CIS-TR-84-02
An Introduction to ORBS

Stephen Fickas

Department of Computer and Information Science
University of Oregon

CIS-TR-84-02

An Introduction to ORBS

Stephen Fickas

Department of Computer and Information Science
University of Oregon

Abstract

This paper introduces the ORBS expert-system writing environment. In defining ORBS, we have attempted to use the key ideas from past expert system language efforts in building an environment that supports incremental, interactive construction of expert systems. Among others, ORBS inherits ideas from Hearsay III, YAPS, RLL and LOOPS. The paper follows an extended example of the ORBS representation of a VLSI silicon compilation rule.

An Introduction to ORBS

by

Stephen Fickas
Computer Science Department
University of Oregon

July 1984

1. Introduction

Groups at the University of Oregon and Oregon State University are working on the construction of an expert-systems environment that attempts to incorporate the important ideas of past expert-system language efforts in an environment that is general, supportive and simple. This paper describes the environment, called ORBS (Oregon Rule Based System). We introduce ORBS by discussing its major pieces: the data base (section 2), rules (section 3), control and scheduling (section 4), and the development environment (section 5). We will carry a single example through the paper, one taken from a silicon compiler in the VLSI domain.

ORBS traces its lineage to four ancestors: Hearsay III [1], YAPS [2], RLL [3] and LOOPS [4]. We see the major contributions of each as follows:

- *Hearsay III*: separation of performance and competence knowledge, blackboard model.
- *RLL*: language extension.
- *YAPS*: simple syntax, manageable system.
- *LOOPS*: meld of procedural, rule, object oriented, active value paradigms into a single system. Powerful interactive, incremental development environment using Interlisp-D and graphics.

We only slightly overgeneralize by saying that each system above lacks the key features of the others. ORBS attempts to bring as many of these features together as possible.

2. The ORBS Data Base

In outline form, an ORBS system contains a set of relational facts in its data base, a set of forward-chaining rules that trigger on those facts, and a set of scheduling rules that determine which triggered rule to execute. We will look at each of these pieces in more detail in coming sections. In this section, we concentrate on the data base.

ORBS uses a data base that is identical to that of YAPS [2]. It is similar to the Hearsay III blackboard in that it is relation-based. Its key features are as follows:

(1) Objects within the ORBS data base are either Lisp objects or Flavor instances (an ORBS flavor is similar to a Zetalisp flavor [5], which is more or less similar to a Smalltalk object). Object attributes and components can be defined either through relations, or directly through instance variables and methods.

(2) ORBS relations are free-form and untyped s-expressions.

(3) Each instance of a relation on the blackboard has a unique identification number associated with it ala OPS5 [6]. Identical relations can appear in ORBS; they are differentiated by their ID number. This capability is crucial to maintaining a process history, which can be used by a debugger to

perform process backtracking. Further, ID numbers are monotonically increasing, hence the recency of relations can be determined by numerical ordering.

In summary, the ORBS data base attempts to walk the line between simplicity and representational power. It allows simple, free-form relations when that is all that is called for. It provides a more powerful modeling representation (i.e., object-oriented programming) when that is needed. In essence, ORBS provides a direct correlation between problem complexity and system complexity: simple problems can use simple representations, complex problems can use complex representations.[1]

2.1. A VLSI example

The example we will use throughout the rest of the paper is taken from a silicon compiler in the domain of VLSI. Because this is a complex domain, we will use the more powerful modeling features of ORBS, namely object oriented, flavor representation.

We will use Hearsay III as a side-by-side comparator in the VLSI example. We do this for two reasons: 1) it is instructive to show the differences between a typed, blackboard data base, and an untyped, free form data base extended with flavors, and 2) much of ORBS scheduling is motivated by the Hearsay III model of control. We hope to show the similarities between the two languages, and the places where we have extended Hearsay III ideas to achieve simplicity and extensibility. We will first present the ORBS representation, and then that of Hearsay III.

The specific domain object we are interested in is a ring-oscillator. This is defined in terms of a more general object called a circuit-component. Using flavor terminology, we will *mix-in* a circuit-component flavor into a ring-oscillator flavor to get the desired attributes. First we define the circuit-component flavor:

```
(defflavor circuit-component
        (x-dim y-dim)        ;instance variables, i.e., attributes
        nil)                 ;a circuit-component inherits from nothing in this example


(defmethod (circuit-component area) ()      ; a method to handle "area" messages
           (quantify-area x-dim y-dim))


(defun quantify-area (x y) ... )            ;computes area from x,y dimensions
```

Now we define a ring-oscillator:

[1]Note that this is not the case in many systems where system setup/initialization for any problem requires a large ante, and a single representation is provided irrespective of problem complexity.

```
(defflavor ring-oscillator
        (timing)                ; an instance variable
        (circuit-component))    ;a ring oscillator inherits from circuit component


(defmethod (ring-oscillator speed) ()       ; a method to handle "speed" messages
             (quantify-speed timing))


(defun quantify-speed (time) ... )          ;computes speed from timing
```

Above, the ring-oscillator flavor inherits instance variables (x-dim, y-dim) and methods (area) from circuit-component. It also adds its own instance variable (timing) and method (speed). Once we have defined the ring-oscillator flavor, we must instantiate it, and make it available to our rules, i.e., add it to the data base. The following will accomplish this:

```
(fact ring-osc (make-instance 'ring-oscillator ... ))
```

"fact" will place the relation ring-osc in the data base with a single argument of a ring-oscillator flavor instance.

Now let's look at the Hearsay III representation. We will rely on Hearsay III's relational database, and a predefined relation to model the ring oscillator. Hearsay III provides no object oriented representation paradigm, so we will be unable to directly represent abstract objects such as circuit-component, and the associated flavor inheritance. The first step in setting up the Hearsay III representation is defining the unit and relation typing structure. Hearsay III's machinery for accomplishing this is both conceptually and syntactically complex. Instead, we will paraphrase the process: we must define a ring-oscillator type, and a set of typed relations between a ring-oscillator and its attributes. Hearsay III provides a built-in relation called ROLE-OF to define attributes or components of a structure, so we will use this.

Once the type structure has been defined, we are ready to place a ring-oscillator on the blackboard. We first create a blackboard unit of type ring-osc. We then associate the three attributes x-dim, y-dim and timing by use of Hearsay's ROLE-OF relation ("@" places a relation on the blackboard):

```
(SETQ ringo (MKUNIT 'ring-osc))
(@ (ROLE-OF 'timing ringo 8))
(@ (ROLE-OF 'x-dimension ringo 2))
(@ (ROLE-OF 'y-dimension ringo 3))
```

Unfortunately, the derived attributes area and speed cannot be explicitly attached to the ring-oscillator unit using the ROLE-OF relation. Instead we implicitly state them as Lisp predicates.

Before leaving this section, we wish to add one final note. In building Hearsay III systems, we have found that much of system development and debugging time can be traced to setting up and maintaining complex type hierarchies. In contrast, ORBS uses a much simpler representation: untyped lisp s-expressions (*with* flavor instances as possible elements). Strong arguments have been made for both approaches. Our experience in building systems with both languages is that working prototypes can be brought up and debugged more quickly, and understood more readily using the ORBS representation.

3. ORBS Rules

ORBS rules are forward chaining. A rule contains one or more left-hand-side (LHS) patterns, zero or more filters, one or more right-hand-side (RHS) actions, several system-defined fields, and zero or more user-defined fields. Before describing the rule syntax, we will look at the match/schedule/execute cycle.

For each rule R,

(1) The LHS of R is matched against the data base. Separate *activations* are created for each different match.

(2) Each activation of R is passed through R's (optional) filtering predicates. A predicate returning nil eliminates the activation from further consideration.

(3) Each filtered activation of R is placed in a conflict resolution set. The conflict resolution set holds the accumulated activations of all rules.

(4) The conflict resolution set is passed through a set of scheduling rules (previously defined by the user). The outcome is a conflict resolution set containing a single activation, which is chosen for execution.

To introduce syntax, we will use an example taken from an ORBS system to do silicon compilation. The piece of knowledge which we wish to represent is the following:

> If you are building a ring oscillator that must be both small and fast, then use inverter-type-3 as the basic building block

We will first look at the ORBS representation of this knowledge, and then for contrast, the Hearsay III equivalent. Assume that we are using the flavor representation of circuit-component and ring-oscillator introduced in the last section. The ORBS rule is as follows:

```
(defp small-fast
        (goal (choose-inverter))
        (ring-osc -ro)
 test   (eq (send -ro 'speed) 'fast)
        (eq (send -ro 'area) 'small)
 -->
        (remove 1)
        (fact ro-cell inverter-3)
 :::
        status: active
        author: simoudis)
```

Figure 1

The LHS is made up of two patterns that will match data base facts (goal and ring-osc); pattern variables are prepended with a hyphen. The (optional) keyword "test" marks the beginning of the filtering predicates. The "-->" marks the beginning of the RHS actions; "remove" removes the fact matching the ith LHS pattern from the data base, "fact" adds a relation to the data base. The ":::" marks the beginning of further rule attributes; "status" is an ORBS-defined field, and "author" a user-defined field. Note that the pattern variable -ro will be bound to the instance of the flavor object ring-oscillator created in the last section.

The goal relation is used for control, providing a means of stepping through a set of tasks. For the rule to match, we must be in the "choose-inverter" task, i.e., someone else must have inserted the goal relation into the data base.

It is instructive to compare the ORBS representation with the Hearsay III representation. The corresponding Hearsay III knowledge source is as follows:

```
(DECLARE-KS small-fast (ro x y z)
    (APAND
        (ring-osc ro)
        (ROLE-OF timing ro z)
        (ROLE-OF x-dimension ro x)
        (ROLE-OF y-dimension ro y)
        (EQ (quantify-speed z) 'fast)
        (EQ (quantify-area x y) 'small))
    'choose-inverter
    (@ (ro-cell inverter-3)))
```

The pattern variables of this knowledge source are ro, x, y, z; no hyphen is needed to distinguish them since they are formally declared. The "APAND" marks the beginning of the trigger, which consists of 4 patterns matching the relations ring-osc and ROLE-OF on the blackboard. The trigger also contains two filtering predicates using EQ. "choose-inverter" is a scheduling level which we will discuss in section 4. The body of the knowledge source consists of a function ("@") that places an instance of the relation ro-cell onto the blackboard. The match/schedule/execute cycle of Hearsay III is similar to ORBS in that triggering is separated from execution by a scheduling step. However, there are major differences which we will postpone discussing until section 4.1.

3.1. Rule Matching

The ORBS rule small-fast includes two patterns: (ring-osc -ro) and (goal (choose-inverter)). Assume that data base relations exist that match these patterns. Before this match can be included in the conflict set, it must pass the filtering tests. All expressions following the test keyword are evaluated in turn. If any returns a nil value, the match is removed from consideration. In this case, the filter asks the ring-oscillator (i.e., the flavor instance representing the ring-oscillator) for its speed and area by sending the appropriate messages to -ro.

```
test (eq (send -ro 'speed) 'fast)
     (eq (send -ro 'area) 'small)
```

Note that using the Flavor approach means that retrieving a directly stored value like "x-dim", and a derived value like "area" uses uniform machinery: message passing. Note also that unlike the Hearsay KS, the ORBS rule has no need to reference or know about the instance variables "x-dim", "y-dim" and "timing".

3.2. RHS Actions

The action part of an ORBS rule contains one or more s-expressions, each of which is evaluated in turn. Certain pre-defined functions, such as "remove" and "fact", are built-in. Just as Hearsay III will allow any valid Interlisp expression to appear in an action, ORBS will allow any valid Franzlisp or flavor message passing expression to appear in an action. The only difference is that Hearsay expects a single expression, whereas ORBS supplies an implicit *progn* to a sequence of expressions.

The RHS of the ORBS rule small-fast contains two actions. The first is a built-in function that removes the relation from the data base that matches the ith LHS pattern. In this case i = 1, hence a goal relation is removed. The second action, also a built-in function, adds relations to the data base. In this case, we note the fact that the ring-oscillator cell has been chosen by adding the relation ro-cell with argument inverter-3.

The RHS of the Hearsay KS uses the built-in function "@" to add a relation to the blackboard. We will discuss later why no goal relation, and hence no remove action, is needed in Hearsay III.

3.3. The Conflict Set

We will call the collection of relations that match a LHS pattern/filter in ORBS, or a LHS trigger in Hearsay III, an activation. Multiple activations are created when more than one rule matches, or the same rule matches different ways. Once the set of activations, called the conflict set, has been gathered, the system must decide which to choose for application. Currently, ORBS allows only one activation to be chosen[2] , whereas Hearsay III allows multiple activations to be applied. Both Hearsay III and ORBS are based on the view that competence knowledge -- domain knowledge like our small-fast rule -- should be separated from performance knowledge -- knowledge that helps choose among competing activations in the conflict set. Both allow the user to tailor performance knowledge to fit the application (in contrast to languages like YAPS and PROLOG, which have a single, built-in strategy). As discussed in more detail in section 4, ORBS attempts to simplify the definition of performance knowledge by providing a uniform interface, and a catalog of scheduling knowledge found useful in past systems.

3.4. Extensibility

Before leaving our discussion of ORBS rules, we note their extensibility. An ORBS rule may contain zero or more user-defined fields. Using an ORBS rule-extension declaration, the user may add one or more attributes to one or more rules. In the case of the rule small-fast, the extension might have been defined as follows:

```
(rule-extension *all-rules* (author))
```

ORBS gives the rest of the system machinery for accessing these fields. Hence, we could define a scheduling rule that choose one rule over another depending on the confidence in the rule author. In general, we have found this ability to extend a rule a powerful part of the language. When properly integrated with the rest of the system, it allows a rule to be viewed as just another object to be analyzed and modified.

4. Scheduling

Hearsay was a pioneering system in separating competency knowledge from performance knowledge. In Hearsay, KS firing is separated from KS triggering by a scheduling step. A triggered KS, called an activation, is placed on a *scheduling blackboard* (SBB). The user may define *scheduling knowledge sources* (SKS) that trigger on this placement. These in turn cause activations to be created and placed on the scheduling blackboard (the process stops here: an SKS is not allowed to trigger on the placement of an activation of another SKS). SKS are generally used to organize and order KS activations on the SBB. Once ordered by SKS, the activations of KS are invoked by a domain scheduler (a piece of user-defined Interlisp code). It is this scheduler that normally determines the structure of the SBB, e.g., a priority queue, an ordered agenda.

Moving back to our example from the last section, the Hearsay KS small-fast included a value "choose-inverter" in its scheduling-level field. This associates "choose-inverter" with any activations of this KS. In the Hearsay VLSI system, we structure the SBB around priority levels. "choose-inverter" is one such

[2]We are experimenting with multiple activation applications on any cycle, but have yet to work out the problems of one RHS action invalidating the application of a following activation.

level. It has levels above it and below it, i.e., "start-up", "order-cells". When a KS activation is created, it is placed on the level determined by its scheduling-level field. Hearsay III's (user-defined) VLSI scheduler simply moves down levels looking for activations. If one is found, it is fired, and its activation is removed from the SBB. In this way, all "start-up" activations are executed before all "choose-inverter" activations, which are executed before all "order-cells" activations, etc.

In ORBS, we have attempted to simplify Hearsay III's scheduling process. Scheduling in ORBS involves 1) defining a set of scheduling rules in Franzlisp, and 2) declaring how those rules are to be combined to form a scheduler. As with Hearsay, the triggering of an ORBS rule causes an activation to be created. However, the activation is placed in a conflict set as opposed to on a SBB. Once the conflict set has been built (i.e., all activations have been collected), it is passed through the scheduler. The first rule in the scheduler takes the initial conflict set, and returns a new conflict set. The new conflict set is input to the second rule, which outputs a new conflict set. Generally, each rule either removes or weights one or more activations. This process continues until either no activations remain in the conflict set, or the last rule is called. In the former case, the system halts. In the latter case, the scheduler returns the one remaining activation in the conflict set; if more than one exists, an error message is printed and the break package is called.

An example might be useful here. Suppose that we wanted to define a simple scheduling strategy in our VLSI system that chooses the activation that includes the most recent goal[3] in its LHS pattern. If more than one activation contains the most recent goal relation, the tie will be broken arbitrarily.

We need two scheduling rules: one that will find the rule activation(s) with the most recent goal; one that will arbitrarily select among ties. ORBS provides these rules in a predefined set of scheduling rules. The first is called KW, and the second AD1. Note that the user is not forced to choose from ORBS predefined set: he or she is free to write his or her own rules, or modify existing ones.

We now must inform ORBS that 1) the rules KW and AD1 are to be used in scheduling, and 2) they are to be applied in a certain order. ORBS accepts an extended form of Forgy's scheduling expressions [7] to accomplish this:

```
[(KW goal)] > AD1
```

This defines a scheduling strategy that applies the rule KW to the conflict set, using "goal" as the key-word, and then passes (as denoted by the ">") the resulting non-empty conflict set to AD1. The square brackets say that if the KW rule produces an empty conflict set (i.e., no activations contain a goal relation in their LHS patterns) then halt the system. AD1 will arbitrarily delete all but one of the remaining activations.

The above scheduling strategy assumes that goals will be added to the data base in the appropriate order. This means that we are mixing competence and performance knowledge. Is there a way to remove the tasking information from the data base (and hence rule patterns)? Since the Hearsay III knowledge source seems to have accomplished this, we might take a closer look at how this was done.

The Hearsay III VLSI system used "tasking levels" to represent the order that rules should be run. It did this by associating with each rule a specific task level. As a start, we might do the following in ORBS:

```
(rule-extension *all-rules* sched-level)
```

When we define a rule, we can fill its sched-level field appropriately. We're now half way home. We still have to worry about defining a scheduler that can use this information to order tasking. In Hearsay, this was accomplished by writing code that would process the SBB levels in the right order. We can perform

[3] In ORBS, the most recent goal is the goal relation instance that was most recently added to the data base.

the same thing in ORBS be defining the appropriate scheduling rule:

```
(setq *priority-levels* '(start-up choose-inverter choose-cell ... )) ;a global

(defschedrule PL (cs)
      (find-highest cs *priority-levels*))
```

PL takes as input a conflict set cs, and returns the set of activations (another conflict set) with the highest priority. The function find-highest accomplishes this by looking at each activation in the conflict set, checking the value of its sched-level field against the user-defined priority levels.

Using PL, we can redefine our scheduling expression as follows:

```
[PL] > AD1
```

This allows us to remove the goal relations from the data base. Our new small-fast rule is now as follows:

```
(defp small-fast
      (ring-osc -ro)
 test  (eq (send -ro 'speed) 'fast)
      (eq (send -ro 'area) 'small)
 -->
      (fact ro-cell inverter-3)
 :::
      author: simoudis
      sched-level: choose-inverter)
```

Figure 2

4.1. State Triggering versus Modification Triggering

Although we have shown how ORBS can model some portions of Hearsay III's scheduling machinery, there remains an important difference: ORBS and Hearsay handle triggering semantics differently. In Hearsay, a KS is triggered only when one of the patterns in its LHS matches against a *newly added* fact. In ORBS, a rule triggers when all of the patterns in its LHS match facts in the data base. In summary, a Hearsay KS will trigger only once for any particular set of facts of the blackboard. An ORBS rule will trigger on every cycle that its LHS matches a set of facts in the data base.

The consequence of this is that Hearsay must "remember" what KS activations were created on each cycle. It cannot choose to forget an activation since that activation may be applicable on a subsequent cycle. Since an activation will not be regenerated, it must be kept on the SBB. All activations are placed on the SBB, and will not go away until they are chosen for invocation, or some other process explicitly deletes them. It is up to the user-defined scheduler, on every cycle, to weed through current and past activations, and decide which to invoke.

In ORBS, no activation history is necessary (the one exception is discussed shortly). This is because a rule triggering does not prevent it from triggering again on the same data, i.e., the same activation can be generated multiple times. As an example, suppose that rules R1 and R2 trigger creating activations A1 and A2. Suppose A1 is chosen on the current cycle for execution. On the next cycle it is quite possible for A1 and A2 to be generated again. The only reasons they would not both be generated was if the execution of A1 on the previous cycle caused the data base to be changed so that R1 or R2 no longer triggered, i.e., their LHS patterns no longer matched facts in the data base on the current cycle. If both A1 and A2 are regenerated, the scheduler is free to choose A1 again, A2 or any other activation in the conflict set.

Moving back to our example, there is nothing to stop our new small-fast rule in figure 2 from repeatedly firing. In the old rule in figure 1, a RHS action removed the goal relation so that the rule no longer fired, i.e., the rule effectively shut itself off. Our new rule has done away with the need for the goal pattern, so it appears we are stuck. Assuming we do not want the small-fast rule to be executed more than once, we have two answers. First, we can write a scheduling rule that will eliminate activations executed on previous cycles. This requires a history of executed activations. ORBS keeps such a list in *executed-activations*. Thus the user can define a scheduling rule (actually, ORBS provides one predefined) that searches this list, and eliminates any activations in the current conflict set which are members.

Our second solution attempts to improve on the first. In particular, using scheduling to weed out previously executed activations is not very efficient. For one, it can lead to a large amount of time being spent in the matching process. That is, the matcher may have to regenerate a large number of activations that will always be deleted from the conflict set, e.g., because they've been seen before. Also, what gets through to the conflict set should be activations that have a chance of being chosen; it is up to meta-knowledge in the form of the scheduling rules to choose the best. Here we are letting through activations that are noise, a by-product of our regeneration policy.

To get around these problems, we have included two fields in each ORBS rule that hold information relatively to triggering and activation creation. We have seen one, "status", already. When a rule's status is inactive, the matcher will not attempt to trigger it. The other is "meta-trigger". The meta-trigger field is filled with either

- *nil*: no information is available.
- *no-repeat*: do not allow the same activation to be chosen more than once.
- *1-instance*: during any cycle, stop generating activations after the first new one, i.e., one that is not on *executed-activations*. In other words, do not allow more than one activation from this rule in the conflict set during any one cycle. Also, don't allow repeats.
- *1-shot*: shut a rule off when any one of its activations has been executed.

Any ORBS process (a RHS action, a scheduling rule) can (re)set a rule's status or meta-trigger field. This is consistent with our view of a rule as just another data object.

Using the meta-trigger field in our small-fast rule, we get:

```
(defp small-fast
       (ring-osc -ro)
 test  (eq (send -ro 'speed) 'fast)
       (eq (send -ro 'area) 'small)
 ->
       (fact ro-cell inverter-3)
 :::
       author: simoudis
       sched-level: choose-inverter
       meta-trigger: 1-shot
       status: active)
```

That is, the choosing of an inverter is a one shot process. Once it is chosen, there will never be reason to make the choice again. Hence, turn off small-fast once it has executed.

no-repeat, 1-instance and 1-shot are first attempts to include triggering information in an ORBS rule (LOOPS attempts to keep similar types of information about its rules [4]). They are simple, and hence will not handle complex cases. For instance, small-fast is just one rule in a set of rules that choose an inverter type for a ring-oscillator. Others include small-slow, large-fast, large-slow. This set is mutually exclusive in that no more than one rule will actually trigger, *ever*. Hence, once one of them triggers, we'd like to turn all of the rest off. All we have done with small-fast is to turn it off when it triggers; all of the other rules in the set will continue to be processed by the matcher on every cycle. Since none will ever trigger, this is quite wasteful of the matcher's time. We could include code in each RHS to turn off all rules in the set. However, a cleaner solution is to mark the (any) set of rules as mutually exclusive. ORBS currently does not support this.

Using a related example, if we have more than one ring-oscillator, then we want the rule small-fast to be applied to both, i.e., we want an inverter to be chosen for both bindings of the pattern variable -ro. Because it is not sensitive to context, 1-shot would turn the rule off after only one inverter was chosen, i.e., after a single activation has been executed. We could resort to replacing 1-shot with no-repeat. Then, one activation would be chosen on each cycle. However, this leads to wasteful attempts to match the rule on subsequent cycles; we know the rule will never fire again after cells are chosen for the two inverters. What we'd like to say is turn yourself off after two executions. Better yet, turn yourself off when all ring-oscillators have an inverter (this would also handle our mutually exclusive rule-set problem above). Formalizing these problems is one of our current interests. In particular, we are working on a general triggering/scheduling model that will allow us to handle such cases (c.f. [7,8,9]). For now, if no-repeat, 1-instance or 1-shot are too weak, the user must resort to mixing control knowledge into the data base as seen in the goal relation of our small-fast rule in figure 1.

5. Interactive Development

ORBS takes a dynamic view of system development and debugging. Like Interlisp, ORBS gives the user the ability to patch bugs and continue on. This is in direct opposition to systems like PROLOG, OPS5, YAPS, and Hearsay III. Each of these systems uses a more traditional cycle of edit/load/compile/run. When a bug is encountered, the cycle is repeated. Conversely, ORBS allows the user to place break points at convenient locations during a run. When a break is reached, ORBS transfers control to the break-package. From here, the user may interrogate various portions of the computation state, modify the state, back up to previous states, and continue processing.

The effort we have placed in building an interactive development environment is motivated by our experience building expert systems in languages like OPS5, YAPS, and Hearsay III. We find that each of these languages resists an incremental approach to development. Unfortunately, this approach is exactly what is called for in the systems we have attempted to build. As Swartout and Balzer point out, you cannot hope to work out all the details of a problem before you commence development [10]; a complete problem description is defined only after building a prototype, testing it, changing the prototype, testing it, etc.

ORBS provides the user with a break package. The break package is called either when explicitly requested by the user through setting of break points, or when the system encounters an error. Once in the break package, the user may interrogate the current state, change it, and continue processing. First we will look at the types of break points available. These are settable (and unsettable) at any point in a computation:

- Break on <pattern> being added to the data base. <pattern> is a relation with pattern variables, and is exactly equivalent to a pattern in the LHS of a rule.
- Break on <pattern> being deleted from the data base.
- Break when <rule> matched. <rule> is the name of a specific rule, e.g., small-fast. The break package is called when both pattern and test match.
- Break when <rule> chosen. The break package is called when final scheduling rule returns conflict set that consists of single activation of <rule>.
- Break when <rule> fired. The break package called after activation of <rule> chosen, and all RHS actions executed.

- Break every cycle. The break package is called after activation chosen and RHS actions executed, i.e., right before matching starts.
- Break on system halting. The system halts when no activations remain in the conflict set.
- Break before scheduling. The break package is called when initial conflict set is built, but before scheduling rules are invoked.
- Break after scheduling. The break package is called when last scheduling rule returns a single activation, but before that activation is executed.
- Break after <scheduling rule>. The break package called after <scheduling rule> returns a conflict set.

The break points above allow a user to enter the break package at convenient times. Once in the break package, the user has the following options (some actions are dependent on the state of computation):

- Print bindings of an activation, i.e., what data base facts have been used to match LHS patterns.
- Pretty print the data base (for any cycle).
- Add a new fact to the data base.
- Delete a fact from the data base.
- Edit the data base (destructively edit an existing fact).
- Add a rule.
- Edit a rule, i.e., call the ORBS rule editor.
- Print conflict set. Pretty prints each activation in current conflict set.
- Execute <activation>. Manually schedule an activation in the conflict set for execution.
- Remove <activation> from the conflict set.
- Determine why <rule> did not fire in <cycle>. Answer is either "did not match" or name of scheduling rule that eliminated <rule> from the conflict set.
- Match <rule> against {<fact>}. User chooses subset of facts to match against <rule>. Answer is either "match" or LHS pattern(s) that failed to match.
- Single step the scheduling rules. The break package is called after each scheduling rule returns.
- Print process history. This includes, on a cycle by cycle basis, what relations were added, what relations were deleted, what activation was executed.
- Revert to a previous cycle.
- Continue processing.

One major component that is missing from our interactive model is a graphics interface. Current versions of the system use CRT technology. This makes it awkward to display component structures. Further, simultaneous display of ORBS components (the data base, conflict set, rules) is difficult. Finally, dynamic tracing of system execution is limited to non-graphical representations, i.e., text descriptions. Modern systems such as LOOPS have shown that each of these problems can be tackled by integrating bit-mapped screen graphics into the interactive system. Our medium term goal is to move ORBS onto a machine that supports bit-mapped graphics (a Symbolics 3600), and build the necessary graphics support.

6. Summary

ORBS is an attempt to marry the good ideas that have come out of expert-system language research with those that have come out of interactive, incremental, software development research [11]. A major goal is to provide an environment that matches representational complexity with problem complexity. We want simple problems to be handled with simple machinery (e.g.,a relational database). We want complex problems to have available more extensive representational machinery (e.g., object-oriented programming).

We strongly feel that system construction most profitably follows an incremental approach. It is infeasible to work out all details of a problem before commencing testing. Our experience shows that a prototype or strawman should quickly be constructed, and used to highlight missing knowledge or potential problem areas. The ORBS model of interactive development is based on these ideas.

Another goal of ORBS, and one that effects both complexity and incremental development, is the reduction of setup costs to the user. In languages such as Hearsay III, there is a large amount of detail that must be defined and debugged before testing can commence. Languages like PROLOG have virtually no initial startup costs. In ORBS, we are trying to strike a balance. We are attempting to provide complex machinery, but at the same time allow the user to ignore that complexity until he or she needs to deal with it (sometimes never). To strike this balance, we follow two general ideas.

(1) Each new system we build tends to *reuse* parts of previous systems. For instance, many problems require a priority queue or agenda based scheduler. So we catalog reusable scheduling rules. One of our long range goals is the cataloging of reusable domain knowledge as well.

(2) Through judicious use of defaults, many of the decisions that a user may wish to make later in the development process (e.g., after several prototypes have been tested) can be suspended. Thus, early systems may be simple, but inefficient. Once later systems become more solid, efficiency concerns can be addressed.

The ORBS system is being implemented on a VAX using the Maryland extension to Franzlisp [12]. The major components of the system (rules, data base, facts, activations, etc.) are flavor objects; computation is message-based. We are currently translating the Franz/Maryland implementation to Zetalisp on the Symbolics 3600.

7. Research Plan

A working prototype of the ORBS system currently exists. In this section, we will describe our future efforts. Our research plan can be broken into short term (1 year or less) and long term (more than 1 year) goals.

Short term:

1. Port the system to the Symbolics 3600.

2. Construct a prototype graphics interface on the 3600.

3. Functionally test the system on varied domains. This will be tied in with the graduate Expert Systems course.

4. Improve the matching algorithm. This is a non-trivial problem given ORBS interactive, incremental model. For instance, it is assumed that rules, the data base, and cycles (in the process of backtracking) will all be changed as the system runs. Efficient matching algorithms like those of OPS5 and YAPS demand a static, non-incremental model.

Long term:

1. Extend the graphics interface to include dynamic displays such as those found in LOOPS.

2. Explore the reuse of performance knowledge. Our overall goal is a scheduling assistant that will help a user build a scheduling strategy to match the problem domain. This will include cataloging general scheduling rules such as those found in [7,8,9].

3. Explore the reuse of competence knowledge. We conjecture that different systems within the same domain will likely have common objects and rules. We are attempting to catalog skeleton or schematic forms of domain knowledge for reuse on new systems; [13] discusses our approach in more detail. We are currently looking at the domains of office systems, and transportation systems.

Acknowledgments

Members of the ORBS project include Michael Hennessy, David Novick, and Rob Reesor from the University of Oregon, and Bill Bregar from Oregon State University. Through discussions and coding, they have been a major influence on the shape of ORBS to date.

Jane and Dana Laursen have also helped with various coding tasks.

8. References

[1] Erman, L., London, P., Fickas, S.
The design and example use of Hearsay III,
In *7th International Joint Conference on AI*, Vancouver, 1981

[2] Allen, E.
YAPS: Yet Another Production System,
TR 1146, Computer Science Dept, University of Maryland, 1283

[3] Griener, R., Lenant, D.
A representation language language
In *1st National Conference on AI*, Stanford, 1980

[4] Bobrow,D., Stefik, M.
The LOOPS Manual,
Xerox Parc, Palo Alto, 12//83

[5] Weinreb, D, Moon, D.
Objects, Message Passing, and Flavors,
Lisp Machine Manual, Ch. 20, Symbolics Inc., 1981

[6] Forgy, C.,
OPS5 User's Manual,
Tech Report, Computer Science Dept, CMU, 1981

[7] McDermott, J., Forgy, C.
Production system conflict resolution strategies,
In *Pattern-Directed Inference Systems, Academic Press, 1978*

[8] Clancey, W.
The Advantages of Abstract Control Knowledge in Expert System Design,
Tech Report HPP-83-17, Computer Science Dept, Stanford, 11/83

[9] Genesereth, M.
Meta-Level Architecture,
Memo HPP-81-6, Computer Science Dept, Stanford, 12/82

[10] Swartout, W., Balzer, R.
On the inevitable intertwining of specification and implementation,
CACM **25**(7) (1982)

[11] Shiels, B.
Power tools for programmers,
In *Interactive Programming Environments*, McGraw-Hill, 1984

[12] Allen, E., Trigg, R., Wood, R.
Maryland Franzlisp Environment,
TR 1226, Computer Science Dept, University of Maryland, 1183

[13] Fickas, S.
Specification Automation,
In *Workshop on Models and Languages for Software Specification*, Orlando, 1984

A System to Hardcopy Screen Images of the Symbolics 3600 LISP Machine

J.M. Wilczynski, K. Chen, D. Meyer, R. Reesor

Department of Computer and Information Science
University of Oregon

A System to Hardcopy Screen Images of the Symbolics 3600 LISP Machine

J.M.Wilczynski, K.Chen, D.Meyer, R.Reesor

Department of Computer and Information Science
University of Oregon

ABSTRACT

This document describes a system that allows screen images from a Symbolics 3600 LISP Machine to be sent to an Imagen Imprint-10 laser printer by way of a VAX 11/750.

The introduction gives an overview of the problem and the steps taken to resolve it. Then the software changes and additions, both on the Symbolics 3600 and on the VAX, are covered in detail. Appendices contain examples of code and printed screen images.

July 31, 1984

A System to Hardcopy Screen Images of the Symbolics 3600 LISP Machine

J.M.Wilczynski, K.Chen, D.Meyer, R.Reesor

Department of Computer and Information Science
University of Oregon

1. Introduction

Three Symbolics 3600 LISP machines were recently added to our department's site configuration. Our environment had previously consisted of two VAX 11/750's, running UNIX 4.1 bsd, with a LP-25 printer and an Imagen 10 Laser Printer (ILP) hardwired to one VAX (VAX2, see Appendix A) and accessible to the other VAX (VAX1) by way of the UUCP network. One of the many things the LISP machines are used for is bitmap graphics. Although a laser printer could be purchased from Symbolics to be hardwired to the new LISP machines, it was felt that we should be able to utilize the ILP that was already installed if we could somehow make the Symbolics 3600's interface with it.

This report gives an overview of the process of interfacing the two systems. The five main steps in the process are as follows.

First, there was a general period of research. This included gaining familiarity with the LISP machines, the ILP, and what use could be made of the fact that we had to use the VAX2 as a relay. We had to learn how the bitmap graphics were represented on the LISP machine, how we might copy a screen image to a LISP machine file, and send it off to the ILP by way of the VAX2. A major task here was to discover if the format of the bitmap leaving the Symbolics 3600 was acceptable to be received by the ILP. As it turned out, it wasn't; the bits needed to be swapped before being sent to the ILP. This swapping could take place either on the Symbolics 3600 or on the VAX2 system. We decided that in order to get a prototype of the new system up and running, we would do the work on the Symbolics 3600. This seemed sensible in that we could prepare everything using the Symbolics 3600 and then send the bitmap file to some VAX2 directory to be later sent to the ILP.

Next we conducted a search of the software provided with the Symbolics 3600 LISP machines to see if anything might be useful to our project. Although many functions relating to copying screen images were referenced in the LISP machine documentation, we found that Symbolics had not provided most of them at our site, presumably since we did not have the Symbolics laser printer for which the software had been designed. We did, however, find a few things to build upon and set about to do so.

At the same time that the second step, described above, was happening, we were experimenting with the ILP by hand building some small bitmaps on the VAX2 and trying to print them. IMPRINT-10, the operating system of the ILP, uses a low level machine language called IMPRESS-10 which allows commands to be inserted into a file before being sent to the ILP so that the ILP will know the format of the information it is receiving. There was also some experimentation here to see if IMPRESS-10 might be able to swap the bits instead of doing it on the LISP machine, but this didn't seem plausible.

At this point we had a group of ZETALISP functions that could be compiled that allowed a screen image to be copied into a file, after swapping the ordering of the bits. This file could be on the LISP machine's file system or on the VAX2's file system; any file could be specified. Next we had to log on to the VAX2 and add IMPRESS-10 commands and then send the file to the ILP. This process was obviously unwieldy since the user had to stop work on the LISP machine and switch to the VAX2 if the output was desired right away. It also turned out that the bit swapping on the LISP machine was slow, but the prototype was running and usable.

Finally, we decided that we needed to automate the process so that the user could hit some sequence of keys on the LISP machine and the screen image would be printed on the ILP with no intervention. This turned out to be a good place to make further use of the VAX2. The bit swapping was moved from the LISP machine to a C program on the VAX2. Also, instead of having the user give a file name where to send the bitmap, a file was built and automatically sent to a VAX2 directory. A spooler was written to run in this directory so that it would collect any bitmaps coming from the LISP machines. Finally, a shell script was designed that combined the bit swapping program with code to add the IMPRESS-10 commands and send the final product to the ILP.

2. Symbolics 3600 Software Support

The Symbolics 3600 is a computer system in which each active user is assigned a medium scale processor, a suitable amount of memory, and a swapping disk. Files are stored in a centralized file system accessed through CHAOSNET. CHAOSNET is used to access other shared resources in addition to the file system (e.g. printers, tape drives, processors, and I/O devices). Each network node (CHAOSNET) consists of the transceiver, interface, and a computer which executes the Network Control Program (NCP).

The Symbolics 3600 CHAOSNET support consists of a set of ZETALISP functions and data structure definitions in the CHAOS package. The NCP on the VAX is implemented entirely in the kernel as a device driver and is accessed from user programs with the normal I/O system calls (packets received from the network are processed at interrupt level). Stream mode (default for opening CHAOS device) makes the connection behave like a UNIX file.

Our configuration (see Appendix A) consists of three Symbolics 3600's, one VAX 11/750 (VAX2) accessible over the CHAOSNET using the ETHERNET protocol, and another VAX (VAX1) accessible over the UUCP net. The ILP is connected, using an RS-232C serial interface, to the VAX2. Because the Symbolics 3600 file system is accessible over the CHAOSNET, all we had to do after we created a bitmap was to define a pathname specifying a directory, either on the VAX2 or the Symbolics, to put the file in. However, the Symbolics 3600 differs from the VAX in the way that it handles the serial I/O stream data transfers. The Symbolics 3600 character set, using 8 bits per byte, differs from the 7 bit ASCII set. Most devices that are likely to use serial communications use the standard ASCII set. The serial I/O stream of the Symbolics 3600 is also different from the other streams in that it is buffered on the output side using LGP:TYO, the method of the BASIC-LGP-STREAM flavor. Only after the output buffer is filled, or if the stream is closed, are the characters transmitted.

If we define a connection between two users as a principal service provided by the CHAOSNET, we can define a stream as a standard I/O stream which transmits to and receives from a connection using a 16 bit number as an address for each node, or host. As we said before, stream mode (default for opening the stream) makes the connection behave like a UNIX file. The host name, always specified in the pathname for the file either on the VAX or the Symbolics 3600, serves as an address of the host's file system where the host's name serves as the address for the CHAOSNET connection.

Our hardcopy system was distributed over the CHAOSNET using different file systems and processors (Symbolics 3600 and VAX2 with peripherals). The Symbolics 3600, which provides high resolution bitmap graphics, is the source of the images which can be displayed on a high resolution terminal screen or with the use of a laser printer for even better resolution. Originally, the Symbolics 3600 included a hardcopy system in the LGP package with some constructor functions defined in the TV package. This hardcopy system, and in particular the part which defines bitmap printing, was not applicable in its current state to our configuration. The part of the original hardcopy system for printing bitmaps was set up to communicate with a Symbolics Laser Graphics Printer (LGP) which, although based on the same MC68000 microprocessor, uses a different driving program. In particular, the way that the bitmap is represented in the image's memory is different. The self diagnostics, driving program, and built in fonts are contained in the

ROM storage (up to 65556 bytes) for both printers. Also since the ILP differs from the LGP as far as the instruction set is concerned, we had to include different instructions with the bitmap being sent over the net. We had to change the following in the Symbolics provided hardcopy system:

1. Replace "old" LGP related instructions with new ones.

2. Structure the representation of the image (bitmap) into a set of fragments (windows) according to the ILP's format.

3. Open the CHAOSNET connection: Symbolics-VAX2, to send the image to a specified directory on the VAX2.

It turned out that since packets are transmitted over the transmission medium (coaxial cable used be the CHAOSNET) in reverse bit order and not restored in the case of bitmap transmission (in the receiving host), the bit reversal program had to be written as well. It was decided that the creation of a bitmap and structuring it for the ILP with only the basic IMPRESS-10 command, BITMAP, would be done on the Symbolics 3600. Adding the ILP's required instructions, in IMPRESS-10, and the bit reversal program would be done on the VAX2.

3. The Symbolics 3600 part of the hardcopy system

The Symbolics 3600 part of the *modified* hardcopy system, for release 4.5, contains two files. The first one (included in the LGP package) contains a totally new ZETALISP program including two newly defined methods. The first method LGP:SEND-COMMAND1 (see Appendix B) is defined for the BASIC-LGP-STREAM flavor and defines a stream operation, for SI:OUTPUT-STREAM, of passing a character to the stream. This method was defined to pass to the stream (the object of BASIC-LGP-STREAM flavor) the IMPRESS-10 command, BITMAP, with the type of operation and the dimensions of the array holding the bitmap as arguments. Next, the original bitmap, in the form of an array of bits, had to be converted into an array of 8 bit bytes within the LGP:SHOW-BITMAP method of the LGP-BITMAP-STREAM flavor. The method, LGP:SHOW-BITMAP, was rewritten entirely. The created array of bytes was structured according to IMPRINT-10 acceptable format.

The difference in the handling of bitmaps by the two printers is in the way that the bitmap is structured by the operating system of each printer. The ILP requires it to be structured as 32x32 subarrays of pixels. These subarrays, or windows, are used as the smallest units of the bitmap manipulation in IMPRESS-10. The two arguments given to IMPRESS-10's BITMAP command specify the width and height of the bitmap in 32x32 subarray of pixel units. For example, the command BITMAP 20 20 specifies a bitmap which has 20 rows of 20 32x32 windows.

The LGP format for handling the bitmaps is different. The word argument of the comparable command specifies the number of predefined scan lines to be sent. Each scan line consists of 64 words, 16 bits each, of raster data, which using the resolution of either laser printer (240 pixels per inch) gives a 4.2 inch long line which is shorter than the standard line for 8x11 format. So for the LGP the unit window was 16x16 pixels/bits.

As we mentioned earlier, since the instruction sets are different for the two printers, the LGP:SHOW-BITMAP function was rewritten. First, the IMPRESS-10 BITMAP command was added with the appropriate arguments using the previously defined method LGP:SEND-COMMAND1. Since IMPRESS-10 is a low level machine language, each command is represented as a 1 or 2 byte number. The appropriate numbers (command and arguments) were output to the stream, SI:OUTPUT-STREAM, one byte at a time using the LGP:TYO method of the BASIC-LGP-STREAM flavor. Within the old hardcopy system, the LGP:SHOW-BITMAP method of the LGP-BITMAP-STREAM flavor was used to format a bitmap for the LGP printer.

Another file, included in the TV package, holds functions which use the LGP:SHOW-BITMAP method. These functions actually copy the screen (create bitmaps) and use the

LGP:SHOW-BITMAP method to open the destination stream. There are four ZETALISP functions and one data structure which do this (see Appendix C). There was no need to change the functions which create the bitmap (an array of bits representing a one-to-one mapping of pixels of the screen's image onto the elements of the array). We only had to change the function which opens an I/O stream using the WITH-OPEN-STREAM macro. Since the stream mode used by default makes a CHAOS connection which acts like a UNIX file, we in fact defined, by specifying a pathname, an address to store our bitmap in the form of a file. Since it was our intention to do all additional processing on the VAX2 and since the ILP communicates only with the VAX2, we defined the pathname to specify the VAX file name VAX:/usr/spool/ipcd/changingpart.bit so the bitmap would be placed directly into the /usr/spool/ipcd directory.

The changing part of the pathname was generated using the time function and parsing the output string (hrs/minutes/seconds). This was done using the ZETALISP function TV:OUT-FILE (see Appendix C).* So at 13:40:15, the file created by the function TV:OUT-FILE would be VAX:/usr/spool/ipcd/134015.bit. Now with the stream defined to be a file on the VAX, we could execute the LGP:SHOW-BITMAP method to actually create the file.

All the methods needed to create a bitmap are included in the ZETALISP function KBD-ESC-Q (see Appendix C) which can be considered to be the main driver of both the old and the modified hardcopy systems. In addition, it defines three possible options for hardcopy operations:

1. Copy selected window.
2. Copy main screen.
3. Copy main screen and who-line.

4. VAX Software Support

The VAX software for the above project consists of a spooler, a few auxiliary programs, and a "shell script" to tie them all together. In this section, we describe the general problem, and our solution and then describe the components of the system.

4.1. The General Problem

After the logistics of providing the bitmaps (to the VAX2) to be printed on the ILP were worked out, the process of printing a bitmap needed to be automated. The general problem, then, was to design a system that would be aware of bitmaps comming over the CHAOSNET (from the Symbolics 3600's) to the VAX2. Note that the "ideal" solution to the problem would be to design a system much like the printer spooler, that is, have the requesting program "exec" the daemon, if one was not already active (*i.e.*, if a line printer daemon is not running when one types "lpr", one is started up to service the newly spooled print request). The problem, however, was that we couldn't implement a method for starting up a process under UNIX from the Symbolics (that is, over the CHAOSNET), given our time frame (note that solving this problem leads to an efficient spooling system).

4.2. The General Solution

Since we couldn't start a process on the VAX2 from the Symbolics, we decided to implement a spooler that is always around, and check a specified directory for work (note that this can be done from cron, so that the daemon doesn't have to run continuously). Thus, the overview of spooler operation is as follows:

(i). One or more files are written into a spool directory (/usr/spool/ipcd on our system);

* This function was contributed to the project by Will Goodwin and Allen Brookes.

(ii). The spooler (/usr/local/bin/ipcd) notices
that files are present;

(iii). Ipcd then reads the directory, and "exec's"
an IPCOPY (actually, it exec's an ipc, which is
described below) for each spooled file;

(iv). Finally, ipcd cleans up the spool directory
by deleting the work (spooled) files.

4.3. Auxillary programs

(i). swap.c --
Maps Symbolics bitmaps onto ILP bitmap format.
Note that the mapping is table driven, and the
"swapping" table is generated by mkswaptb.c.

(ii). ipc --
A "shell script" to put all of the above
together; Ipc also performs functions such
as concatenating the header and trailer files
to the remapped bitmaps (IMPRESS-10 control
information); One ipc is "exec'ed" by ipcd for
each bitmap it finds in its spool directory.
Note that after the bitmaps are processed, the
result is "piped" down to ipr, the Imagen supplied
spooler for ILP format bitmaps. What follows is
a picture of ipc:

```
#! /bin/sh

#
#
#    ipc --
#
#    Ipc is a script to take input files from the Symbolics
#    3600 LISP machines and format them for printing on the
#    ILP.
#
#    There are 3 basic auxiliary files:
#    (i).   /usr/local/bin/swap              -- maps bits into ILP format
#    (ii).  /usr/local/src/uo/lib/head.imp   -- ILP header format
#    (iii). /usr/local/src/uo/lib/tail.imp   -- ILP trailer format
#
#    The source code for all of the ipcopy system can be found in
#    /usr/local/src/uo/ipcopy
#
#

LIB=/usr/local/src/uo/lib              # headers and trailers here
BIN=/usr/local/bin                     # executable code here
SPOOL=/usr/spool/ipcd                  # spool directory
```

```
if [ $# = 0 ]; then                     # any file args?
  echo ipc: no args found > /dev/console      # complain
  exit 1                                # no...get out
fi
trap 'rm -f /tmp/$$.imp;exit 1' 1 2 3 14 15   # trap some signals
rm -f /tmp/$$.imp                       # just in case
look
if [ -f $SPOOL/$1 ]; then               # only do this if we have a file
  $BIN/swap < $SPOOL/$1 > /tmp/$$.imp            # swap the bits around
  cat $LIB/head.imp /tmp/$$.imp $LIB/tail.imp | ipr     # and ipr it
else
  echo ipc: can´t get at $SPOOL/$1 > /dev/console     # register complaint
  exit 1                        # no file...crash and burn
fi

rm -f /tmp/$$.imp               # get rid of our tmp file
```

5. Concluding Remarks

The modified hardcopy system for the Symbolics 3600 release 4.5 was finally saved, after thorough testing, in the form of a world load on one of our Symbolics 3600's (LM3).

Before the world load was actually saved, a system of the two binary files that hold the compiled code was added to the original world load (release 4.5). Now we had a world load with the hardcopy functions, either rewritten or modified, available in compiled form on the system. By saving the modified version of the world load, we made the modified hardcopy system immediately available to all users, after initial booting (see Appendix E).

Acknowledgements

We would like to thank Steve Fickas, Mike Hennessy, and Kent Stevens from our department and Jan Stoeckenius from Imagen Inc. for their help and critical suggestions.

References

[1] R.Mathews and S.Finkel, *Program Development Tools and Techniques,* Symbolics, 1983.

[2] D.Moon, *Chaosnet,* Technical Report of Artificial Intelligence Laboratory, MIT, 1982.

[3] S.Reisler, *LGP-1 Laser Graphics Printer,* Technical Manual, Symbolics, 1982.

[4] C.Roads, *3600 Technical Summary,* Symbolics, 1981.

[5] C.Ryland, E.Martinez, J.Stoeckenius, J.Tein, *Imprint-10 System Manual,* Imagen Inc., 1983.

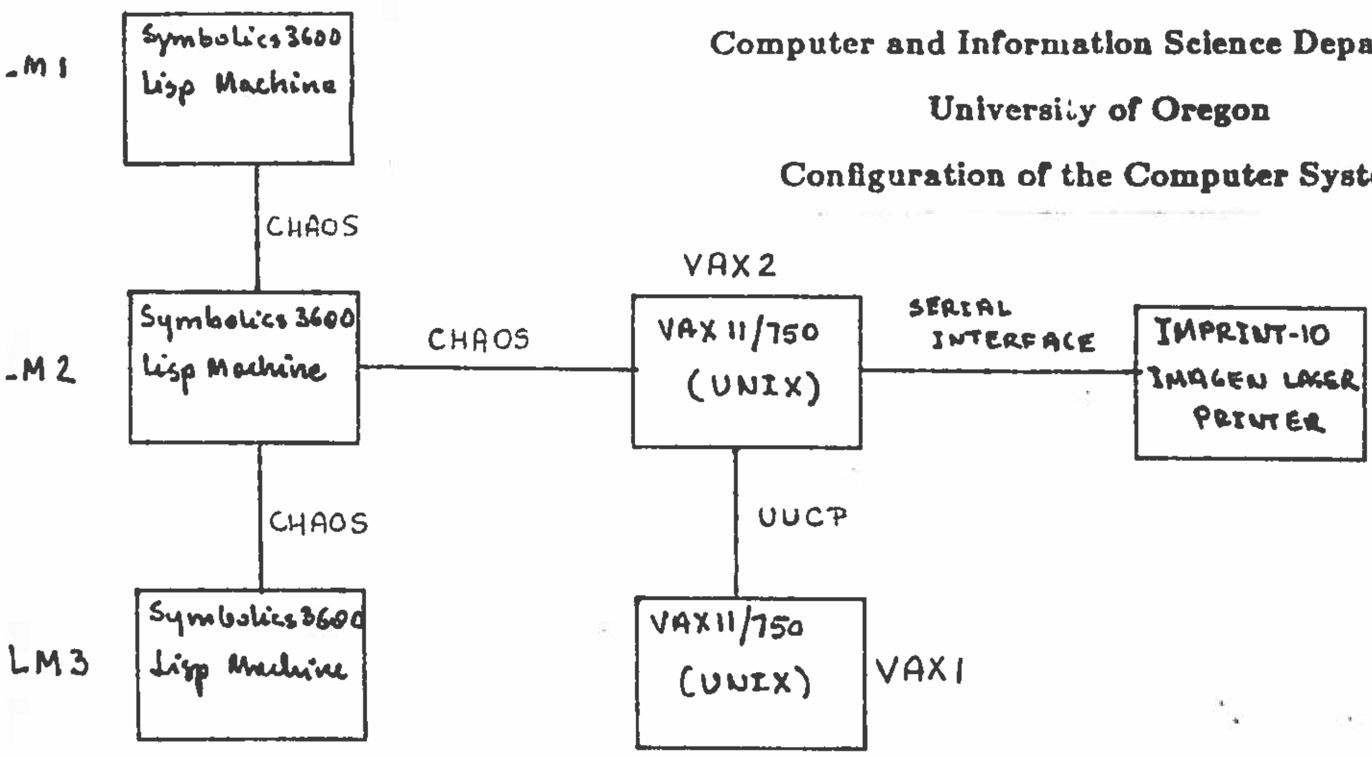
[6] D.Weinreb and D.Moon, *LISP Machine Manual,* MIT, 1981.

Appendix A

Computer and Information Science Department

University of Oregon

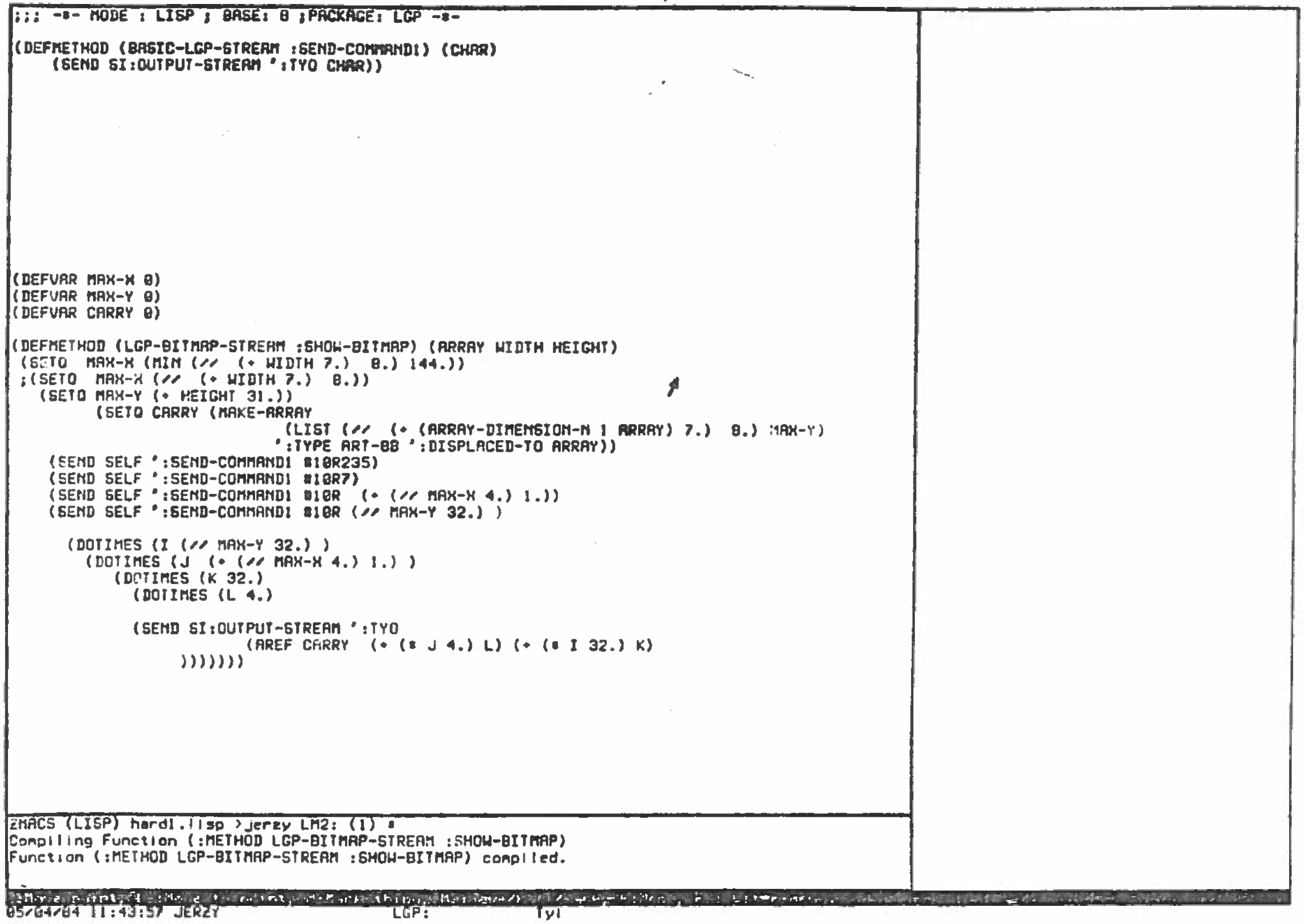
Configuration of the Computer Systems



LM1
Symbolics 3600 Lisp Machine
CHAOS
LM2
Symbolics 3600 Lisp Machine
CHAOS
LM3
Symbolics 3600 Lisp Machine
CHAOS
VAX2
VAX 11/750 (UNIX)
SERIAL INTERFACE
IMPRINT-10 IMAGEN LASER PRINTER
UUCP
VAX11/750 (UNIX)
VAX1

LGP:SHOW-BITMAP and LGP:SEND-COMMAND1 methods

defined for the modified hardcopy system

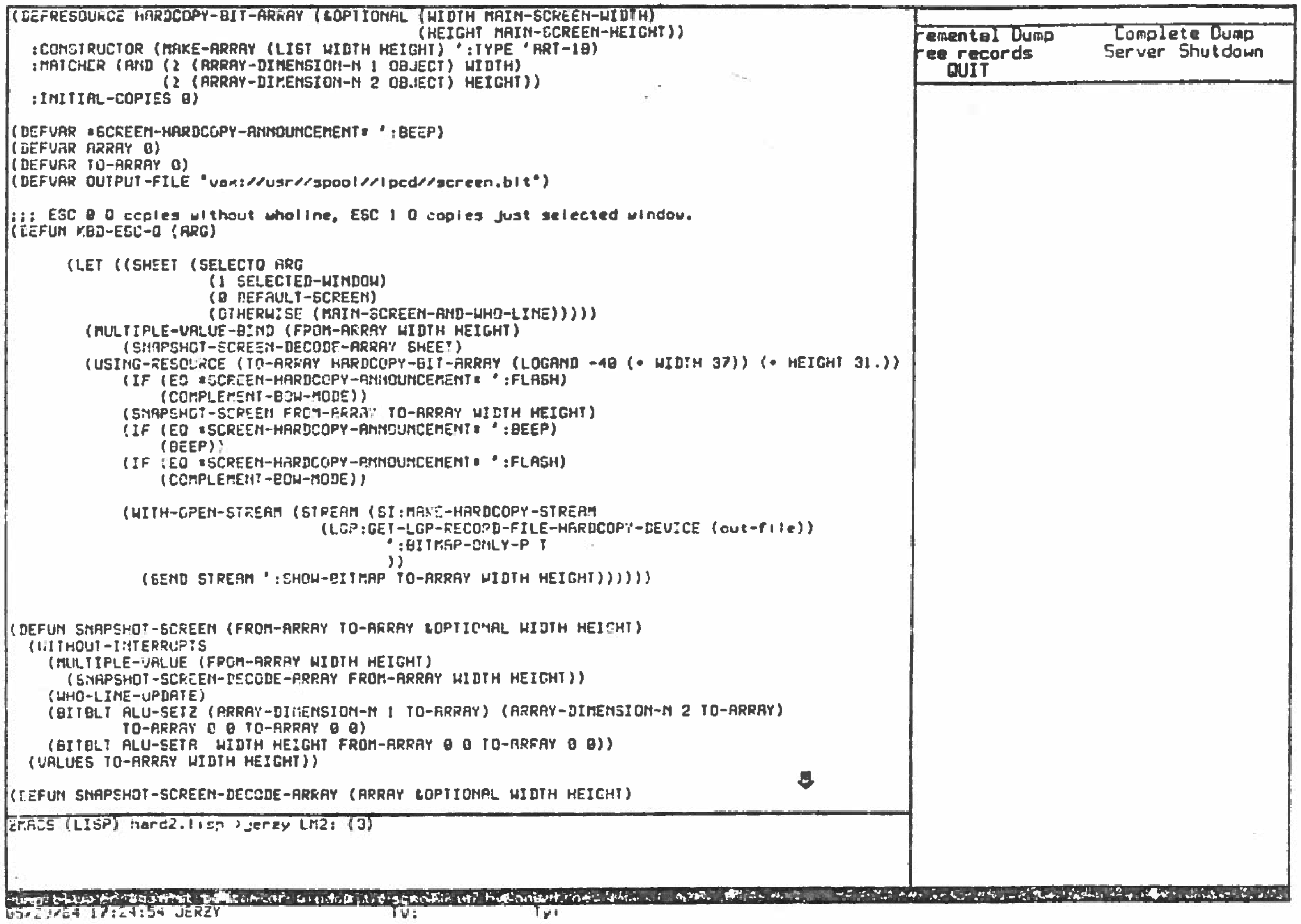


```
;;; -*- MODE : LISP ; BASE: 8 ;PACKAGE: LGP -*-

(DEFMETHOD (BASIC-LGP-STREAM :SEND-COMMAND1) (CHAR)
    (SEND SI:OUTPUT-STREAM ':TYO CHAR))











(DEFVAR MAX-X 0)
(DEFVAR MAX-Y 0)
(DEFVAR CARRY 0)

(DEFMETHOD (LGP-BITMAP-STREAM :SHOW-BITMAP) (ARRAY WIDTH HEIGHT)
 (SETQ  MAX-X (MIN (//  (+ WIDTH 7.)  8.) 144.))
 ;(SETQ  MAX-X (//  (+ WIDTH 7.)  8.))
   (SETQ MAX-Y (+ HEIGHT 31.))
         (SETQ CARRY (MAKE-ARRAY
                                    (LIST (//  (+ (ARRAY-DIMENSION-N 1 ARRAY) 7.)  8.) MAX-Y)
                                   ':TYPE ART-8B ':DISPLACED-TO ARRAY))
    (SEND SELF ':SEND-COMMAND1 #10R235)
    (SEND SELF ':SEND-COMMAND1 #10R7)
    (SEND SELF ':SEND-COMMAND1 #10R  (+ (// MAX-X 4.) 1.))
    (SEND SELF ':SEND-COMMAND1 #10R (// MAX-Y 32.) )

      (DOTIMES (I (// MAX-Y 32.) )
        (DOTIMES (J  (+ (// MAX-X 4.) 1.) )
           (DOTIMES (K 32.)
             (DOTIMES (L 4.)

             (SEND SI:OUTPUT-STREAM ':TYO
                         (AREF CARRY  (+ (* J 4.) L) (+ (* I 32.) K)
                  )))))))
```

```
ZMACS (LISP) hard1.lisp >jerzy LM2: (1) *
Compiling Function (:METHOD LGP-BITMAP-STREAM :SHOW-BITMAP)
Function (:METHOD LGP-BITMAP-STREAM :SHOW-BITMAP) compiled.

[illegible]
05/04/84 11:43:57 JERZY                LGP:          Tyi
```

Appendix C

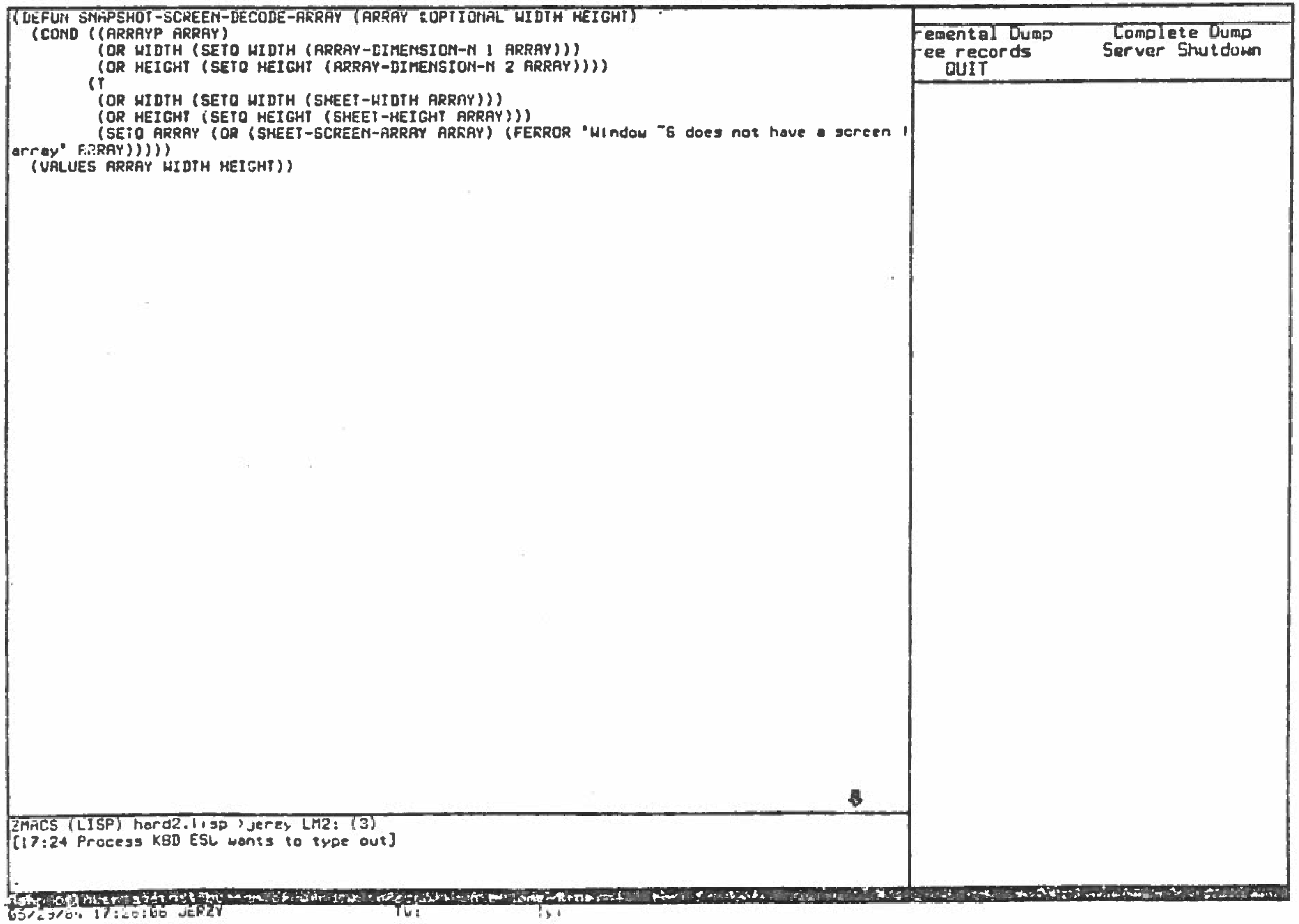
Methods to Create a Bitmap on Symbolics 3600



```
(DEFRESOURCE HARDCOPY-BIT-ARRAY (&OPTIONAL (WIDTH MAIN-SCREEN-WIDTH)
                                           (HEIGHT MAIN-SCREEN-HEIGHT))
  :CONSTRUCTOR (MAKE-ARRAY (LIST WIDTH HEIGHT) ':TYPE 'ART-1B)
  :MATCHER (AND (= (ARRAY-DIMENSION-N 1 OBJECT) WIDTH)
                (= (ARRAY-DIMENSION-N 2 OBJECT) HEIGHT))
  :INITIAL-COPIES 0)

(DEFVAR *SCREEN-HARDCOPY-ANNOUNCEMENT* ':BEEP)
(DEFVAR ARRAY 0)
(DEFVAR TO-ARRAY 0)
(DEFVAR OUTPUT-FILE "vax://usr//spool//lpcd//screen.bit")

;;; ESC 0 O copies without wholine, ESC 1 O copies just selected window.
(DEFUN KBD-ESC-O (ARG)

      (LET ((SHEET (SELECTQ ARG
                     (1 SELECTED-WINDOW)
                     (0 DEFAULT-SCREEN)
                     (OTHERWISE (MAIN-SCREEN-AND-WHO-LINE)))))
        (MULTIPLE-VALUE-BIND (FROM-ARRAY WIDTH HEIGHT)
            (SNAPSHOT-SCREEN-DECODE-ARRAY SHEET)
        (USING-RESOURCE (TO-ARRAY HARDCOPY-BIT-ARRAY (LOGAND -40 (+ WIDTH 37)) (+ HEIGHT 31.))
            (IF (EQ *SCREEN-HARDCOPY-ANNOUNCEMENT* ':FLASH)
                (COMPLEMENT-BOW-MODE))
            (SNAPSHOT-SCREEN FROM-ARRAY TO-ARRAY WIDTH HEIGHT)
            (IF (EQ *SCREEN-HARDCOPY-ANNOUNCEMENT* ':BEEP)
                (BEEP))
            (IF (EQ *SCREEN-HARDCOPY-ANNOUNCEMENT* ':FLASH)
                (COMPLEMENT-BOW-MODE))

            (WITH-OPEN-STREAM (STREAM (SI:MAKE-HARDCOPY-STREAM
                                   (LGP:GET-LGP-RECORD-FILE-HARDCOPY-DEVICE (out-file))
                                          ':BITMAP-ONLY-P T
                                          ))
              (SEND STREAM ':SHOW-BITMAP TO-ARRAY WIDTH HEIGHT))))))


(DEFUN SNAPSHOT-SCREEN (FROM-ARRAY TO-ARRAY &OPTIONAL WIDTH HEIGHT)
  (WITHOUT-INTERRUPTS
    (MULTIPLE-VALUE (FROM-ARRAY WIDTH HEIGHT)
      (SNAPSHOT-SCREEN-DECODE-ARRAY FROM-ARRAY WIDTH HEIGHT))
    (WHO-LINE-UPDATE)
    (BITBLT ALU-SETZ (ARRAY-DIMENSION-N 1 TO-ARRAY) (ARRAY-DIMENSION-N 2 TO-ARRAY)
            TO-ARRAY 0 0 TO-ARRAY 0 0)
    (BITBLT ALU-SETA  WIDTH HEIGHT FROM-ARRAY 0 0 TO-ARRAY 0 0))
  (VALUES TO-ARRAY WIDTH HEIGHT))

(DEFUN SNAPSHOT-SCREEN-DECODE-ARRAY (ARRAY &OPTIONAL WIDTH HEIGHT)
```

```
ZMACS (LISP) hard2.lisp >jerzy LM2: (3)
```

```
05/23/84 17:24:54 JERZY                 Tv:             Ty:
```

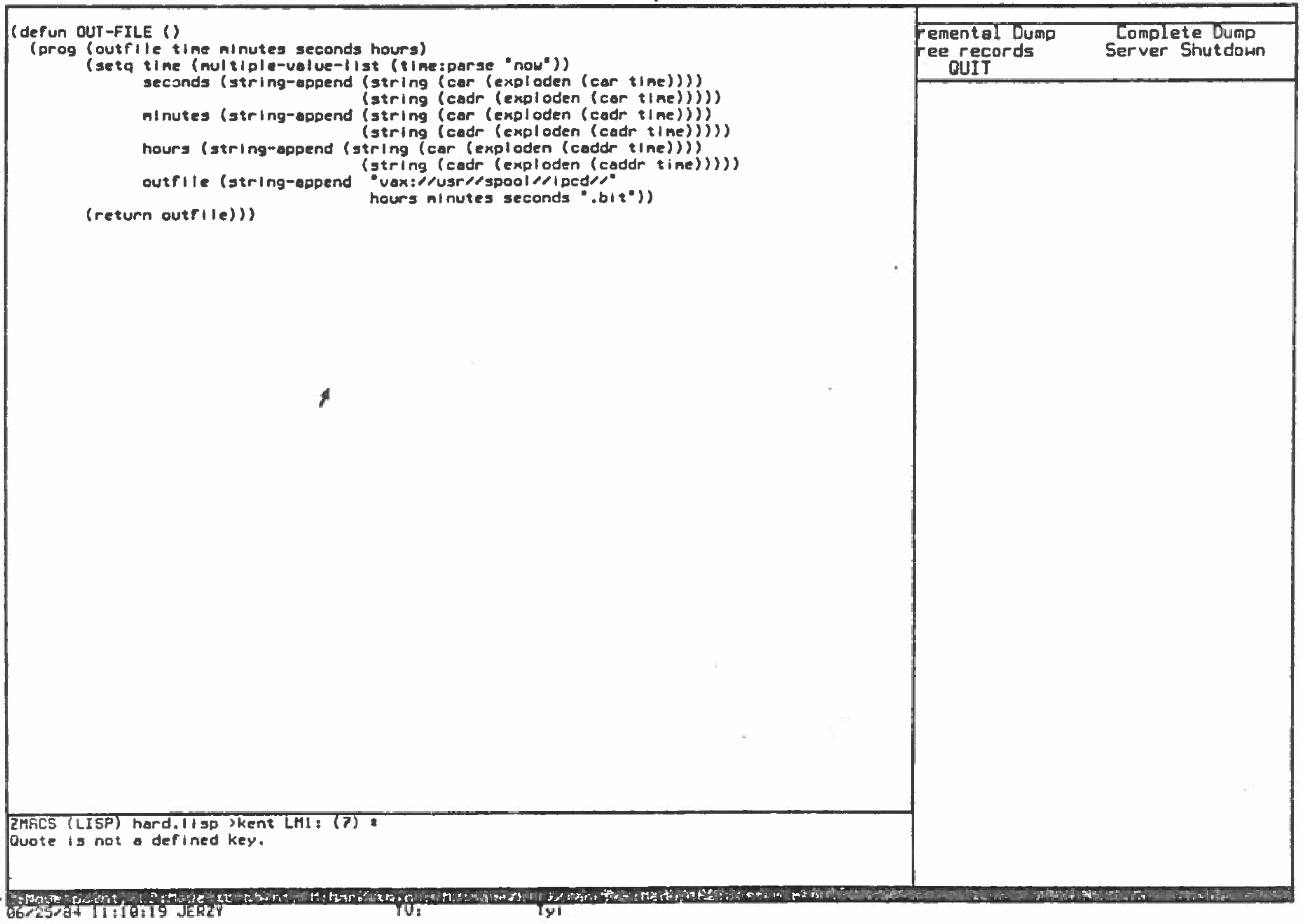


```
(DEFUN SNAPSHOT-SCREEN-DECODE-ARRAY (ARRAY &OPTIONAL WIDTH HEIGHT)
  (COND ((ARRAYP ARRAY)
         (OR WIDTH (SETQ WIDTH (ARRAY-DIMENSION-N 1 ARRAY)))
         (OR HEIGHT (SETQ HEIGHT (ARRAY-DIMENSION-N 2 ARRAY))))
        (T
         (OR WIDTH (SETQ WIDTH (SHEET-WIDTH ARRAY)))
         (OR HEIGHT (SETQ HEIGHT (SHEET-HEIGHT ARRAY)))
         (SETQ ARRAY (OR (SHEET-SCREEN-ARRAY ARRAY) (FERROR "Window ~S does not have a screen !
array" ARRAY)))))
  (VALUES ARRAY WIDTH HEIGHT))
```

```
ZMACS (LISP) hard2.lisp >jerzy LM2: (3)
[17:24 Process KBD ESC wants to type out]
```

```
05/29/84 17:26:06 JERZY              Tv:            [illegible]
```

APPENDIX C (CONTINUED)



```
(defun OUT-FILE ()
  (prog (outfile time minutes seconds hours)
        (setq time (multiple-value-list (time:parse "now"))
              seconds (string-append (string (car (exploden (car time))))
                                     (string (cadr (exploden (car time)))))
              minutes (string-append (string (car (exploden (cadr time))))
                                     (string (cadr (exploden (cadr time)))))
              hours (string-append (string (car (exploden (caddr time))))
                                     (string (cadr (exploden (caddr time)))))
              outfile (string-append  "vax://usr//spool//ipcd//"
                                      hours minutes seconds ".bit"))
        (return outfile)))
```

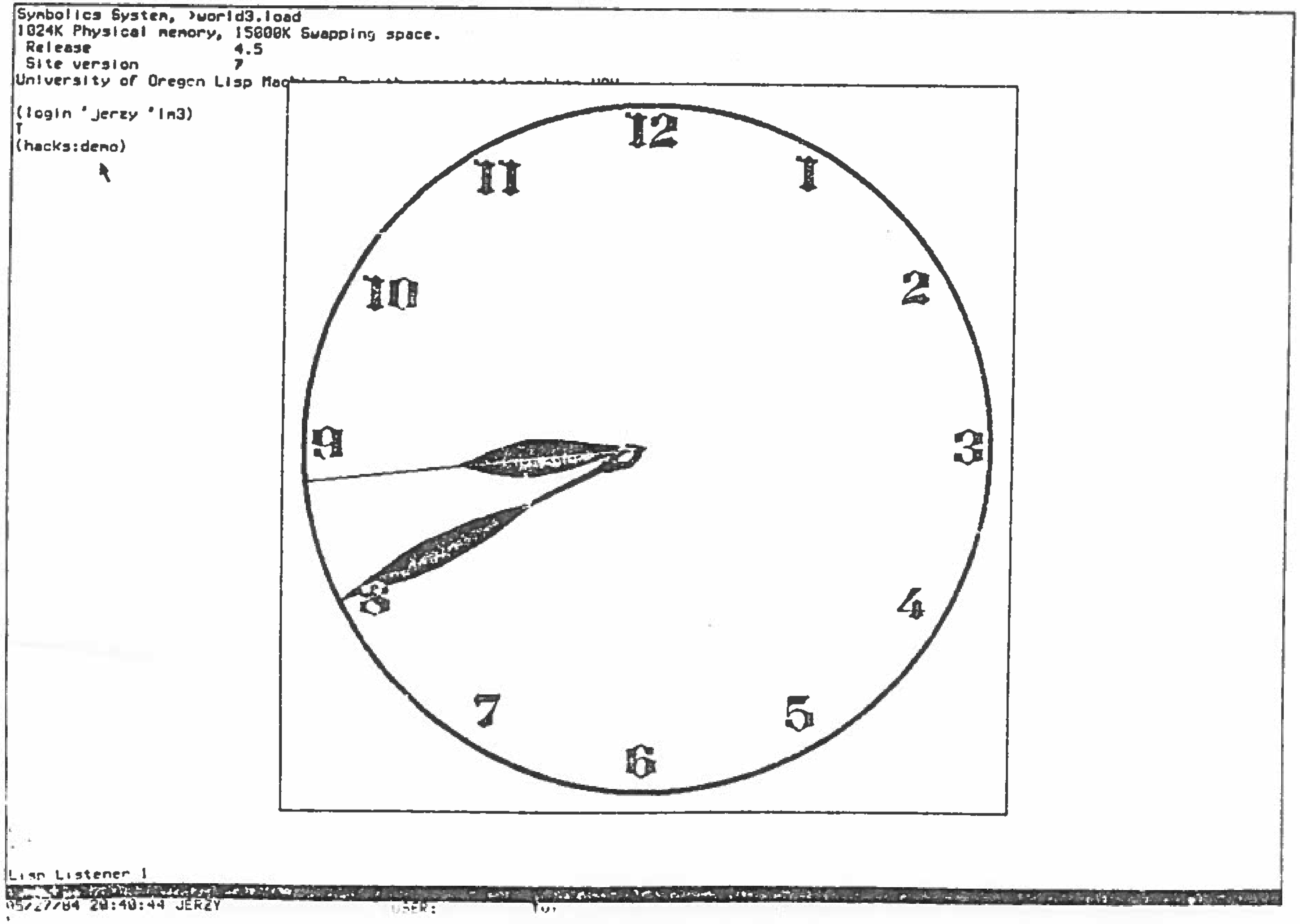
```
ZMACS (LISP) hard.lisp >kent LM1: (7) *
Quote is not a defined key.
```

```
[illegible]
06/25/84 11:10:19 JERZY                    TV:            Ty:
```

An Example Screen Image

(generated by Symbolics provided HACKS:DEMO program)

printed on ILP with the use of the modified hardcopy system.



```
Symbolics System, >world3.load
1024K Physical memory, 15000K Swapping space.
 Release                4.5
 Site version           7
University of Oregon Lisp Mac

(login 'jerzy 'lm3)
T
(hacks:demo)
```

```
Lisp Listener 1
05/27/84 20:40:44 JERZY                USER:
```

Appendix E

An Example Instruction to the Users How to Use the Modified Hardcopy System

We saved (on LM3) a new version of the system which includes all hardcopy functions. Now if you boot the LM3 (for some time it will be only available on LM3) you can immediately use the hardcopy system:

FUNCTION 0 Q (hit: FUNCTION 0 and Q keys) copies main window with the who line.
FUNCTION 1 Q copies selected window (the most recent one).
FUNCTION n Q copies main window with the who line (where n is from the $3 \leq n \leq 9$ interval).

The first time you use the hardcopy system function (one of the above) after you log in to the Symbolics, you will be prompted with the following message:

Process KBD ESC wants to type out

Now you need to grab (with the mouse) from the main menu, the SELECT option, which gives you the menu of all windows presently on the system. You need to select the ***Backgroud Lisp Interactor*** window which has the **KBD ESC** message. **KBD ESC** process asks for your VAX2 login name and password. If your VAX2 login name is equivalent to the one you have on Symbolics you need only to type in your password. After the Network Control Program has your VAX2 login name and password it establishes contact with VAX2 and sends the bitmap over the CHAOSNET to the /usr/spool/ipcd directory. From this directory it is taken by the spooler daemon (ipcd) and another programs are executed included in /usr/local/bin/ipc to convert it to the ILP format and next it is sent (using ipr spooler) to the ILP.