpart describes the characteristics of the class, without any specific values for them. This is nice, because otherwise you have to go back to EMACS Editor to change the values if you want to see a dice with other dimensions.

Another form of input is the optional input.

4.7. What about :optional-inputs?

These are inputs for which a **default value** is supplied, that means, a value is defined in the defpart definition. This value will be shown at the moment that you create an instance in the ICAD Browser. If a value for the input is not supplied in the definition of the parent, its value will be the default value provided in the child. The default value is only **overridden** when the value of the input is given in the :parts list, or supplied in the 'Make Part' command.

Syntax of :optional-inputs:

```
:optional-inputs
    (:name-1 expression-1
     :name-2 expression-2
     ...
    )
```

Some remarks have to be made:

1. Inputs have a name and an expression.
2. The name is specified in the current defpart. The value is specified in the next higher level.
3. Values of inputs are specified in:
   - the Top Level Inputs (for the root node);
   - the :parts section of a defpart (for all other nodes). :Parts and nodes are explained further on.
4. Many attributes of primitives (i.e. :height and :width for box) are automatically inputs.

4.8. What other ways are possible to ask for input?

The way in which input is asked, is called the user interface. This can be done as described above. Another way to do this, is to use the user interface type, for which you use spec-sheets en spec-attributes. An example will be presented in one of the exercises of chapter 'EXERCISES'. If you want to know more about these user interfaces, please see **the ICAD Production UI Toolkit manual**. For your work you will probably need this info, so do not forget to read it!

EXERCISE 4.1:

Create a defpart like described here, compile it and use Makepart. Describe what happens.
Save this defpart description as: house1.lisp

```
(in-package 'IDL-User)

(defpart house (box)
    :inputs
        (:ceiling-height :material)
    :optional-inputs
        (:wall-thickness 1)
    :attributes
        (:height (twice (the :ceiling-height)))
    :parts
        ((first-floor   :height (the :ceiling-height)
                        :wall-thickness (the :wall-thickness)
                        :position (:bottom 0))
         (roof          :type box
                        :height (the :ceiling-height)
                        :position (:top 0))
        )
)
```

4.9. What about the tree structure?

If someone would ask you to describe the family you belong to, you problably would model your family by drawing a hierarchical tree, and you would use names like father, mother, parents, ancestors. The same we can do with products. In ICAD part descriptions are used to **name relations between parts**:

Tree structure:

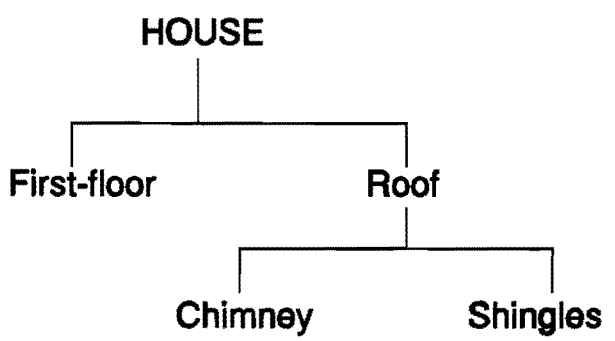

Figure 2

NODE:

An object in the tree which represents a part. The tree in the figure has five nodes.

CHILD:

A node which is immediately 'below' another node. First-floor and Roof are the cildren of HOUSE. Singles and Chimney are the children of Roof.

PARENT:

A node which has other nodes immediately below it in the tree. House is the parent of First-floor and Roof. Roof is the parent of Shingles and Chimney.

SIBLINGS:

Nodes which have the same parent. First-floor and roof are siblings, and Shingles and Chimney are siblings.

ROOT:

The single node in the tree which has no parent. HOUSE is the root in the model.

LEAVES:

The nodes in the tree which have no children. First-floor, Shingles and Chimney are the leaves in the model.

DESCENDANTS:

Those nodes which are directly below another node. First-floor, Roof, Shingles, and Chimney are the descendants of HOUSE; Shingles and Chimney are the descendants of Roof.

ANCESTORS:

Those nodes which are directly above another node. HOUSE and Roof are the ancestors of Shingles and Chimney. HOUSE is the ancestor of First-floor.

SUB-TREE:

A node and its descendants. Roof, Shingles, and Chimney form a sub-tree in the model.

EXERCISE 4.2:

\# Try to draw the tree from the text here below is given:

```
(Defpart Robot (box)
    :inputs
        (:robot-height)
    :attributes
        (:length (* 0.1 (the :robot-height))
         :height (half (the :robot-height))
         :width  (* 0.3 (the :robot-height)))
    :parts
        ((torso      :type box)
```

```
            (head       :type box
                        :height (* 0.15 (the :robot-height))
                        :width  (* 0.15 (the :robot-height))
                        :position (:above 0))
            (left-arm   :type arm
                        :robot-height (the :robot-height)
                        :position (:on-left 0 :top 0))
            (right-arm  :type arm
                        :robot-height (the :robot-height)
                        :position (:on-right 0 :top 0))
        )
)

(Defpart Arm (box)
    :inputs
        (:robot-height)
    :attributes
        (:height (* 0.4 (the :robot-height))
         :width  (* 0.1 (the :robot-height)))
    :parts
        ((upper-arm  :type box
                     :height (* 0.8 (the :height))
                     :position (:top 0))
         (hand       :type grabber
                     :robot-height (the :robot-height)
                     :height (* 0.2 (the :height))
                     :position (:bottom 0))
        )
)

(Defpart Grabber (box)
    :inputs
        (:robot-height)
    :parts
        ((palm      :type box
                    :height (half (the :height))
                    :position (:top 0))
         (finger    :type box
                    :height (half (the :height))
                    :width (* 0.05 (the :robot-height))
                    :length (* 0.05 (the :robot-height))
                    :position (:bottom 0))
        )
)
```

4.10. What about inheritance?

There is something you should know about inheritance. See next two descriptions:

Example inheritance:

```
(defpart  wall (box)
    :attributes
        (:height 8)
    :parts
        ((beam :type box
        :length 2
        :width  4
        :height (the :height)
        )
    )
```

```
(defpart  wall (box)
   :attributes
     (:height 8)
   :parts
     ((beam :type box
          :length 2
          :width  4
     )
  )
)
```

These examples tell you, that you do not have to specify the height of the beam, when it has the **same** height **as the wall** itself. This is called inheritance. Not only :height, but :width and :length have this behaviour too. Other attributes do not.

4.11. What about reference chains?

You already have seen reference chains and the usage of it, probably without realizing that it were reference chains. What is a reference chain and what is it used for, are questions we will answer here.

Example reference chains:

```
(defpart Robot (box)
        :attributes
        (:height 8)
        :parts
        ((beam :type box
              :length 2
             :width  4
             :height (the :height))
        )
)
```

The macro called 'the', which is used in the :parts definition of the beam (the :height), **begins** a reference chain. In part 'beam' we say: 'The height of the beam is the same as the Robot's height'. We could also refer to the height of another part or subpart.

Syntax reference chain:

```
(the :partname ... :attribute-name)
```

If no :partname is given, reference is to an 'attribute' of the defpart in which 'the' appears (we have already seen this usage). If no :attribute-name, reference is to a part.

You must define the **explicit** path down the tree to a part or attribute. Defining the path means explicitly **naming** the parts down from the parent part that 'you' and the target part have in common. You do not name the parts in the path up to the common parent, nor the parent itself. This is what makes a reference into a reference chain.

You may refer to attributes of children, grandchildren or more distant descendants. It is not good practice to define the value of an attribute with a 'long' referencing chain. It is better to define an input, and use the 'long' referencing chain to specify the value from the part above. Using inputs makes the 'lower' part generic, since it does not use the referencing chain. This is also the way to build generic parts.

In the definition of house in the following example.

Example 'House':

```
(defpart house (box)
      :inputs
      (:ceiling-height :wall-thickness )
      :attributes
      (:height (twice (the :ceiling-height)))
      :parts
      ((first-floor      :height (the :ceiling-height)
                         :wall-thickness (the :wall-thickness)
                         :position (:bottom 0))
      )
)
```

The height of the house is defined as twice the ceiling-height, which is an input. The first-floor part refers to the definition of :ceiling-height defined by the house for the definition of its height. The first-floor part refers to the definition of :wall-thickness defined by the house for the definition of its :wall-thickness.

In the definition of the robot in the following example 'the head' refers to the definition of the left-arm for its width:

(the :left-arm :upper-arm :width), where

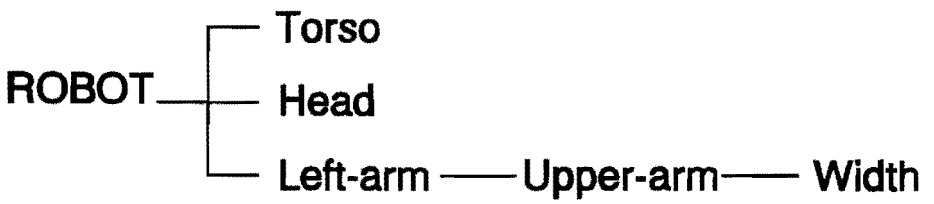

Figure 3

EXERCISE 4.3:

\# Generate next code and predict and show what happens.

```
(in-package 'IDL-User)

(defpart house (box)
      :inputs
      (:ceiling-height :material :wall-thickness)
      :attributes
      (:height (twice (the :ceiling-height)))
      :parts
      ((first-floor     :type first-floor
                        :height (the :ceiling-height)
                        :wall-thickness (the :wall-thickness)
                        :position (:bottom 0))
       (roof            :type box
                        :height (the :ceiling-height)
                        :position (:top 0))
           )
)

(defpart first-floor (box)
      :inputs
      (:wall-thickness)
      :parts
      ((front-wall      :type box
                        :length (the :wall-thickness)
                        :position (:front 0))
       (rear-wall       :type box
                        :length (the :front-wall :length)
                        :position (:rear 0))
       (right-wall      :type box
                        :width (the :wall-thickness)
                        :position (:right 0))
       (left-wall       :type box
                        :width (the :right-wall :width)
                        :position (:left 0))
      )
)
```

CHAPTER 5. THE ICAD BROWSER.

5.1. What do you have to know about the ICAD Browser?

Everything about the ICAD Browser can be found in the ICAD Browser User's Manual. A lot of the subjects described in this manual are not needed to start with. Here we present the most important aspects of the Browser. These aspects are the ones that you probably will use most often.

5.2. What is the ICAD Browser used for and how does it work?

The ICAD Browser is a program that lets you create and examine instances of defparts. Defparts are objects that are created and compiled in the GNU EMACS Editor. Examples of instances of defparts e.g. a house are: a villa, a two storey house, a castle, etc.

The Browser screen consists of menubars, a workspace and menu tabs. You can work with it by using the mouse and keyboard. In the figure below the Browser screen is shown.

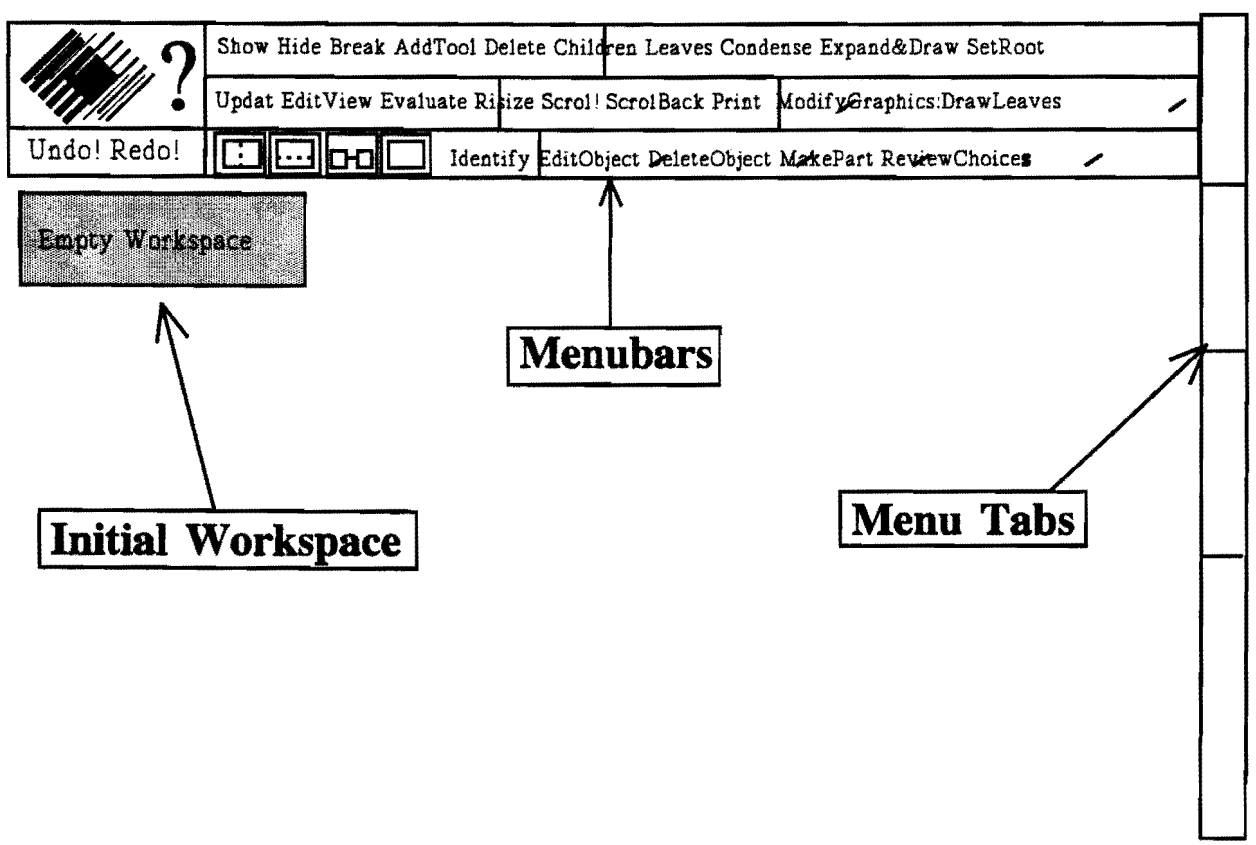

Figure 4

5.3. How to switch to the ICAD Browser?

To switch to the ICAD Browser your text cursor has to be in the Compilation area of the GNU EMACS Editor. This is the area where your compilation remarks are shown (See chapter about GNU EMACS Editor). If the text cursor is still in the Text area, you have to move the mouse cursor to the meant Compilation area and click the left mouse button. Then enter the command:

:CONT , followed by [RETURN]-key

Perhaps you will notice that some things started blinking in the ICAD Browser window. This means that the ICAD Browser is active. Now you just have to move your mouse cursor to the header of this window and click the left mouse button. The ICAD Browser is brought to the front of the screen.

5.4. How to quit and leave the ICAD Browser?

For quiting the ICAD Browser the mouse cursor has to be placed on the ICAD symbol, which can be found at the top of the window. Then push the 'stop' key, this key can be found at the upper left corner of the keyboard. The GNU EMACS window is active again. Select this window and it wil be ready to use again.

You don't have to quit the Browser to use the GNU EMACS Editor. You can always select it. If you haven't quited the Browser you are able to change and save the text in the GNU EMACS Editor but you won't be able to compile it, for this you have to quit the Browser.

5.5. How to work with the ICAD Browser?

Moving the mouse around, you can see that an arrow (the mouse cursor) moves around the screen in a similar fashion. To use the Browser, you will put the mouse cursor on various items and operate the mouse buttons. All three of the mouse buttons have a different function.

The left mouse button is to **select**.
The middle mouse button is to **match**.
The right mouse button is to **open a template**.

The meaning of the words select, match and open a template will get clear to you when you go on reading.

5.5.1. What to do with the menubars?

In this course we assume that you use a three-line menu because it contains all the options you can use. Is this menu shown at the top of the screen at this moment? If not, act as follows:

Move the mouse cursor to the ICAD logo. Select the menu you want to use by clicking the left mouse button. You may have to click a few times for this.

5.5.2. How to get an instance of a defpart on the Browser screen?

5.5.2.1. What do I use the MakePart command for?

You use the MakePart command to create instances of defparts. The MakePart command is found in the bottom line of the menu. If this command is chosen by clicking the right mouse button on it, you will notice that a template is opened almost similar to the one shown in the picture below.

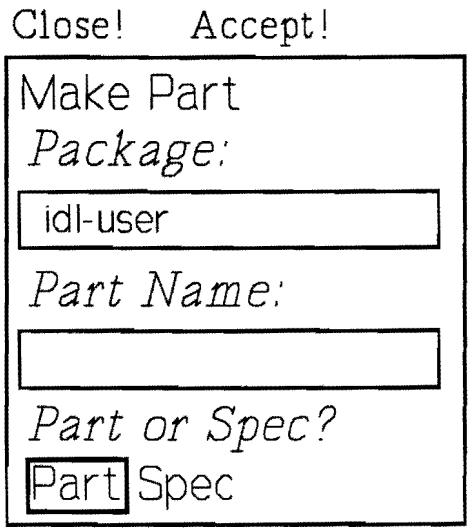

Figure 5

Lets examine it closer. The **Close!** command at the top of the MakePart template makes the template go away, without you changing anything. The MakePart template has two entries, Package and Part name. The Package is already specified. The entry for part name is blank. After you have selected this entry, you can type the name of the desired defpart you want to instantiate. When you have accepted this input, by selecting the **Accept!** command at the top of the template, you have to match the MakePart command with the Workspace.

This is how matching is done: first you have to select the desired command (e.g. MakePart, ?, EditView, etc.) by clicking the left mouse button on the desired command. You will notice that the command starts to blink to show that it is selected. Secondly you have to match the command with an object (e.g. Empty Workspace, Partname, etc.), this is done by clicking the middle mouse button on the object you want to match the command with.

EXERCISE 5.1:

Retrieve in the GNU EMACS Editor the file: house1.lisp.
Compile this defpart description.
Switch to the ICAD Browser.
Open the MakePart template, by clicking the right mouse button.
Select the second entry under the words "Part Name". (This places the text editing cursor (an

I-beam) in the text box, showing you where your text will be entered.)

Type house, followed by the Line Feed key and accept this input.

Match the MakePart command with the Empty Workspace.

5.5.2.2. How do I get the instance drawn?

In the previous paragraph you have defined the defpart that you want to draw an instance of. You will see that the workspace contains three aspects of this part, the Top Level Inputs, the Free Graphics Viewpoint and the product Structure Tree. See the figure below.

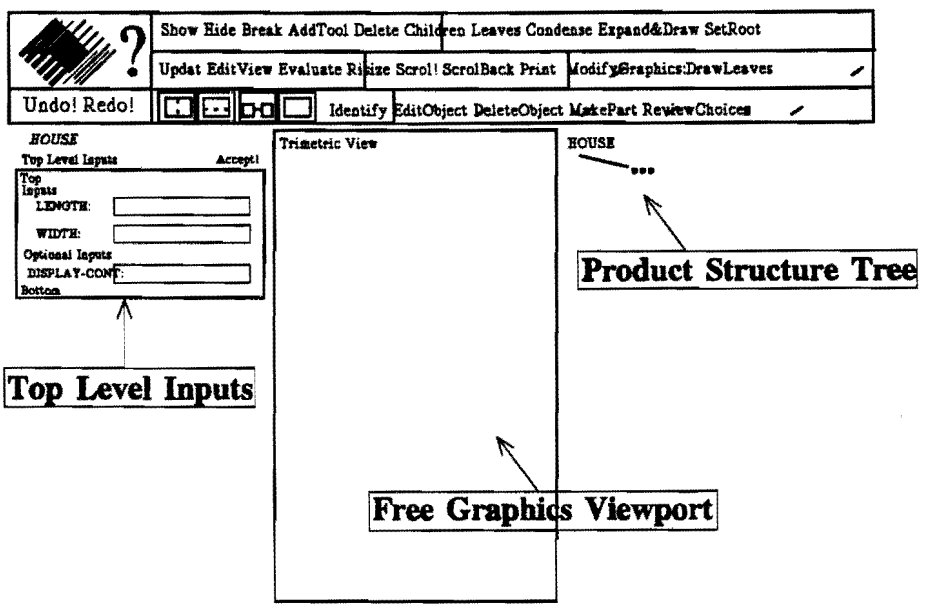

Figure 6

Top Level Inputs is a template showing you the list of inputs that the defpart requires before it can be fully instantiated. You can fill in this template, just like you filled in the MakePart template earlier. A house's inputs are Length, Width and Height. Other defparts have other inputs. Additionaly, all part have optional inputs. These inputs can be filled in but don't have to.

When you have accepted the settings the inputs are applied to the instance of the defpart. The Top Level Inputs template goes away whenever you use the Accept! command. You now have a fully specified "house" instantiated in the workspace. As you will see, when you use ModifyGraphics to show the geometry of your part, and EditView to show the Top Level Inputs template again.

Notice that the Product Structure Tree shows only the root node "House", followed by "...", showing that this part may have children.

EXERCISE 5.2:

Fill in the Top Level Inputs using any number you want. Don't forget the Line Feed key to signal completion of each entry. With your inputs specified, select the Accept! command at the top of the Top Level Inputs.

Select the ModifyGraphics command and match it with te Product Structure Tree. (The geometry appears in the Free Graphics Viewport.)

5.5.3. What are the leaves and nodes used for?

You can expand any node of a tree by matching it with the Leaves or Children command. Children shows only the immediate Children of a node, and Leaves shows all of the node's descendants.

EXERCISE 5.3:

Check whether the house has Children and Leaves.

5.5.4. What is the ModifyGraphics command used for?

Ordinarily, you use ModifyGraphics to draw the leaves of the matched node, then to Draw Node, Add Node, or Add Leaves. The ModifyGraphics command can be set to other drawing commands though. Once you set up ModifyGraphics, it does what you specified, until you change the set up.

EXERCISE 5.4:

Open the ModifyGraphics template.
Change the set up of the ModifyGraphics command and match the command with the workspace. (First predict what will happen.)

5.5.5. What is the EditView command used for?

With the EditView command you can change the Browser screen. EditView can be matched with the following browser objects:

1. Graphics viewport
2. A node of the Product Structure Tree
3. The entire Workspace (or browsable part)
4. The ModifyGraphics command
5. The Print command
6. Certain attributes in inspector windows

The first three possibilities will be explained in this course, for the other ones we refer to the ICAD

Browser User's Manual.

5.5.5.1. How to use Editview with the graphics viewport?

When you match the EditView command with the graphics viewport, a pop-up window will appear. This window shows you the ways you can edit the graphics view. Selecting **Cancel!** cancels the EditView operation with no changes made. Selecting any item in the list makes the appropriate change.

5.5.5.2. How to use EditView with the product structure tree?

When you match the EditView command with a node of the tree of your defpart, a pop-up window will appear. This window shows you the appropriate commands for that node.

If you select **Inspector**, a window is placed below the the node of the tree. The inspector viewport shows you all of the node's attributes and their values.

The inspector command is probably the one you will use most often. The explanation of the other commands is explained in ICAD Browser User's Manual and pages 1-55 to 1-57 of the Introduction Manual.

5.5.5.3. How to use EditView with the workspace?

You can match EditView with the entire browsable part to change the way you view the part. This makes that an EditView window pops up. Then you can select any command displayed. E.g. selecting the **Remove Graphics** command will wipe out the Free Graphics Viewport.

EXERCISE 5.5:

Try some of the options of the EditView pop-up window in combination with the Graphics Viewport.
Try some of the options of the EditView pop-up window in combination with the Product Structure Tree.
Try some of the options of the EditView pop-up window in combination with the Workspace.

5.6. Where to find more information about the Browser commands?

For more information about the Browser and all the commands that can be used we refer to the ICAD Browser User's Manual.

CHAPTER 6. POSITIONING AND ORIENTATION.

6.1. What is needed for building objects?

From the preceding chapters you know, that you have to define defparts, :attributes and :parts. But until now we have not discussed something important, namely the positioning and orientation of the parts with regard to the root (defpart). A lot of your programming time will be spend on how to position and how to orientate the parts. We think that this is one of the difficult issues of ICAD. For this reason we will spend some time on discussing these subjects. First we wil discuss the positioning, later we will discuss orientation.

6.2. What are faces and axis of a defpart and a part?

Assemblies are usually defined as a box in order to define a new local coordinate system with conveniently placed faces for positioning its parts, even if the assembly is not really box-shaped. This box may be a bounding box, but does not have to be.

In the next figure the bounding box with the axis and the faces of a reference part is shown. To these axis and faces a part can be positioned and oriented to in respect to the reference part.

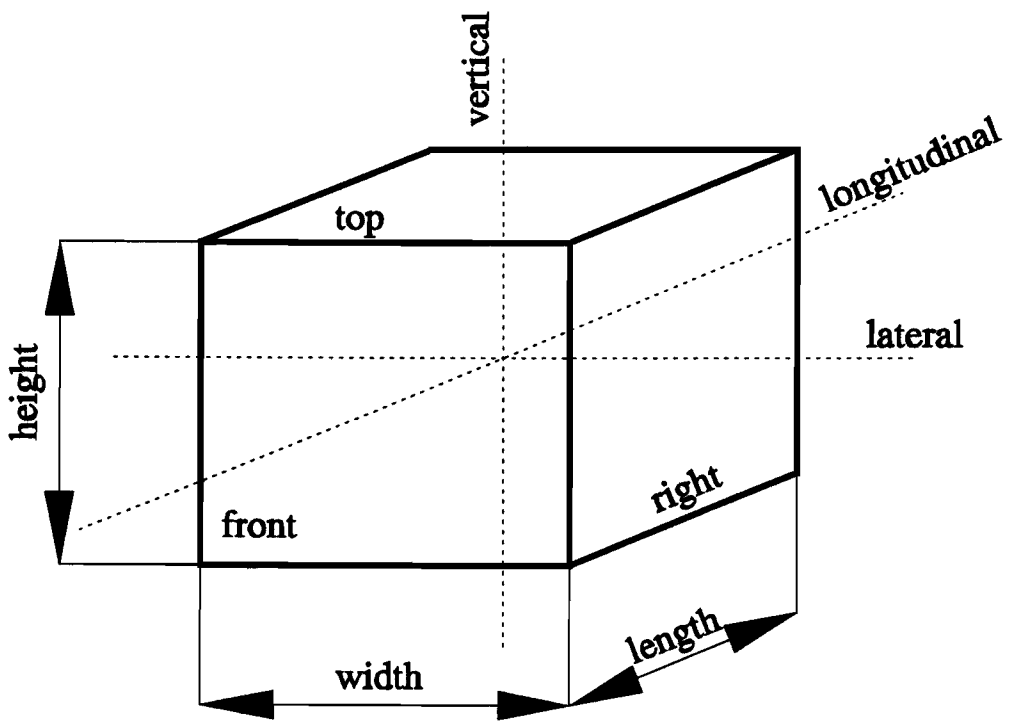

Figure 7

6.3. What about :position ?

With the position of a part, the position of a part in respect to a reference part, is meant.

Each part has a default position, at the center of the local coordinate system of the containing defpart. The part's default position can only be overridden in the **:parts** statement of its' parent, by using a :position keyword. Explicit positioning can be defined either **face-to-face** or **center-to-center** in respect to the reference part.

Face-to-face positioning allows positioning of a part relative to a face of the reference object. Center-to-center positioning allows the positioning of a part relative to the center of the reference object rather than a face.

Some important rules in ICAD for positioning are:

- You can mix and match the different kinds of positioning.
- You can use only one keyword for each axis.
- If you do not specify a position in a dimension the part will be positioned in the center of that dimension of the containing defpart.

Example of positioning:

```
(defpart house (box)
      :attributes
      (:height 10
       :length 20
       :width  10
      )
      :parts
      ((first-floor :type box)
       (roof        :type box
                    :position (:above 0)
       )
      )
)
```

6.4. What keywords are used for face-to-face positioning?

For positioning inside a face defined by the defpart containing the :position use the keywords: **:right :left :top :bottom :front :rear**. For example, the **:top** keyword will put the top face of a part certain distance from the top face of the containing defpart. In this case, 'top' is defined as what the containing defpart would think the top is --not what the part bing positioned does. This can sometimes be difficult to see. Positive expressions position the part closer to the center.

For positioning outside a face defined by the containing defpart, use these keywords: **:on-right :on-left :above :below :in-front :in-rear**. For example, the **:above** keyword will position the bottom

face of a part a certain distance from the top face of the containing defpart. 'Bottom' is defined as what the defpart would think the bottom is. Positive expressions position the part further from the center.

6.5. What keywords are used for center-to-center positioning?

For positioning a part relative to the center of another object rather than a face the following keywords can be used: **:lateral (positive to right), :vertical (positive to top), :longitudinal (positive to rear).**

The expression used in:

```
:position
(position-keyword expression)
```

should return a number. It can be positive or negative.

6.6. What if you do not wish to position in respect to the defpart?

You may have noticed that in the previous example all the parts were positioned relative to its parent. It is also possible to position a part with respect to an object other than its parent. This can be done with the LISP command **:from**. The format of the **:position** command is than changed to:

```
:position
(position-keyword
      (:from (the :part-name) expression
      )
)
```

With this command you are able to position sub-assemblies relative to other object in the assembly which sometimes comes in very handy. You have to be very carefull to use this command though because the positioning in other dimensions isn't affected.

Example:

```
(defpart house (box)
      :attributes
            (:height 10
             :length 20
             :width  10
            )
      :parts
      ((first-floor       :type box)
       (first-floor-door  :type box
                          :height 0.1
                          :width  3
```

```
                    :length 3
                    :position (:left (:from (the :roof) 0.5)
                                     :vertical -5
                              )
          )
          (roof     :type box
                    :position (:above 0)
          )
          )
)
```

EXERCISE 6.1:

Define a wall that has a door and a window. The door is positioned in respect to the defpart 'wall'. The window is positioned in respect to the door.

	height	width	thickness	positioning
Wall	10	20	0.1.	
Door	5	2	0.05	3 left from wall center
Window	2	1	0.05	1 right from door

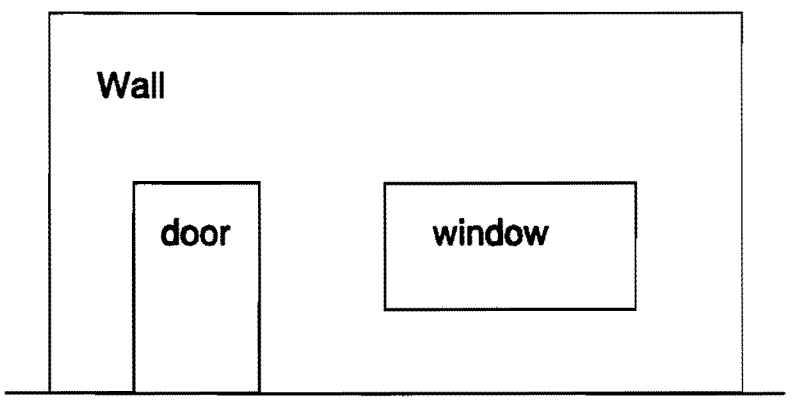

Figure 8

6.7. What about :orientation ?

Objects which are not explicitly oriented default to the orientation of the containing defpart. And in contradiction to positioning a part can only be reoriented in respect to the faces or axis defined by the containing defpart. Two types of rotation are possible: orthogonal and non-orthogonal rotation.

Syntax for an orthogonal rotation:

```
:orientation
      (:rotate rotation-keyword
      )
```

Syntax for a non-orthogonal rotation:

```
:orientation
      (:numeric
      (roll axis angle
      )
)
```

The keywords for a 90 degree orthogonal rotation are: **:right :left :front :rear :top :bottom**. These keywords rotate the part according to the right-hand rule. Point your right thumb in the indicated direction, and your fingers will curl in the direction of rotation. Keywords for 180 degree rotations are: **:lateral :longitudinal :vertical**, and will cause a rotation around the selected axis of the defpart.

To rotate an object at other than right angles, the non-orthogonal rotation is used. The options for axis are: **:lateral :longitudinal :vertical**. The angle is expressed in radians (2π radians equals 360 degrees). There is a function called **degree** that you can use for expressing angles in degrees. You can refer to π by using ***pi***, **2*pi***, and ***pi*/2** are also defined.

Example of rotation:

```
(defpart cube (box)
:attributes
      (:height 10
       :length 20
       :width  10
      )
:parts
((oriented-cube  :type box
                 :height 2
                 :length 4
                 :width  2
                 :position (:rear 0)
                 :orientation (:rotate :top)
 )
)
)
```

EXERCISE 6.2:

Type in this example. In the ICAD Browser you will notice that the oriented-cube is positioned and oriented according to the programcode.
Change the keywords in the :position and :orientation commands and predict the results.

6.8. Can I specify more than one rotation and translation?

As you might have guessed, you can specify more than one rotation and translation to get the object in the desired orientation and on the right place. For this you can use the same LISP commands as were previously explained. You simply adapt these commands to.

Syntax for more than one rotation and translation:

```
:position
    (position-keyword1 expression1
     position-keyword2 (:from (the :part-name) expression2)
     position-keyword3 expression3
    )

:orientation
    (:rotate rotation-keyword1
     :numeric (roll axis angle)
    )
```

For the translation part the order in which the translations are presented do not matter, for the orientation part though, the order in which the rotations are specified determines the end result.

CHAPTER 7. QUANTIFICATION.

7.1. What is Quantification?

Quantification is a means of replicating parts using a single **:parts** statement. In this way there is not the need for specifying two or more similar parts that only differ in position or orientation.

The **:quantify** statement that is used for this reason, can be used as if it were an input in the part specification. All the other inputs on the other hand are given to each of the quantified part.

The default positioning of quantified parts is to evenly fill the parent's bounding box. The parent's bounding box is the box that holds defpart. Al the defpart's parts are positioned and oriented in respect to the axis and sides of this bounding box. The replication itself can be done in one of the three ways mentioned below:

in case of a pair of parts:

```
:quantify (:pair direction-keyword)
```

in case of a series of parts:

```
:quantify (:series direction-keyword quantity)
```

in case of a matrix of parts:

```
:quantify (:matrix direction-keyword-1 quantity-1
                   direction-keyword-2 quantity-2
          )
```

The direction keywords that can be used in the statements above are: **:lateral**, **:longitudinal** and **:vertical**. At the place of quantity, used in the second and third statement, you have to fill in the number of similar parts you want to place in one direction.

Example:

```
(defpart I-beam (box)
    :attributes
    (:width 0.8
     :height 0.6
     :length 5
     :flange-thickness 0.2
    )
    :parts
    ((flange    :type box
                :width (the :flange-thickness)
```

```
                    :quantify (:pair :lateral)
        )
        (web        :type box
                    :height (the :flange-thickness)
                    :width (- (the :width) (twice (the :flange-thickness)))
        )
       )
)
```

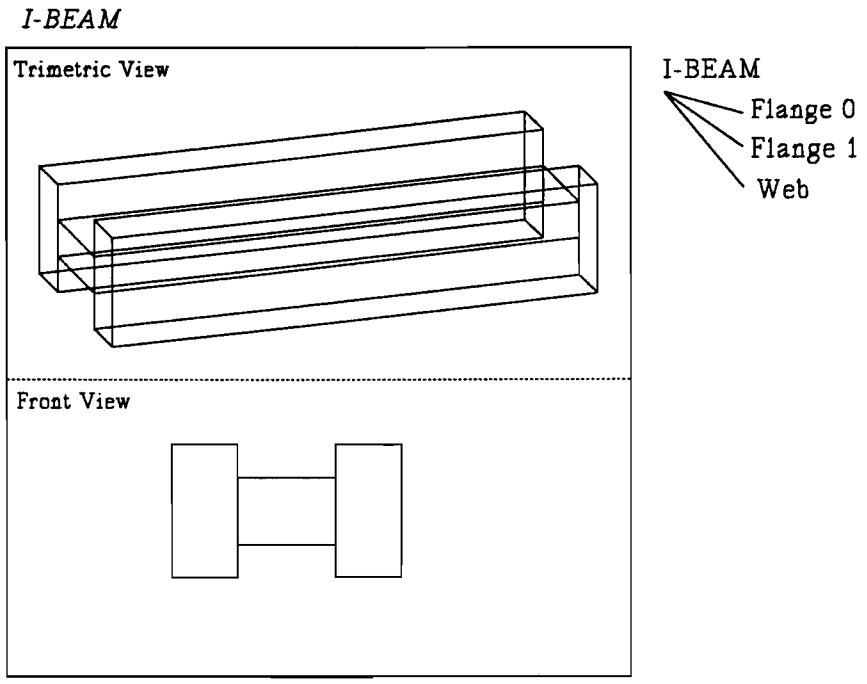

Figure 9

EXERCISE 7.1:

Define a wall which has a series of 3 windows and one door.

7.2. What if I don't want the quantified objects evenly spaced?

As was mentioned in the previous paragraph, the default positioning of quantified objects is evenly spaced along the specified axis, and centered otherwise. If this is not the position that you want the parts to have, you will have to specify an alternate bounding box (rather than the parent's bounding box) to fill evenly with the quantified parts.

The default position of this alternate box is the center of the parent's local coordinates, but it can be placed in any place you want.

The format of the IDL statement that defines the box and places it, is:

```
:quantify-box (dimension expression
                    :position position-specification
                )
```

Example:

```
(defpart wall (box)
    :optional-inputs
    (:window-width 3
     :window-height 4
    )
    :attributes
    (:length 0.6)
    :parts
    ((wall :type box)
     (windows :type box
              :width (the :window-width)
              :height (the :window-height)
              :position (:top 1)
              :quantify (:series :lateral 3)
              :quantify-box (:width (- (the :width) 10)
                                    :position (:left 1)
                                    )
     )
    )
)
```

7.3. What about referencing quantified objects?

In the chapter 'Defpart Elements' some aspects of reference chains are explained. Now we will explain how you can refer to a specific member of a quantified set of objects. Referencing will be used to refer direct or indirect to an attribute of that member. There are certain conditions that a referencing chain has to forfil.

Firstly, the referencing chain has to be augmented with keywords which specify the member. Secondly, the chosen keywords should respect the axes that were defined in the **:quantify** statement, and be chosen for their readability.

One commonly used keyword is **:any**. This allows you to refer to one member of the set, without having to be specific about which particular one. Someone reading your code will then infer that all of the members of the set are the same in this respect.

An example is:

```
(the stud :any :stud-length).
```

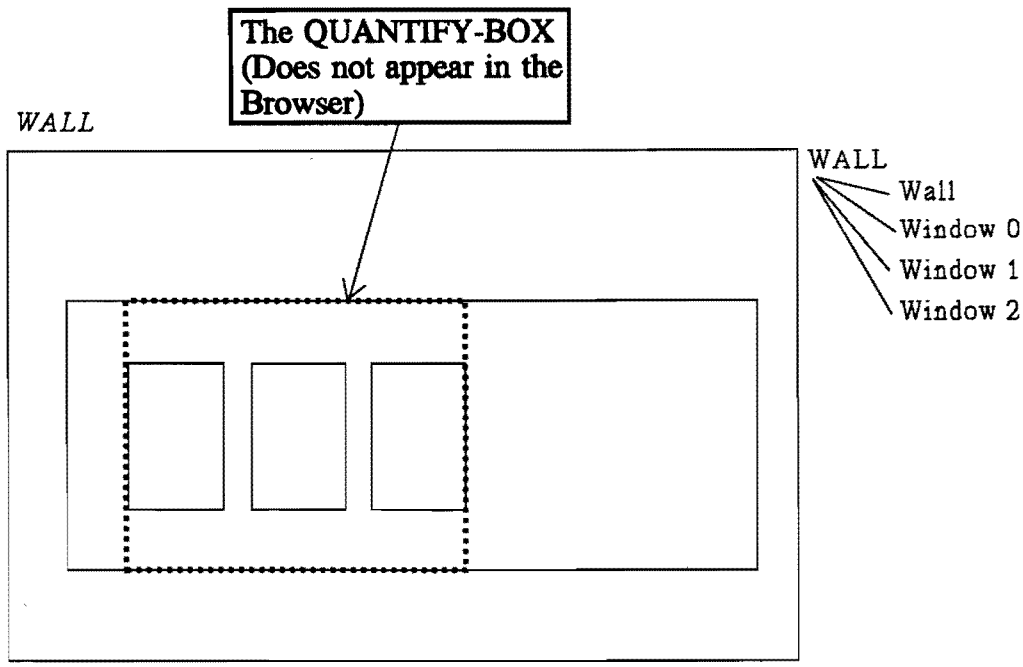

Figure 10

Another set of keywords are the direction keywords. You specify a direction and expression which indicates the relative position of the part. Note that the numbering starts with 0 ! The direction keywords that can be used are: **:top, :bottom, :right, :front, :rear**.

Examples:

(the :stud (:front 2) :height),
which refers to the height of the third from the front member of a series quantified along the longitudinal axes.

(the :stud (:right :rear 3 2) :height),
which refers to the height of the fourth from the right and the third from the rear member of a matrix quantified along the lateral and the longitudinal axes.

Other commonly used keywords specify a relative position: **:first, :last.** :First and :last specify the first and last members. respectively, for a series or pair. And finally you have got the **(:first n)** command, which refers to the nth member of a pair or series.

Example:

```
(defpart wall (box)
    :attributes
        (:length 0.15
         :no-of-studs (round (the :width) 0.4)
        )
    :parts
```

```
            ((stud      :type 2x4
                        :quantify (:series :lateral (the :no-of-studs))
             (small-window-1    :type box
                                :height 0.6
                                :width 0.2
                                :position (:on-right
                                           (:from (the :stud :first)
                                                  0.1))
             )
             (small-window-2    :type box
                                :height 0.6
                                :width 0.2
                                :position (:on-right
                                           (:from (the :stud :first 4)
                                            0.1))
             )
             (wall :type box)
            )
)

(defpart 2x4 (box)
      :attributes
            (:length 0.05
             :width 0.1
            )
)
```

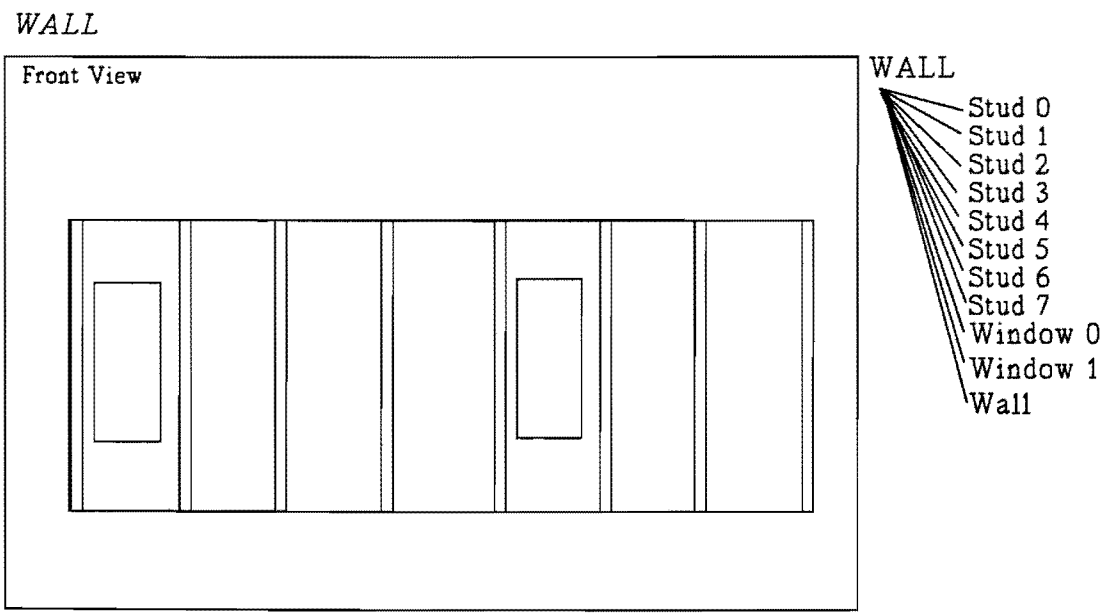

Figure 11

7.4. How to distinguish a certain quantified object?

In order to make quantified objects different from each other, you must be able to distinguish one from another in the **:parts** statement. The following keywords do this.

:Index is an attribute associated with quantified parts. Its value is a number (for a pair or series) or a list of two numbers (for a matrix).

Example:

(the :stud :first :index) results in 0

You must use the Lisp functions **first** and **second** in order to get the individual numbers out of this list.

The messages are also very useful: **(:is-nth-from-end direction-keyword)** is the same as **:index** but lets you specify a direction context in which to compute the index. The direction must correspond to the axis or axes of the **:quantify** statement. Options for the direction are: **:right**, **:left**, **:front**, **:rear**, **:top** and **:bottom**.

:First? and **:last?** return true when referring to the first or last member, respectively, of a quantified set. (Note that these are different from **:first** and **:last** without the **?**) These keywords are used in conditional expressions.

```
(defpart wall (box)
      :attributes
            (:length 0.15
             :height 3
             :width 4.5
            )
      :parts
            ((wall      :type box)
             (window    :type window
                        :position (:top (+ (* (the-child :index) 2) 0.25))
                        :quantify (:series :lateral 3)
                        :quantify-box (:width (- (the :width) 5)
             )
            )
)

(defpart window (box)
      :attributes
            (:width 2
             :height 3
            )
)
```

EXERCISE 7.2:

Define a wall with a series of three windows and a door. Position the door 0.3 meter to the right of the last window in the series of windows. Experiment with different alternatives for referring to the last member of the series of windows to insure they give the same result. What effect does rotating the wall have on the position specification for the door? On the keyword used for referencing?

CHAPTER 8. TRICKS SPECIAL.

8.1. What can you use this chapter for?

The meaning of this chapter is to give you some guidelines to work easier with EMACS and ICAD Browser. But also some main errors will be explained.

8.2. What compile-time and edit-time errors can be made?

Often, most of these compile errors can be avoided, or more fully explained, by using IDL Help (**CTRL-META-?**).

8.2.1. Unbalanced parentheses.

'Unbalanced' really means that there are not enough closing parentheses. Easily detectable because the defpart will not compile. Sometimes you will get a very long error message that scrolls quickly through the bottom window of the editor. This is an unbalanced parentheses error. The editor commands, CTRL-META-b (move backward to the beginning of the expression) are helpful for finding unbalanced parentheses.

Another way of finding these parentheses is to remove some 'closing' parentheses in a defpart and put new ones in. For each new one EMACS Editor will show the matching 'opening' parentheses. Enter now as many 'closing'-parentheses until you match the 'opening' parentheses, which is in front of 'defpart'. However, this does not guarantee that all the parentheses are placed correctly.

8.2.2. Misbalanced parentheses.

'Misbalanced' means that you have the right number of parentheses, but they are not in the right place. Slightly harder to detect. Use the editor indenting commands, CTRL-META-\ or CTRL-META-Q (indent expression) and TAB to see if the indentation is correct (or use LINE). This will lead you to the place with the misbalanced parentheses. Using non-standandard indentation (by manually spacing over) means you can not use this important tool.

8.2.3. Function undefined.

You probably misspelled a function name. Possibilities are:
- You might have used an extra set of parentheses, so that you have some other (valid) symbol in the first position of an expression (an invalid place).
- You left out a ':' in front of an attribute name, or a quote in front of a list of data.

8.2.4. Redefining old version.

You have chosen a name for a defpart that is already being used as one, and was not defined in this file. DO NOT IGNORE THIS WARNING. Answer 'NO', and choose a new name (unless you really want to redefine the old version).

8.3. What run-time errors can be made?

8.3.1. No parent could handle the XXX message.

You referred to an attribute that was not available because:
- you misspelled the name;
- it is not a descendant-attribute;
- it is not really there.

8.3.2. The function XXX is undefined.

Often means that you have misspelled the name of a part, or that you just forgot to compile it.

8.3.3. The control (or binding) stack overflowed.

You have referred to an attribute while trying to define it. This is called a circular definition. This is a nasty one, because this condition does not produce an error message. You know you are in an 'infinite loop' when the browser command takes an uncommonly long time, and in fact will never finish. Return to the EMACS Window, then press '**CTRL-C**' twice to stop the ICAD Browser. Correct the circular reference, and recompile.

8.3.4. The variable XXX is unbound.

Probably missing a quote in front of a symbol.

8.3.5. The argument given to the SYS: -INTERNAL instruction, XXX, was not a number.

You supplied XXX where a number was expected. Usually means a problem with a referencing chain.

8.4. How can you generate descriptions without introducing failures?

It is difficult to generate defpart descriptions without making failures. The answer to this question is, you need a lot of experience, and even then it is not quaranteed that no failures will be present. However there are no rules, which prevent you from making failures, there are some guidelines to minimize the number of bugs.

8.5. How to minimize the number of bugs?

First of all, avoid complex defparts. Use small defparts. Create hierarchy of defparts which are tested one by one. Test out each change that you make to your defparts. That way you catch the bug as soon as it is introduced in the defpart.

Second, evaluate first before you update text.

Third, make generic parts. Generic parts, by definition, can be tested independently of the rest of the tree.

Fourth, try to make defpart descriptions with a uniform layout. Try to reuse verified layouts. And this applies especially for the way you put the parentheses.

Fifth, make backups of your IDL description.

Six, use names for the objects from which you can easily read what the meaning of the object is. So do not use 'mb1' when you mean 'motorbike-1'. This way of describing will be not only a help to you, but also to another who tries to read your description for the first time.

8.6. How to debug?

Since it is not guaranteed failures are not present, you need some strategy to solve the failures. An important step is to ISOLATE. The trick to debugging quickly is to isolate the bug. Perhaps next items will help:

- Work only on one bug at a time.
- Selectively expand and draw branches of the tree to isolate erroneous parts.
- Use the inspector in the ICAD Browser to evaluate that attribute to insure that the error is isolated.
- Evaluate dependent attributes to see if they give the expected results.
- Comment out the suspect attribute or replace with a new attribute definition to see if any dependency problems result.
- Aggravate your bugs, do not coddle them. That helps to isolate them.

EXERCISE 8.1:

Try to find the errors in the file 'error1.lisp'. This file contains four errors.

CHAPTER 9. EXERCISE.

9.1. What is the meaning of this final exercise chapter?

This is the last part of the introduction course to ICAD. You are supposed to know how to work and to describe defparts with it. From now on you will have to use the manuals to look up the answers for the problems that will occur but were not dealt with in this course.

The goal of this exercise is to define a clock as shown in the figure. To build this clock the exercise is divided in five logical parts. In this way the problem will be tackled in the most easy manner.

EXERCISE 9.1.

A block is defined which has a clock on one or more sides. The specification of a clock is: it has a clock plate, a minute and an hour hand and ticks. There is a set ratio between the length's of the hands. One can choose whether the clock has 4, 12 or 60 ticks and what time the clock has to show.

1. Construct the tree of this block with the clocks.
2. Define the input and output variables.
3. Describe in a global way how you would construct the IDL description (do not write the IDL description in detail!).

EXERCISE 9.2.

The input can be realized in different ways. One possibility is to use the user interface type as described below. This will be shown as a input window in the Browser as soon as you create an instance of the defpart.

1. Type the IDL description as shown below and compile it. Explain it's working.
2. Add the IDL-commands needed:
   - so that the information from the interface is used;
   - to draw a block with the selected body in the center of it, this body must have dimensions smaller than the block.
3. Save the description as 'EXER2.LISP'.

```
(defpart block (box base-spec-sheet)
    :attributes
        (:length (the :width)
         :width  (the :length-spec)
         :height (the :width)
        )

    :spec-controls
```

```
        (:graphics-after (:action :draw))

    :spec-attributes
        (:length-spec
            (:domain   :number
             :default 10
             :prompt "length"
            )
        :body-spec
            (:domain (:item-list
                              '(:cylinder :box))
             :prompt "select body"
            )
    :parts
        (...)
)
```

EXERCISE 9.3.

1. Define a cylinder which has a block on it. This block, block1, is placed on the edge of the top of the cylinder. The length of the cylinder is 0.5, the radius is 5. The dimensions of block1: height 0.5, width 0.5, length 0.5. This total object has to be placed on the outside surfaces of another block, block2. The dimensions of block2: length, width and height 10. Use the IDL description which is given below.
2. Save the description as 'EXER3.LISP'.

Hints:

Use for orientation of the cylinder the following command **:orientation(:numeric (alignment etc.**

```
(defpart block2 (box base-spec-sheet)
    :attributes
        (:length (the :width)
         :width  (the :length-spec-block2)
         :height (the :width)
        )

    :spec-controls
        (:graphics-after (:action :draw))

    :spec-attributes
        (:length-spec-block2
            (:domain   :number
             :default 10
             :prompt "length"
            )
```

```
        :position-spec
            (:domain (:selection-list
                              '(:front :rear :top :bottom :right :left))
             :default '(:front)
             :prompt "select position"
            )
    :parts
        ()
)
```

EXERCISE 9.4.

1. Define a block which has one clock plate. The plate itself has a minute and an hour hand that are positioned in the center. The clock-plate has 4, 12 or 60 ticks. The time can be chosen. Use the IDL description given below. You have to be able to position and orientate the plate on any surface of the block so that the hand is on the outside.

2. Save the description as 'EXER4.LISP'.

Hints:

Use for the positioning of the ticks and hands the **:position-about** command.

```
(defpart clock (box base-spec-sheet)
    :attributes
        (:length (the :width)
         :width  (the :length-spec)
         :height (the :width)
        )

    :spec-controls
        (:graphics-after (:action :draw))

    :spec-attributes
        (:hour-spec
            (:domain :number :display-width 2
             :invalid?
                (cond ((> (the :hour-spec) 24)
                       "Hour cannot be larger than 24")
                      ((< (the :hour-spec) 0)
                       "Hour cannot be smaller than 0"))
             :prompt "hour"
            )
         :minute-spec
            (:domain :number :display-width 2
```

```
                :invalid?
                    (cond ((> (the :minute-spec) 59)
                           "Minute cannot be larger than 59")
                          ((< (the :minute-spec) 0)
                           "Minute cannot be smaller than 0"))
                :prompt "minute"
               )
            :length-spec
               (:domain   :number
                :default 10
                :prompt "length"
               )
            :ratio-spec
               (:domain   :number
                :default 0.8
                :invalid?
                    (cond ((> (the :ratio-spec) 1)
                           "Ratio cannot be larger than 1")
                          ((< (the :ratio-spec) 0)
                           "Ratio cannot be smaller than 0"))
                :prompt "ratio"
               )
            :ticks-spec
               (:domain (:item-list '(4 12 60))
                :default 4
                :prompt "select ticks"
               )
            :position-spec
               (:domain (:selection-list
                                '(:front :rear :top :bottom :right :left))
                :prompt "select position"
               )
    :parts (...)
)
```

EXERCISE 9.5.

1. Do the same as in the preceding exercise, but now clock-plates can be placed on one or more clock-block surfaces at the same time.
2. Save this description with the name 'EXER5.LISP'.

Hints:

Use the **:quantify** and the **:index** command.

APPENDIX A. ANSWERS TO EXERCISES.

CHAPTER 2. GNU EMACS EDITOR AND SYSTEM COMMANDS.

EXERCISE 2.1: -

EXERCISE 2.2: -

EXERCISE 2.3: -

EXERCISE 2.4: -

CHAPTER 3. ICAD DESIGN LANGUAGE AND COMMON LISP.

EXERCISE 3.1:

a. number
b. symbol
c. number, keyword, number
d. list, with one element
e. list, which also is a legal expression

EXERCISE 3.2:

a. illegal
b. illegal
c. illegal
d. illegal
e. illegal
f. legal
g. legal
h. illegal
i. illegal
j. legal
k. illegal
l legal
m. illegal
n. illegal

EXERCISE 3.3:

a. FALSE
b. TRUE
c. TRUE
d. FALSE
e. FALSE

EXERCISE 3.4:
When the owner of a chair is the president or the vice-president, then the chair will be of leather. Otherwise the chair will be of vinyl.

EXERCISE 4.1:
In Browser a input window will be shown. You have to enter the :ceiling-heigth. It is not necessary to enter the :wall-thickness. A box will be drawn with the desired dimensions. One other box will be drawn in this first (bounding) box. Another will be drawn on top of it.

EXERCISE 4.2:

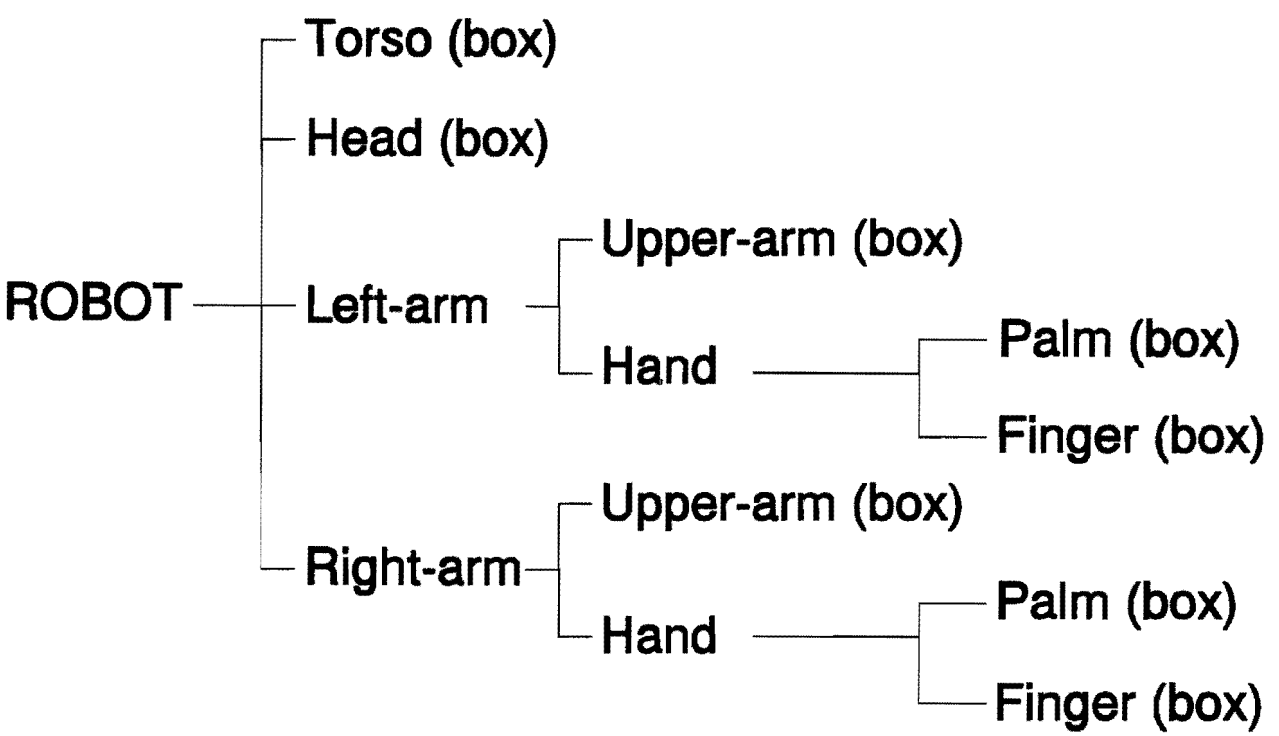

Figure 12

EXERCISE 4.3: -

CHAPTER 5. THE ICAD BROWSER.

EXERCISE 5.1: -

EXERCISE 5.2: -

EXERCISE 5.3: House has two children and two leaves.

EXERCISE 5.4: -

CHAPTER 6. POSITIONING AND ORIENTATION.

EXERCISE 6.1: -

EXERCISE 6.2: -

CHAPTER 7. QUANTIFICATION.

EXERCISE 7.1:

for example:

```
(defpart wall (box)
    :optional-inputs
    (:window-width 0.9
     :window-height 1.2
    )
    :attributes
    (:length 0.15)
    :parts
    ((wall      :type box)
     (windows   :type box
                :width (the :window-width)
                :height (the :window-height)
                :position (:top 0.3)
                :quantify (:series :lateral 3)
     )
     (door      :type box
                :height (* 2 (the :window-height))
                :position (:bottom 0)
    )
)
```

EXERCISE 7.2:

for example:

```
(defpart wall (box)
    :optional-inputs
```

```
    (:window-width 0.9
     :window-height 1.2
    )
    :attributes
    (:length 0.15)
    :parts
    ((wall      :type box)
     (windows   :type box
                :width (the :window-width)
                :height (the :window-height)
                :position (:top 0.3)
                :quantify (:series :lateral 3)
                :quantify-box (:width (- (the :width) 3)
                                 :position (:left 1))
     )
     (door      :type box
                :height (* 2 (the :window-height))
                :position (:on-right (:from (the :window :first) 0.1))
     )
)
```

CHAPTER 8. TRICKS SPECIAL.

EXERCISE 8.1.:
Lines:
#3 (10) → 10
#14 the: roof → the :roof
#15 unbalanced parantheses after '0.5)'
#17 misbalanced parantheses → have to be placed between #20 and #21

CHAPTER 9. EXERCISE.

EXERCISE 9.1.: -

EXERCISE 9.2.: -

EXERCISE 9.3.: -

EXERCISE 9.4.: -

EXERCISE 9.5.: see listing klok2.lisp.