The research reported in this paper was sponsored by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command under contract F1962867C0004, Information Processing Techniques, with the System Development Corporation.

TECH MEMO

a working paper

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406

Information International Inc. / 11161 Pico Boulevard / Los Angeles, California 90064

TM- 3417/700/00

AUTHOR Jeff Barnett for L.H.

L. Hawkinson

TECHNICAL Jeff Barnett

J. Barnett

RELEASE [illegible]

C. Weissman, S.D.C.

[illegible]

D. Anschultz, I.I.I.

for J. I. Schwartz

DATE 4/26/67 PAGE 1 OF 4 PAGES

(Page 2 Blank)

LISP 2 Core Image Generator (CIG) Specifications

ABSTRACT

This document specifies functions to be performed by the Core Image Generator (CIG) within the LISP 2 system proposed for the IBM S/360 computer. It defines the components of the CIG, how they are to be written, and alternative approaches to the core image generation process.

1. INTRODUCTION

Core image generation is a bootstrapping process by which an initial implementation for some target machine is acquired. It is itself a LISP program written on some existing system called the bootstrapping system.

The core image generation consists of the computation of the contents of the target machine's core storage such that it can be loaded and started operating as a viable LISP system. This computation is effectively the product of compiling the code for the system (as written in LISP 2). There is, however, one major difference between a LISP CIG and, say, a JOVIAL compilation: that is the complicated processes of storage allocation that must take place. Allocation processes for such things as identifiers, free variables, arrays, strings, list nodes, etc., must be simulated in the CIG for the target machine.

2. CIG COMPONENTS

In order to define the components of the CIG and how they are to be written, let us imagine that we have complete code for a LISP 2 system on some target machine, and let us examine what we might do to get that code transformed into binary which will actually run on the target machine. The components include:

1. Syntax Translator
2. Compiler
3. LAP (generating external code)
4. Storage Management Simulator

Each of these components must be prepared to run on the bootstrapping machine.

Clearly the CIG code for the first three components is semantically the same as the corresponding system components that are to be put through the CIG. Let us consider each in turn, assuming for the moment that the CIG components are syntactically as well as semantically equivalent to those in the system, i.e., that bootstrapping is performed within a compatible LISP 2 system.

2.1 SYNTAX TRANSLATOR

The Syntax Translator for LISP 2 is machine-independent in respect to both its coding and the transformation it performs. Therefore, the Syntax Translator available in a bootstrap system will suffice without change for the CIG and for the target system.

2.2 COMPILER

The Compiler consists of several passes, some of which are machine-independent in both respects, some only in their coding, and some not at all. However, the Compiler as written for the target system may be used as is for the CIG.

2.3 LAP

The LISP 2 Assembly Program (LAP) is written for the target system to plant machine instructions into core storage. This obviously will not work during core image generation since the CIG merely wants to determine the target machine address and contents for each instruction and then write them on to some output device. Thus the LAP component of the CIG differs from the corresponding components of the target system in its disposal of assembled code. In order to minimize the amount of rewriting required to adapt the target system LAP to the CIG, it is coded in two parts. The first--and by far the larger part--takes assembly language into a list of address-content-mapping triples. The second--and smaller part--must create in the target system a binary program image (BPI) and fill it according to the address-content-mapping list; in the CIG, on the other hand, it must output the list in a format suitable for loading into the target machine.

2.4 STORAGE MANAGEMENT SIMULATOR

For each function entered into the CIG, the Storage Management Simulator must be able to simulate the storage allocation processes that would take place in the target system if that function had been entered into the target system itself.

The product of the simulation is a set of address-content pairs for all structures allocated. These allocated structures fall into two groups: unique and non-unique. Non-unique structures, such as strings and unhashed list nodes, can have their address-content pairs written out as they are allocated. On the other hand, unique structures such as identifiers, hashed lists, and function descriptors, must be kept track of in tables during core image generation, and can only be written after all target system code has been run into the CIG. These tables for unique structures essentially form a model of the structural scheme through which the target system allocators manage to preserve the property of uniqueness.

3. ALTERNATIVE APPROACHES

From the preceding consideration of the CIG components, it is apparent that if one has completed the target system coding, and if one has a compatible LISP 2 available for bootstrapping, then the amount of work required to code the CIG is quite small. Indeed, it consists of writing the BPI output part of the LAP component and coding the Storage Management Simulator. The first of these tasks is straightforward, requiring perhaps a page of code. The second requires a large volume of code, but the logical structure can be borrowed from the primitives as written for the target system.

Core image generation is thus a very effective technique, given the availability of a compatible LISP 2. If a compatible LISP 2 system is not available, the following alternative approaches could be considered.

First, let us imagine that we are to use a LISP 1.5 (or perhaps a primitive LISP 2) as the bootstrapping vehicle. It might be emphasized at this point that not just any LISP system will do, since several thousand words of list structure--say 15,000--are needed for the Storage Management Simulator tables. If we again consider the problem of producing the CIG components (but this time for a LISP 1.5 bootstrapper) it is clear that the Syntax Translator, the Compiler, and the assembly part of LAP will not suffice in their target system form. They must, in fact, be translated from the more complicated LISP 2 language to the simpler LISP 1.5 language. Complications can arise in this process of translation; for example, any function using locatives in the target system components cannot be easily translated. What tends to happen when the bootstrapping language is simpler than the target system language (as was the case in the development of Q-32 LISP 2) is that the system components that must operate in the CIG are written first in the simpler language, to be later translated to the more complicated one. Although this simplifies the procedure for obtaining an initial target system, it results in the production of a somewhat inferior system.

The alternative to core image generation is to proceed from LISP 1.5 to a LISP 2 system by evolutionary stages. This approach is certainly more costly than bootstrapping a LISP 2 from some existent compatible LISP 2. Whether the same can be said of bootstrapping from LISP 1.5, however, is a question which is not easily answered because of the human factors involved. When the bootstrapping path is chosen, there are no concrete demonstrable products between the design of the system and the obtaining of a more or less complete working system. If one travels the evolutionary path, one always has some sort of working system in hand.