What does Symbolics really do? What do we sell? What do we have to do with *AI*? How do we continue to expand our market? All of these questions are of immediate relevance to the entire company. In the past we have been perceived as an *AI company* (whatever that means!), though I don't think that anyone can say exactly what AI we've done. Certainly we've contributed to most of the companies, laboratories, and universities working with or around AI. However, we ourselves have done, for all practical purposes, no AI. It's interesting (and quite irritating, at least to me) that we are touted as the "only profitable AI company" and as having sold the "highest dollar volume of AI software."

To the extent that we continue to promote this point of view, we do ourselves a tremendous disservice. We continue to give "AI seminars," we fail to correct misimpressions created by the media, and our sales force isn't trained in what we really do. So what is it that we really do? And why is promoting this AI image a disservice to us?

I think that we do *symbolic processing*. This term is almost as hard to define as *artificial intelligence*. Great. I have two working definitions of symbolic processing that I use. We need to make a concerted effort to produce a standard definition. Here are mine:

(A1) Computing with data structures that more closely model a person's thinking than those used in numeric processing;

(B1) Computing with data structures that are appropriate to the problem being solved rather than strictly numbers.

Since these two telegraphic statements come from a transparency, there's some prose that goes with them. Let's discuss these definitions.

One of the major objections to either of these definitions is that the term *data structure* doesn't mean any more than the term symbolic processing (and I assure you that that doesn't make for a useful definition!) When I use the term data structure, I mean, in a loose sense, a representation of information (data) inside a computer. I like to keep *data structure* in the basic definitions, since I think it makes them more concise. Also, the term carries some mental baggage along for those in the audience who've had exposure to the technical side of computers. Here's a rewritten definition:

(A2) Computing with representations of information inside a computer that more closely model a person's thinking than those used in numeric processing;

(B2) Computing with representations of information inside a computer that are appropriate to the problem being solved rather than being strictly numbers.

OK. But (A2) still discusses a person's thinking. Isn't this related to AI?

No.

Here's why: the person in quesion in (A2) is not an arbitrary person, but rather the **programmer,** or the person **specifying the problem.** And, when I say "numeric processing," I'm really refering to a conventional programming environment/language like Fortran, which deals strictly with numbers. So let's try that again:

(A3) Computing with representations of information inside a computer that are closer to the way a programmer, or the person specifying the problem, thinks about the problem than those representations used in Fortran-style "computing with numbers".

That's pretty close to (B2). Good! Those definitions should mean the same thing. Is (A3) understandable? Let me know.

I think this definition proves my point. If symbolic processing is really at the heart of what we do, then it doesn't really have much to do with AI. In fact, it would be like saying that "IBM sells the largest volume of spreadsheet programs for microcomputers." I suspect that Lotus Development Corporation would not deny that they use an IBM assembler to make their product work, but would certainly deny that IBM itself is selling the spreadsheet program! We are in the Same Boat. Vendors may sell more ART's and KEE's for 3600's than for any other machine, but I suspect that Inference Corporation and Intellicorp would violently deny that Symbolics gets the revenue!

But maybe even this analogy is to esoteric. Try another. Answer these two questions: A) how many houses do Stanley and Sears sell? B) how many houses are built with the **tools** that Stanley and Sears sell? As far as I know, these companies don't sell houses. But I'll bet that almost every house has had one of these company's tools used in its construction.

Symbolics sells a **tool.**

Generally, our tool is symbolic processing, which is a better way to implement computer programs.

Specifically, our tool allows someone (a programmer or end-user) to take advantage of symbolic processing in an efficient and effective way that's better than other ways of taking advantage of symbolic processing.

The tool of symbolic processing is generally excellent for writing programs that deal with non-numeric aspects of the real world. Let's discuss this. Conventional programming languages force the programmer to express the problem in mostly numeric terms. Where problems are mostly numeric, this is very appropriate, and symbolic processing **per se** doesn't help much (but the environment does, this will be discussed later.) Fortran works well for many scientific calculations (so does symbolic processing.) But most problems are not numeric, and the programmer is forced to go through extra levels of conversion between the actual problem and the realization of that problem in the computer. In other words, the computer is working with information that's not appropriate to the problem being solved (see the definition.)

But wait a minute! Aren't all computers just on and off switches, when you get right down to it? And there's a well known way of putting ones and zeroes (ons and offs) together to make numbers (binary). So how can you get away from numbers anyway? That's right, our machines are *turing machines* and so, in a very fundamental sense, can't do anything more than any other computer with a similar amount of memory. Turing proved this years ago, and this remains a very fundamental and important result in computer science: that all machines of a certain class can be made to do the same things. What symbolic processing does is make the computer do more of the translation work. That is, the computer participates actively in converting non-numeric data structures into the ons and offs that the computer ultimately manipulates. So, the programmer is able to communicate with the computer in a more effective manner. Which ties us back into the definitions.

Let's carry the tool analogy a little bit further. There's an interesting thing that happens when you improve your tools enough: sometimes things that were too hard to do **practically** (but possible in theory, of course) become **practical** with the improved tools. The early Egyptians were able to build very nice pyramids with mostly their bare hands and simple equipment, but they didn't build all that many of them and the idea didn't spread too far. Today, we build skyscrapers with ease and prolificacy all over the world. Physics didn't change, it didn't suddenly become **possible** to build skyscrapers (it always was, in the abstract), it just became **practical**. And that's why it doesn't really matter that a symbolic

processor is still a turing machine, because it moves certain things from the realm of the *abstractly possible* to the realm of *imminently practical.*

OK. Maybe you're willing to admit that something like *symbolic processing,* in the abstract, might be a good idea. After all, if it not only makes a programmer more productive, but also makes certain new techniques, like modelling machine intelligence (i.e. AI), practical, then it's a very interesting idea. I think it's important to discuss some of the key ideas to understand how symbolic processing gives us these benefits.

One of the key features of symbolic processing is the availability of several basic kinds of data structures, both numeric and non-numeric. On the numeric side, many kinds of numebers are available, including integers and "bignums," floating points, rationals, and complexes. Bignums are integers that can represent any size number, beyond the range of *standard* integers. A rational number is an integer divided by an integer (a fraction, essentially), and is used to represent exactly non-integer quantities. One-third (1/3), is an example of a rational number. What numeric processing language offers these kinds of built-in data structures?

The simplest non-numeric datum in symbolic processing is one which indicates two other data. These are historically called *conses.* The other data can be **anything,** including numbers or other conses. Out of conses you can build structures of arbitrary complexity. One of the most useful ones is called a *list,* which is simply a structure of arbitrary size that forms a collection of data. A list is readily built out of conses in such a way that it can be extended or shrunk, its elements changed one by one, and so on.

Arrays, either single- or multi-dimensional, are an important data type. In symbolic processing, each element of an array can be any datum, including numbers, lists (or conses), other arrays, and so on. Arrays are useful for modeling things of arbitrary length that need to be indexed (e.g. text strings are implemented as arrays, as one might need the fourth (or nth) character of a text string). A chess board is naturally implemented as a two-dimensional array, with non-numeric elements (i.e. chess pieces).

Other non-numeric types include generically *typed objects,* which are arbitrarily sized objects that can be given explicit types by the programmer. Since the kind of object can be identified by a program, it is possible to make that program behave differently for different types of objects. One possible use of this

feature is for display purposes: an object of a certain type can be displayed in a manner appropriate for that type (e.g. when an object has a name, that name can be displayed.)

Another important non-numeric data type is a *Flavor instance*, which is an object that contains not only data, but also procedures (methods) to operate on that data (see *object-oriented programming*, below). This is commonly called *object-oriented programming*. Even though object-oriented techniques can be built out of more basic symbolic processing concepts, they are so useful that they must be considered as part of any reasonable symbolic processing system.

Implied by the presence of these more "interesting" data structures are *control structures* that operate on them. If we consider data to be the *nouns* of our language, then the control structures are the *verbs*. For example, there's a way to do looping that says: do something for every element of a list. And there's a similar construct for every element of an array. Control structures (verbs) can be **added to the language by a programmer**, too. I'll discuss this in more depth later.

A major feature of symbolic processing that makes non-numeric data types exceptionally useful is *dynamic storage management*. This technique allows a programmer to create data structures as needed, without worrying about pre-allocation. In a typical numeric language, you must say "I am going to use an array 100 words long now," and then when you're done, you must say "I am now done with this array." It turns out that in many major Fortran programs, a significant amount of programming deals with handling the fixed amount of memory available. In symbolic processing, many data are of variable size. Symbolic processing takes care of allocating memory as needed, and freeing it when it is no longer in use (this process is called *garbage collection*). So, no arbitrary restrictions are imposed on any program. If a program is making a list) of things, the list is not restricted to only sixteen (for example) elements, because the program can request storage to hold more elements of the list as required.

David Loeffler of Honeywell related a programming example to me that he likes to use to contrast numeric processing (Fortran, let's say) and symbolic processing. Let's start with a fairly simple program: given a dictionary and a word, find if the word's in the dictionary. In a typical numeric language, like Fortran, this involves writing some amount of code (at a least a loop doing string comparisons). In a symbolic processing language like Lisp, there is a primitive that checks each element of a list against

another thing and tells you if that thing is in the list. Essentially, keeping around collections of things (like words) is simple.

But this example is still somewhat trite. Let's make the problem more difficult. Instead of having a fixed dictionary, if a word isn't found, add it to the dictionary. Do this for an arbitrary number of words. Now I've given you a problem that is fairly difficult for a numeric language. In symbolic processing, however, this is a natural problem to solve. You just use a list (for example), and add to it if the word isn't found. This kind of task comes up all the time in real programs. After all, in the outside world, most problems involve an arbitrary number of elements. It's rare to find a problem where you know in advance how many things you're dealing with.

I like to illustrate the symbolic processing "way of thinking" by discussing the idea of identity. It's an interesting concept that is **completely lacking** in numeric languages. In conventional languages, there is always some sort of equality comparison operator (usually =). I'll call this *equal.* When you ask whether two things are equal, the language looks at both of them and tells you whether or not they *look the same.* Of course, symbolic processing has ways of checking this same thing.

It turns out, however, that when dealing with non-symbolic data the *looks the same* check isn't the right one most of the time. For example, let's say I describe two people to you. Person A is the local doctor, and person B is John Doe. Now, I ask you a question: if I send a letter to person A, does person B get the letter? You could ask me if person A and person B look the same, but that doesn't give you the information you need to answer the question. After all, they could be identical twins. A better question to ask would be whether person A and person B *are the same person.* This is exactly the sort of problem that arises all the time when writing symbolic programs. This is the concept of *identity*, and it's a fundamental concept that doesn't even come up in numeric processing.

Another key concept of symbolic processing is the notion of runtime type checking and *runtime generic operations*, which are made possible by the fact that the symbolic processor knows what kind of data it's manipulating. The simplest example is arithmetic: mathematical operations work correctly for any types of numbers. So, if you write a program that adds two numbers, that program can handle any kind of numbers -- integers, complexs, and so on -- you choose to give it. On special purpose symbolic

processors (especially the 3600 family, which has special hardware) no penalty is paid for this runtime checking.

A more subtle (and I think more useful) example of generic arithmetic is that the system can **change how it represents numbers** without affecting the way a program works. As far as a typical program is concerned, there is no difference between a normal size integer and a *bignum*. So, if you add two normal integers together and that overflows the word size, the system switches to an alternate representation (i.e. a bignum) and gives a **100% correct result**. Normal numeric processors can't do this, and you get either an overflow trap (in the worst of all possible worlds, the system doesn't even tell you this happened!)

The concept of generic operations can be extended to user-defined data structures. Since type information is always maintained, a program can always find out what kind of datum it has, and act accordingly. *Object-oriented programming*, which I think is an integral part of symbolic processing, enhances these concepts even further (I'll discuss this more later on.)

A collateral benefit of runtime datatype checking (and a very important one) is the ability for a symbolic processing system to detect errors at runtime that are not normally caught. For example, if you try to use a variable that hasn't been initialized, the system tells you. In most languages, an arbitrary value, or sometimes a specified value (zero, perhaps) is used. Another example is array bounds checking, which on a symbolic processor is done at runtime. So instead of having your program write all over its memory, or read invalid data, you get an error break from which you can debug your code. The advantage to a programmer is clear. The advantage to a user is that instead of producing random and unpredictable results, the system essentially tells the user: "Hey, there's a bug here, you'd better get it fixed if you want your computation to work correctly."

I believe that *object-oriented programming* is an integral part of a modern symbolic processing system. There's some confusion about what the term actually means, however, and I think it's important to explain. When I say *object-oriented programming*, I'm refering to the technique pioneered by Simula-67 (and later, Smalltalk) that allows breaking a problem into two pieces: 1) a set of programs that perform generic operations on *objects* that meet certain specifications (often called *protocols*), and 2) a set of

objects that encapsulate (i.e. combine) both data (which is sometimes called *local state*, which simply means data belonging to a particular object) and programs (which are sometimes called *methods* or *behavior*).

How is this particular split used? Let's take look at an example. We want to solve the problem of placing furniture in a conference room. I can tell you how to do this without refering to the details of any of the particular pieces in the room. I can tell you that I want at least five chairs around the conference table, and that they can't be any closer than a foot apart. I can tell you I want some of the chairs to have arms. And so on. Then I can give you a collection of different types of chairs, and you can place them around the table. Some chairs might have wheels, and those you can roll. Some may be very heavy, not have wheels, and need to be carried. And so on. The *details* of the chairs **do not matter**. The only thing that matters is that each chair meets some general set of specifications. Each chair needs to have some position and size, needs to hold a single person, etc. So, you can describe the *algorithm* for placing chairs in a generic manner, and it will work for any particular kind of chair. In very much the same way that the real world works. I think that **object-oriented programming is ideal for modeling things in the real world.**

The Symbolics 3600 environment takes great advantage of object-oriented techniques. The *Flavor system* is a design and implementation of object-oriented programming that allows for certain things not available in previous systems. I'm partial to it, as I invented it in order to implement the window management package (*window system*) for the CADR Lisp Machine at MIT. Use of objects at the lowest levels in the system yields some important advantages for the programmer and user. Not the least of which is increased generality of certain system functions. **Ninety percent (or more** of the code that I write these days is object-oriented).

Generic operations and object-oriented programming (amongst other features) tend towards isolating the programmer (and user) from the details of the hardware. This means that programs continue to run even as the underlying hardware base changes significantly. For example, on the 3600 you are able to run graphics programs on either the black and white or color screen without altering the code. This is because the window system is built in a general manner even at its lowest levels. This is directly due to the availability of efficient object-oriented programming. The 3600's symbolic processing environment

(including what people call the *operating system*) was converted from the previous generation machine (the LM-2) in about two man-years [check this number!]. This included changing the word size of the machine from 32 to 36 bits, and completely redesigning the instruction set of the computer. I think this is amazing.

To end the discussion of key symbolic processing ideas, I'd like to bring up one more point. That is, in a symbolic language programs can be data, and data can be programs. The former means that a program, either compiled or not, is simply another kind of data structure in the environment. It can be stored in arrays or lists, passed around between subroutines, etc. The latter means that a program can construct another program on the fly (as data), and then execute it. Both of these features are amazingly powerful and almost never found outside of symbolic processing. And since the compiler is just another symbolic program sitting around in the environment, it is possible for a program to write another program, compile it, and **execute the compiled code.**

```
[mention dynamic linking]

And more to come:

Symbolic Processing Style

   Programming is a language building activity where
     programs implement a "jargon" and
     basic control structures and data-types are
     augmented with new "verbs"  and "nouns."

   Modularity is facilitated.

   Rapid prototyping, progressive refinement, and
     testing of partial solutions are supported
     by a rich and sophisticated programming
     environment.


Object-Oriented Programming with Flavors
(Message Passing)

   - Integrates data and program.

   - Isolates generic algorithms from
        specific implementation details.

   - Supports the "programming style" by aiding
       jargon building, modularity, and rapid prototyping.


Symbolic Processing Graphics Examples

 Windows, Paint Brushes, Rendering,
```

VLSI Design, 3-Dimensional Editing

- Dynamic storage management and a large virtual address space eliminate arbitrary restrictions.
- Flavors permits different implementations (types) of things (e.g. windows) to interact gracefully.
- Rich choice of data structures allows algorithms to be expressed naturally and succinctly.
- Some measure of hardware independence is achieved through the use of object-oriented programming at low levels of the system.
- Effective user interfaces are readily implemented.

Specific 3600 Family Features

- Very efficient function calling and object-oriented programming.
- Large virtual address space.
- Multi-tasking in single address space.