This work was presented to the faculty of the Graduate School of Yale University in candidacy for the degree of Doctor of Philosophy. The author is presently with the Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304.

AUTOMATIC GENERATION OF ASSEMBLERS

John Dryer Wick

Research Report #5Ø

December 1975

This research was partially supported by the Alfred P. Sloan Foundation and by the Department of the Army under contract DAABØ3-74-C-ØØ55.

ABSTRACT

AUTOMATIC GENERATION OF ASSEMBLERS

John Dryer Wick

Yale University 1975

Every new computer requires a basic set of software routines; device drivers, interrupt handlers, program loaders, and symbolic assemblers are examples. Since the function of each of these standard components is well understood, they are a primary target for automated design and implementation. This thesis describes an automatic programming system that generates symbolic assemblers.

A programmer produces an assembler using some preconceived model of the process, which he adapts to a particular computer by a close inspection of the machine reference manual. The automatic generator proceeds from an axiomatic definition of the assembly process and a machine-independent implementation model. In place of the reference manual, the generator uses a formal definition of the computer written in a machine-description language called ISP (Instruction Set Processor [Bell and Newell. Computer Structures: Readings and Examples. McGraw-Hill, New York, 1971]). Based on an analysis of this description, the generator invents a syntax for the assembly language and programs the semantic routines necessary to translate it to machine code.

This system demonstrates the feasibility of using a formal description of a computer as one of the inputs to an automatic programming system. Using this approach, the assembler generator is capable of handling a wide variety of computers with substantially different architectures. The application of this technique to more complex systems programming tasks merits further research.

ACKNOWLEDGMENTS

I am grateful to my advisor, Alan Perlis, who suggested this problem, asked all the crucial questions at the right times ("Now, can you get all this stuff from ISP?"), and who insisted on a living, breathing implementation of almost everything. Other members of my committee contributed much more than a careful reading of the thesis. Ned Irons saw to my proper upbringing in the general area of programming languages and systems software. Larry Snyder provided a working knowledge of ISP, as well as a watchful eye and a fresh outlook on everything I thought was obvious.

Members of the automatic programming seminar -- especially Jim Meehan, Bob Tuttle, Mike Condry, Phil Summers, and Jerry Cooper -- suffered through early presentations of parts of this work, and served as guinea pigs for various experiments with ISP. Dan Hoey and Charles Minter actually used some of the software produced for and by the system.

Finally, Jim Meehan freely provided the kind of support that only a true friend can give.

This research was partially supported by the Alfred P. Sloan Foundation and by the Department of the Army under contract DAAB03-74-C-0055.

TABLE OF CONTENTS

Appendix

1. OVERVIEW

"It would be immoral for programmers
to automate everybody but themselves."

M. Douglas McIlroy [NATO69]

The advent of microprocessors and minicomputers has led to a dramatic increase in the number and variety of machines a programmer must deal with at any given time. As each new machine is introduced, a complete set of basic software routines must be written for it: device drivers, interrupt handlers, program loaders, symbolic assemblers, FORTRAN compilers, and LISP interpreters, to mention a few. Both the function and construction of each of these programs is well understood. Given the specifications of a new machine, and perhaps an existing version of a similar program for some other computer, experienced programmers routinely produce each of these software components; in fact, they are often assigned as projects in undergraduate courses. Hence the construction of these programs, from initial design through coding and testing, is a prime target for automation. In this thesis, we consider one of these components -- an assembler -- and develop an automated systems programmer: a system that generates assemblers.

1.1 The Problem

Every new machine should be delivered with a complete set of basic software routines. By basic software we mean the set of programs that are necessary for the preparation and execution of "user" programs; this set is usually called systems software. The definition of "necessary" obviously expands as the development of programming tools becomes more sophisticated. Even as recently as the 1960´s, the only software provided with some machines consisted of a symbolic assembler, perhaps a few input/output subroutines, and a bootstrap loader. Currently, all but the most special purpose computers are delivered complete with assemblers, linking loaders, interpreters, at least one compiler, and some form of operating system.

Obviously, this basic set of programs will continue to expand. The growth of software systems, coupled with a proliferation of machines, combine to produce a critical software development problem.

Unfortunately, existing techniques which address this problem are based on achieving a greater degree of machine independence in software construction by using higher level languages [SIGPLAN74] or some other portable software methodology [Brown74]. While these techniques can be sucessfully applied to a large class of programs, they are of limited use for problems that are inherently machine-dependent, such as the systems programs mentioned

above.

An example will make this point clear. Suppose we have an assembler currently running on a host machine H, written in a high level language L, and we wish to produce an assembler for some new machine T. Using any of the popular half-bootstrap or full-bootstrap techniques [McKeeman70, Richards71, Richards74], we produce an L compiler that generates code for T, and use it to compile the assembler. We now have an assembler (and possibly also a compiler) running on T, but this assembler is still producing code for H!

In fact, we may not even be that lucky. If the two machines use substantially different representation schemes (suppose H is a binary computer and T is a decimal machine), the output of the assembler will be code for H, but in T´s representation (decimal). Once the source code has been assembled into bit-strings, it is impossible to distinguish instructions from data, so the conversion back to binary may not even be possible.

In any case, this program is clearly not the one intended as a final solution; the assembler should generate code for T. We have omitted the crucial step of every bootstrapping process: first, modify the code generator to produce code for the new machine [Sklansky68]. This is a difficult task, which must be performed for each of the systems programs mentioned above, before any thought can be given to portability.

In light of this discussion, it is clear that portability is only a part of the problem. Before a program can be transported, we must first redefine the task that it performs in terms of the new machine. The difficulty of this redefinition process depends on the degree of machine-dependence inherent in the problem. In systems software, machine-dependence is rampant, so the portability problem is overshadowed by fundamental changes in the processes that the programs perform.

This means that in order to generate systems software we must deal not with the programs themselves, but with machine-independent models of the tasks they perform. The generator must be capable of redefining and adapting pieces of the model, and rebuilding corresponding parts of the program (from the ground up, if necessary), based on the requirements of the target machine.

In order to pose this problem in concrete terms and illustrate the generation process, we will focus our attention on a single program: the assembler. There are several reasons for making this choice. First, it is a reasonably well-defined yet non-trivial problem; assemblers contain all of the basic components found in any language translator. Second, the assembler is a self-contained

system and independent of other support software. Hence it will be useful in the practical sense, even if it is the only system program available for the machine. Finally, variations among assemblers are due almost entirely to machine-dependent properties. We are therefore forced to deal with a large amount of machine-dependent information in a consistent and systematic way.

1.2 The Plan

The first task is to establish a clear notion of the assembly process, as well as a methodology for building assemblers. This must be done in a manner independent of any particular computer, in order to determine the properties common to all assemblers. At the same time, the differences between assemblers must be attributed to specific differences between target machines. Finally, an implementation model must be developed containing the basic building blocks (modules) found in every assembler.

The second task is to formulate a method of describing the target computer to the generator. What exactly will be the input to the system? We might develop a special-purpose assembler-definition language, or work directly from some hardware description of the machine. Each technique must be evaluated in terms of our overall goal: a completely automated system capable of handling a wide variety of computers.

The next task is an analysis of the target machine to determine the particular properties of interest in the assembler implementation. Assuming that all of the relevant questions have been formulated during the study of assemblers, this phase serves to extract the answers from the input. Naturally the complexity of this process depends on the exact form of the machine description.

The final task is the construction of the assembler, with emphasis on tailoring its input language, its processing, and its output to the features of the target machine. The general models developed during assembler analysis must be replaced with actual procedures, data structures must be made explicit, and variables must be replaced by constants wherever possible.

Each of these tasks is discussed in more detail below, and is fully described in four subsequent chapters. Here we will state assumptions, indicate related work, and outline the approach.

1.2.1 Assembly

Even though assembly was one of the first processes performed by computers, there is considerable disagreement on the definition of the term; the discussion appearing in [Ferguson66] contains at least three different points of view. The simplest definition will be used here: assembly is the translation of a symbolic representation of

instructions and data into binary machine code. We will develop a "quick and dirty assembler" [Knuth74], which provides access to all of the machine's features, but is devoid of features often found in existing assemblers, but more properly included in higher level languages.

The motivation for this choice is to focus attention on the machine-dependent properties of the problem, in the hopes that the techniques for machine analysis developed here can be extended to other applications (code generators, for example). The development of sophisticated assemblers is not a primary goal of this research. We assume that the assemblers generated by this system are used only for initial software development on the target machine, and are replaced by more powerful languages as soon as possible. In this context, it is important for the assembler to generate absolute code that can be executed directly, rather than assume the existence of a linking loader or a runtime support system.

Assemblers have received relatively little attention in the literature; [Mealy63] is the classic paper on the subject. Most introductory textbooks contain a section on assemblers, and some even include a description of how to build them [Donovan72]. But these examples are necessarily limited to a single target machine. [Barron69] is the only text devoted exclusively to assemblers and loaders, and contains some discussion of the differences between existing assembly languages. [Ferguson66] addresses some of the machine-dependent aspects of assembly, and introduces the concept of a "meta-assembler". In Ferguson´s system, each instruction is implemented as a macro, and very general facilities are included for defining operands and referencing arbitrary fields in the output stream. Assembly-time global variables are used to represent machine-dependent values such as the wordsize. To produce a new assembler, one merely has to change the parameters and rewrite the macros for each instruction.

The meta-assembler simplifies the task of implementing an assembler by providing common software for syntax analysis, symbol table maintanence, and expression evaluation. But it only goes part way toward automating the task, at great expense in the efficiency of the translation [Olson, Walter]. It is also difficult to change underlying assumptions about the class of machines that can be handled. We will make use of several concepts arising from the work on meta-assemblers, but the generalized macro approach is too inefficient and inflexible for our needs.

Fortunately, the literature does contain numerous examples of assemblers and assembly languages; most of these are found in computer manufacturers´ reference manuals [IBM57(65Ø), IBM6Ø(14Ø1), Honeywell62(8ØØ), Univac(11Ø7), DEC67(PDP-8), IBM67(36Ø/37Ø), CDC67b(66ØØ), IBM68(18ØØ), DEC71(PDP-11), DEC73(PDP-1Ø), DGC74(Eclipse)]. Chapter 2

contains a distillation of the approaches used in these examples. An axiomatic definition of the assembly process is developed there, and the machine-dependent properties are identified and characterized with examples in section 2.3.

Finally, an implementation model is developed based on syntax-directed translation techniques. Since the lexical analysis and parsing phases are well understood, code generation receives most of the attention. In deriving the building blocks of the model, we use information-hiding principles [Parnas72a, Parnas72b], whereby modules are defined around design decisions that are likely to change when the assembler is tailored to a specific machine.

1.2.2 Machine Description

The analysis of existing assemblers must necessarily result in a model that leaves some parts unspecified, since it is not yet bound to any particular computer. We must determine exactly how the machine-specific knowledge is presented to the system. Rather than inventing some specialized assembler-definition language, we will use a complete specification of the computer, written in a machine-description language called ISP (Instruction Set Processor [Bell71]). There are several reasons for this approach.

The assembler we are developing serves to present to the user a convenient representation of the machine. This implies that the hardware itself determines the scope of the language, and it should serve as the base from which to proceed. By using a formal machine description, we are assured that all of the machine-dependent information that might effect the assembly process is available for consideration. We are forced to examine a large machine space, and to cast machine-dependent properties in machine-independent terms. While it is unreasonable to expect the system to anticipate all possible machine designs, the system will at least have the potential for handling computers with as yet unforeseen features.

Using a complete machine description also allows considerable flexibility in determining the hardware level of the assembler desired. The machine can be described in terms of either the physical or the virtual address space, for example. Microprogrammed processors can be described at the microinstruction level, or the description can reflect the machine implemented by the microcode. The microcode might implement some high-level language directly; if the description defines a LISP machine, for example, the assembler will contain "CAR x; CDR x; CONS x,y; EQ x,y", and so on. In general, if the user describes the machine at level X, an assembler useful at level X will result.

The idea of using a formal machine description as an input to a program generator is not new; [Feldman67] mentions this approach as an area for further research in translator-writing systems. No doubt one of the reasons for the lack of progress in this area has been the absence of a well-established, high-level hardware-description language. In chapter 3, several languages are examined and the choice of ISP is justified. In remaining chapters, we will assume that both the assembler generator and the user of the system have a "working knowledge" of ISP.

1.2.3 Machine Analysis

The major consequence of using ISP as the input specification is that the generator must contain a sub-system (called the analyzer) that "understands" computer descriptions. Fortunately, it need not understand everything about the machine; only those properties which effect the assembler design need be considered. Thus it is appropriate to ask the analyzer whether two instructions have the same format, or if the third field in some instruction is used as an op-code. But it is not expected to answer questions outside its domain: What is the cost-performance ratio of this computer? What is the most efficient FOR-loop instruction sequence? What is the floating point division algorithm? By "understanding" the machine description, we mean the set of questions that can be answered by the analyzer.

The section on assemblers serves in part to define the domain and characterize this set of questions. First, the analyzer must recast the primitive notions (op-code, operand, instruction, ...) in terms of precise ISP constructions. Algorithms must be developed to identify instances of these constructs and extract the relevant properties. Fortunately, ISP descriptions resemble well-structured programs in many respects; for example, the analyzer knows (rather, we assume) that every von Neuman machine has an instruction interpreter, and hence a program counter. Since the interpreter can be viewed as a program, some of the techniques used in program analysis [Ruth74] and program verification [King75] are applicable to machine descriptions as well.

The output of the analyzer is a rather complex data structure that describes the instruction and data formats of the machine; these are the items for which the generator must devise symbolic representations. In addition, a considerable amount of information about the machine is represented procedurally and accessed in a question-answering environment; for example, the wordsize of the machine is represented by a procedure which calculates it. It is important to realize that the set of questions that may be asked is fixed; the system does not learn about machines by analyzing them.

1.2.4 Program Synthesis

The analyzer gathers all the information necessary to instantiate the implementation model so that it becomes a program. Since the model is organized around information-hiding modules, it is convenient to organize the program generation in the same way. A generator is assigned to each module; it is an expert in its "field". Each generator contains a model of the program it must produce. The model may be represented simply by the text of the program (if it is machine-independent), or by an arbitrary set of procedures that eventually produce the text. The complexity of a generator depends on the degree of machine-dependence and on the amount of optimization that can be performed when model variables are replaced by machine constants. Some generators may merely determine compile-time parameters, others reorganize data structures and access macros, while others perform a complete rewrite of their programs.

There is no reason to restrict the output of a generator to be a program. One purpose of a module is to provide information but to hide details of structure and method. The generator may choose to present the information as data at program generation time, rather than as a runtime procedure. The character-set module, for example, provides all of its data at compile time in the form of values of literals. At the other extreme, a generator may produce specifications for some other automatic programming system. For example, the syntax generator produces a BNF description of the assembly language, which is later processed by a simple parser-generating system.

The critical issue during this phase is not so much the production of a running program, but the production of a good assembler -- good in the sense that 1) the assembly language is easy to read, write, and learn, 2) the translation from source to binary is efficient, and 3) the output is directly executable on the target machine, and flexible enough to be merged with the output of other systems software. Therefore considerable effort will be expended in designing the language, optimizing the code, and generalizing the output formats. Unfortunately, some of the information required for this is outside the analyzer´s domain, particularly with regard to the input and output formats. So the syntax generator and the load format generator are free to prompt the user for this information. In this sense, the assembler associated with a particular machine description is not unique; in every other respect, however, it is completely specified by the system.

1.3 Goals of the Research

The immediate goal is clear: given a description of the target machine, generate an assembler for it. To meet this goal, the terms "assembler" and "machine description" must be precisely defined, as well as the relation between

them. These definitions lead to a general implementation model in which the specific computer is an unbound variable. We must then verify the hypothesis that information about the target machine is sufficient to fill in all of the implementation details. The fact that assemblers have not evolved appreciably since their conception lends support to this view.

A less obvious hypothesis is that a program can be written to extract all the relevant information by analysis of a formal machine description. Producing such a program will be a difficult task. But this approach, if successful for a wide class of machines, raises the possibility of exporting the techniques and incorporating the analyzer into other software-generating systems. So the emphasis of the development presented here will be on machines, on the machine-dependent aspects of the problem, and on the analysis of formal machine descriptions. Since assemblers are relatively well-understood, I have limited the program space, in order to concentrate on expanding the machine space.

2. ASSEMBLERS

This chapter serves to precisely define the assembly process, and to formulate a general plan for attacking the implementation problem. The development is presented independent of the automatic programming aspects of our goal; the assembler is described without regard to who (or what) must actually construct it. Following a general overview of the assembly process, a formal definition of the machine-independent properties of assemblers is given. Then, a brief look at some "real-life" examples illustrates the nature of the machine-dependent aspects of assemblers. Given this background, we develop an implementation model based on classical syntax-directed translation techniques.

2.1 The Assembly Process

The purpose of an assembler is to translate symbolic assembly language into binary machine language. This involves three basic processes: assigning values to symbols, evaluating symbolic expressions, and converting these values to appropriate machine formats. A typical two-pass assembler defines all symbols during the first pass and formats machine code during the second pass, probably evaluating expressions during both passes. Assemblers are described in most of the standard programming texts, for

example [Donovan72].

2.1.1 Assigning Values to Symbols

Arbitrary symbols are given arbitrary values by the use of direct assignment statements; these symbols are usually called assembly-time parameters or simply parameters. In sophisticated assemblers, parameters may have types (integer, string, register, code) and a different pseudo-op is used for each type (EQU, SET, SYN, OPDEF). In simple assemblers, all values are treated as bit strings.

A restricted set of values may be assigned to the location counter (LC), using a special set of pseudo-ops or a restricted set of expressions. In addition, a special symbol (usually "." or "*") is normally reserved for referencing the current value of the location counter in symbolic expressions. Multiple location counters are sometimes provided.

Every assembler contains a rule for updating the location counter (simulating the program counter) as each instruction is processed. In simple machines (with one instruction per word), the LC may be updated by a constant; in other machines, the increment may depend on boundary alignment as well as operation and operand types. In any case, in the absence of pseudo-ops affecting the location counter, instructions and data are allocated in a "next-available, next-assigned" fashion, subject to the updating rule.

Finally, symbols can be assigned to locations so that they may be referenced symbolically, usually as operands of instructions. Hence statements may be labeled; the label is assigned the value of the location counter when the statement is processed. Unlike parameters, the value of a label is constant throughout the program.

2.1.2 Evaluating Symbolic Expressions

Expression evaluation is usually straightforward, since most useful forms are simple (involving only addition and subtraction). Two facets of the assembly process complicate the situation. First, label symbols may be referenced before they are defined; this requires a two-pass algorithm. Second, expressions involved in direct assignment statements may contain only previously defined symbols; otherwise an arbitary number of passes might be required to define all symbols.

2.1.3 Conversion to Machine Format

In this process, the assembler must reverse the instruction-decode cycle of the target machine, building up the binary representation of the instruction from the values of the symbolic expressions in the source code. These values are mapped onto various fields of a particular instruction format determined by the op-code. In simple machines only one or two instruction formats are possible,

and the mapping from source language syntax to bit fields is straightforward. In more complex machines, an entire tree of formats may be possible, depending not only on the op-code, but also on operand addressing modes. Note that this mapping process applies to data items as well as instructions; in this case the choice of format is normally indicated by a pseudo-op in place of the op-code (for example, "FLOAT" followed by a floating-point number, or "STRING" followed by a character string).

2.2 A Formal Description

This section serves to formalize the definition of an assembler (what it is) and of the assembly process (what it does), and thereby to precisely define the notions presented in the above section. Basic set theory is used to define the classes of symbols processed by the assembler: labels, parameters, op-codes, and operands. Based on axioms which specify the rules for defining symbols, we prove a fundamental theorm which every assembler must abide by.

2.2.1 Domain

In its most general form, input to an assembler is a stream of characters, and its output is a string of bits. In dealing with the semantics of assemblers, it is more convenient to characterize the input as sets of symbols.

2.2.1 Definition: The domain D of an assembler is the set of input symbols S union C union O, where S is an infinite set of identifier symbols, C is an infinite set of constant symbols, and O is a finite set of operator symbols.

2.2.2 Syntax

The formal definition of the input language is given in Backus-Naur Form (BNF); the first section defines symbols of the language (lexemes), the next section defines legal sentences of the language.

2.2.2.1 Definition of Symbols

The recognition and construction of symbols from the input alphabet is the job of the lexical analyzer; it can be described formally by a regular grammar. This grammar varies from one assembler to the next, depending on the character set available. Therefore a minimal (machine-independent) grammar is defined here.

```
<num> ::= dig | <num> dig
<nam> ::= let | <nam> let | <nam> dig
<opr> ::= ´:´ | ´=´ | ´,´ | ´(´ | ´)´ | uop | bop | eol
```

The symbols "uop" and "bop" stand for single characters which denote unary and binary operators (e.g. ´+´, ´-´, ...); "eol" stands for the end-of-line marker; "let" and "dig" represent letters and digits. The non-terminal symbols of the lexical analyzer define the sets of the domain D: <nam> defines the set S, <num> defines C, and <opr> defines O. By convention, characters of the input

alphabet which do not appear in any rule serve to delimit symbols but are otherwise ignored. It is further assumed that there is at least one such symbol (usually a blank).

2.2.2.2 Definition of Sentences

Naturally not all strings of symbols from D are legal input to an assembler; the following BNF grammar defines the meaningful sentences of the language.*

```
<prg> ::= <stm> eol | <prg> <stm> eol
<stm> ::= nam ´:´ <stm>
<stm> ::= nam ´=´ <exp>
<stm> ::= nam | nam <lst>
<lst> ::= <exp> | <lst> ´,´ <exp>
<exp> ::= <atm> | <exp> bop <atm>
<atm> ::= nam | num | uop <atm> | ´(´ <exp> ´)´
```

A program is a list of statements, which may be of two types: a parameter definition "nam=<exp>" or an instruction or data definition "nam" or "nam <exp>, <exp>, ...". In addition, any statement may be labeled, using the notation "nam: <stm>". Expressions are constructed in the usual way.

2.2.3 Basic Definitions

Since we want to deal with the the input at the semantic (rather than the syntactic) level, the output of

* Note that the non-terminal symbols of the lexical grammar are terminal symbols here.

the parser which accepts the language of 2.2.2.2 is now defined. Because the grammar allows two kinds of statements, the parser output will be in one of two mutually exclusive forms, corresponding to instruction statements and parameter definitions.

2.2.3.1 Definition: A program P is a finite ordered non-empty set of numbered tuples of the form:

2.2.3.1.1 k {l1,l2,...} m {e1,e2,...} or

2.2.3.1.2 k {l1,l2,...} a e

where k denotes the ordinality of the tuple in P. Further, the labels "li", the op-codes "m", and the parameters "a" are members of S, and each operand expression "ei" and parameter value "e" is a string composed from elements of S, C and O by the rules for expressions defined by grammar 2.2.2.2.

The notation |P| will be used to denote the number of tuples in P, and Pk is the kth tuple; in addition, k.li, k.m, k.ei, k.a and k.e refer to the various components of the kth tuple. Note that both "li" and "ei" (but not "e") may be empty.

The definition of P states that there are three types of symbols in S: labels, op-codes, and parameters. In anticipation of our theorm, we need to be more specific about the definitions and properties of these subsets, and the relationships between these sets and the symbols used in

expressions.

2.2.3.2 Definition: The set M is a pre-defined proper subset of S.

Axiom: For every k, if k.m exists, it is a member of M.

Op-codes and pseudo-ops (which begin tuples of form 2.2.3.1.2) are a special subset of S; they are reserved keywords of the language and may not be used as labels or parameters. Note that the set M contains all possible op-codes and pseudo-ops, whether they are referenced in the program or not.

2.2.3.3 Definition: Lj = {k.li|Ø<k<j+1, all i}. The set LØ is empty, and L = L|P|.

L is the set of all labels defined in P (syntactically, <nam>´s which begin a <stm> and are followed by a colon); Lj is the set of labels defined through tuple Pj.

2.2.3.4 Definition: Aj = {dot} union {k.a|Ø<k<j}. The set A1 is {dot}, and A = A(|P|+1).

A is the set of all parameters defined in P (syntactically, <nam>´s which begin a <stm> and are followed by an equal sign); Aj is the set of parameters defined before, and not including, tuple j. The element "dot" (standing for the symbol ´.´) is a special member of A used to reference the location counter.

2.2.3.5 Definition: Rj = {r in k.e|0<k<j+1}. The set R0 is empty, and R = R|P|.

Rj is the set of all elements of S which are referenced in expressions "e" of parameter definitions through Pj. R is the set of all such elements.

2.2.3.6 Definition: Uj = Rj union {u in k.ei|0<k<j+1}. The set U0 is empty, and U = U|P|.

Uj is the set of all elements of S which are referenced in all expressions "e" or "ei" through Pj. U is the set of all such elements.

2.2.4 Basic Properties

As is usually the case, not all syntactically correct inputs are semantically correct; this section defines some further restrictions on the domain D and the program P, by giving axioms on the sets defined in the above section.

2.2.4.1 Axiom: The sets M, A, and L (the subsets of S) are disjoint.

This property allows the assembler to get along with a single symbol table, since elements of M, A, and L cannot collide. Some assemblers use a separate symbol table for M, thereby allowing it to intersect with A and L, and resolve ambiguity by searching the tables in some pre-defined order.

2.2.4.2 Axiom: U is contained in (M union A union L).

Lemma: R is contained in (M union A union L).

Proof: by definition, R is contained in U.

This is the "Fundamental Axiom of Assembly"; it says that every symbol referenced in an expression "e" or "ei" must be defined as an element of M, A or L. The axioms which follow place further restrictions on how symbols are defined and referenced.

2.2.4.3 Axiom: For every k and i, k.li is not in Lk-1.

This axiom implies that a label may not be defined twice. Strictly speaking, a label symbol may appear twice as an "li" in the same tuple; this is allowed because it presents no semantic conflict and it simplifies the axiom.

2.2.4.4 Axiom: For every k, and for every r in Rk that is also in L, r is in Lk.

This says that every label used to define a parameter (appears in the expression "e" in form 2.2.3.1.2) must be previously defined.

2.2.4.5 Axiom: For every k, and for every u in Uk that is also in A, u is in Ak.

This says that every parameter must be defined before it is referenced in an expression (this applies to both tuple forms); in particular, it allows statements of the form "X=X+1" only if X has already been defined.

Axioms 2.2.4.4 and 2.2.4.5 taken together ensure that all symbols used in the definition of a parameter are previously defined. These axioms are somewhat stronger than those found in most assemblers; they are necessary to ensure that the value of the location counter symbol "dot" is always well defined, and that forward references can be resolved in at most two passes.

2.2.5 Semantics

We are now in a position to begin defining the semantics of the input (what the assembler does with it). The most complex semantic action of an assembler is to assign values to symbols. We therefore proceed by defining a function val(e,k) which specifies the value of expression "e" at any point "k" in the program. It is assumed that the "val" function on expressions is defined by the normal rules of arithmetic, and that a "rep" function is available for converting the external (character-string) representation of a constant to internal form. Therefore we need only define val(s,k) for all symbols "s" in S.

2.2.5.1 Axiom: For all s in M, there exists an element c in C such that val(s,k)=rep(c) for every k.

Lemma: For all k.m, there is an element c in C such that val(k.m,k)=rep(c).

Proof: directly from axiom 2.2.3.2.

Each element of M (instruction mnemonics and pseudo-ops) has

a pre-assigned constant value independent of k. Although pseudo-ops are not normally permitted in expressions, it simpifies the formalism to allow this and in the implementation to take their value as zero.

2.2.5.2.1 Definition: For every s in A except dot, val(s,1) is undefined.

2.2.5.2.2 Definition: For all k where k.a exists, val(k.a,k+1)=val(k.e,k).

2.2.5.2.3 Definition: For all k where k.a exists, and for all s in A except k.a, val(s,k+1)=val(s,k).

2.2.5.2.4 Definition: For all k where k.m exists, and for all s in A except dot, val(s,k+1)=val(s,k).

These definitions point out an important distinction between label symbols and parameters: the value of a parameter is a function of k, and hence may vary within a program, but labels (as we shall see) always have a single value. Definition 2.2.5.2.1 gives the initial value and 2.2.5.2.2 shows how to process parameter assignments; otherwise the value of a parameter remains the same from statement to statememt (2.2.5.2.3 and 2.2.5.2.4).

2.2.5.2.5 Definition: val(dot,1)=0.

2.2.5.2.6 Definition: For all k where k.m exists, val(dot,k+1) = val(dot,k) + f(dot,k.m,k.ei), where f is a machine-dependent length function.

This defines the location counter symbol, dot; it behaves exactly like a parameter, except that it is initially defined, and it is updated by the function "f" each time a tuple of form 2.2.3.1.1 is encountered. Intuitively, "f" is an instruction-length (and data-length) function; it is not defined here because it is extremely machine dependent. It may be a constant, a function only of "m", or a function of "m" and "ei".

2.2.5.3 Axiom: f(dot,k.m,k.ei) is always defined, even when any (or all) of the arguments k.ei are undefined.

This axiom is necessary to ensure that the location counter is always updated properly, even when operands of an instruction contain forward references. The section on machine dependence contains a more complete description of the length function "f".

2.2.5.4 Definition: For all k and every s in L, val(s,k)=val(dot,j), where s is in Lj and s is not in Lj-1.

This defines the value of a label to be the same as the value of the location counter the first time the label appears in any tuple (strictly speaking, "first time" has yet to be proved).

2.2.6 Theorem

We are now ready to state (and prove) the "Fundamental Theorem of Assembly", which says that if the sets M, L and A

are defined according to the axioms stated above, then all references to symbols (the elements of U), are well defined everywhere. First, we will need some lemmas.

2.2.6.1 Lemma: For all k, Lk-1 is contained in Lk.
Proof: follows directly from the definition of Lk (2.2.3.3).

2.2.6.2 Lemma: For any k and every s in Lk, there exists a unique j<k+1 such that s is in Lj and s is not in Lj-1.
Proof: immediate from lemma 2.2.6.1.

These lemmas guarantee that the "val" function on labels is well defined, provided that "dot" is defined everywhere. It is interesting that axiom 2.2.4.3, which states that labels may not appear twice, is not required; this is because the "val" function is defined in terms of the first occurence of the label, rather than the only occurence.

We now prove that the "val" function on parameters, and in particular on the location counter, is well behaved; this ensures that label values are defined properly.

2.2.6.3 Lemma: For every a in A, val(a,k) is defined for all k.
Proof (by induction on k): For k=1, by 2.2.3.4, A1=dot and by 2.2.5.2.5, val(dot,1)=0. Now assume (for 0<j<k+1) that a in A implies val(a,j) is defined. In particular,

val(a,k) is defined. By 2.2.5.2 and 2.2.5.3, val(a,k+1) is obviously defined whenever val(a,k) is, except possibly for a parameter assignment (case 2.2.5.2.2). In this case, val(k.e,k) must be defined. By 2.2.3.5, this follows only if val(r,k) is defined for all r in k.e. By 2.2.4.1 and 2.2.4.2, there are three cases:

(1) r is in M. By 2.2.5.1, there is an element c in C such that val(r,k)=rep(c) for all k.

(2) r is in A. Obviously from 2.2.3.6, Rk is contained in Uk for every k, and hence r is in Uk. But by 2.2.4.5, r is in Ak and by the inductive assumption, val(r,k) is defined.

(3) r is in L. By 2.2.4.4, r is in Lk. By 2.2.6.2, there is a j<k+1 such that r is in Lj and r is not in Lj-1. Therefore by 2.2.5.4, val(r,k)=val(dot,j). By definition 2.2.3.4, dot is in Aj for every j, and by the inductive assumption val(dot,j) is defined.

We now have that all the parameters in P (and in particular the location counter) are defined in each statement of P. It remains to be shown that all symbol references are defined.

2.2.6.4 Theorem: For every u in Uk, val(u,k) is defined for all k.

Proof: By 2.2.4.1 and 2.2.4.2 there are three cases:

(1) u is in M. By 2.2.5.1, there is an element c in C such that val(u,k)=rep(c) for all k.

(2) u is in A. By 2.2.4.5, u is in Ak and by 2.2.6.3, val(u,k) is defined for all k.

(3) u is in L. By 2.2.5.4 and 2.2.6.2, there is a j for which val(u,k)=val(dot,j), and 2.2.6.3 ensures that val(dot,j) is defined.

These lemmas and theorems explain why forward references in parameter definitions are not allowed (since the location counter might become undefined), and why forward references to labels are permissible.

2.2.7 Output

The above theorems precisely define the semantics of the first pass of a basic two-pass assembler, the symbol definition process. Characterizing the second pass, the output phase, is considerably more difficult, since the output function "g" is naturally machine-dependent. However, some basic axioms can be stated.

2.2.7.1 Axiom: For each k, if k.a exists, g(Pk) is empty.

For parameter definitions, there is no output. In terms of the output function, parameter definitions are side effects.

2.2.7.2 Axiom: For each k, if k.m exists and f(dot,k.m,k.ei) is not zero, then g(Pk) is not empty.

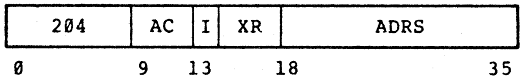
The output of a tuple of form 2.2.3.1.1 depends on all of its components. If the length function is zero (usually in the event of an error), there is no output.

The output function is intentionally left unspecified, in order to allow maximum flexibility in the types of machines that can be modeled. We will examine the input-output translation in more detail in the next section.

2.3 Machine Dependence

Traditionally, instructions and data are described by a picture of the item as it appears in memory, together with an explanation of each of the subitems named in the picture. The following is a typical example from the PDP-1Ø [DEC73].

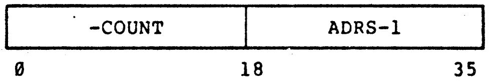
MOVS   MOVe Swapped



Calculate the effective address E from the indirect bit (bit 13), the index register (14-17), and the address (18-35). Then interchange the left and right halves of the word in location E and move it to the register selected by AC. The source is unaffected, and the original contents of the destination are lost.

Such a description is called a format (in this case, an instruction format), and each of the subitems is called a field. Usually several instructions share the same structure, but in the worst case, each op-code may have a distinct format. Data formats are described in similar terms.

IOWD   Input/Output WorD



The left half of the pointer contains the negative of the number of words in the block; the right half contains an address one less than that of the first word in the block.

A machine reference manual typically begins with a summary of the instruction and data formats, followed by a detailed description of each of the data-types and instructions.

The job of the assembler is to provide a symbolic representation for each of the formats of the machine. For instructions, the format can usually be uniquely identified by the op-code; remaining fields are specified by an ordered list of expressions separated by commas, for example,

```
MOVS  5,1,2,POINTER+1
```

A more complex syntax may be appropriate, if it makes the symbolic code easier to read. For this example, we might choose to represent the indirect bit by an at-sign '@', and enclose the index register (if any) in brackets. Also, each of the fields might be specified as a symbolic expression, rather than as a constant.

```
MOVS  AC5,@POINTER+1[R2]
```

A single field in the source code may represent more than one instruction field; for example, an address may map into a base register and a displacement. In addition, the output may depend on state variables of the assembler (a branch address might be relative to the current location, for example).

Data formats are represented in a similar manner; the data format name is followed by a list of expressions, one for each field of the format.

```
IOWD  LENGTH,BUFFER
```

In this case, the symbol 'IOWD' does not represent the value of a field in the format; it serves only to identify which data format is to be constructed. For this reason it is called a pseudo-op and has no value. Note that the assembler implicitly performs the computations on the fields, so that the programmer need not remember the details of the format.

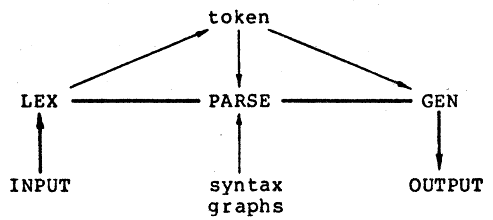
The idea of formats has long been used in machine descriptions and as the basis for constructing assemblers. A generalized format definition and processing facility is described in [Ferguson66]; in that system, each format is described by a procedure defined at assembly time (a macro with several entry points and exits). This approach proved to be too inefficient for use in production assemblers [Walter, Olson], and it is not easily generalized to machines that are not word-organized.

The solution to this problem is to note that, for any particular machine, the formats are constant and need not be defined at assembly time. Exactly how the formats are determined is the subject of following chapters. The next section describes the implementation of an assembler, assuming that all the formats are known.

Formats alone do not characterize all of the machine-dependent properties of the assembler, nor do they capture all of the peculiarities of the target machine. Other properties which effect the assembler implementation are the machine's data-types (representations or encodings) and its data carriers (registers and memories). For example, the format of an IOWD says nothing about the representation of negative numbers; complement, sign-magnitude, or some other scheme may be used. In addition, although the instruction format pictured above might suggest a 36-bit word, it may in fact be stored in memory as four 9-bit bytes. These problems are addressed in subsequent chapters.

2.4 Assembler Anatomy

The assembler implementation follows the classical co-routine model of syntax-directed language translators: a main driver (the parser) calls on a producer co-routine (the lexical analyzer) and a consumer co-routine (the code generator), communicating through a common data base containing the current input token.



2.4.1 Lexical Analysis

The lexical analyzer constructs tokens from the input character stream by simulating a deterministic finite-state machine. Using the techniques described in [Johnson68], a separate system [Wick75a] generates the program from a BNF description of a regular grammar (see appendix A). Except for the character codes used, this grammar is independent of the target machine.

2.4.2 Syntax and Parsing

The syntax of the language is an elaboration of the grammar in 2.2.2.2; unlike the lexical grammar, it is target machine dependent, since each machine has its own structure for op-codes and operands. However, these differences can be localized to the syntax of instruction and data statements alone, without disturbing the rest of the language. Typically a statement consists of an op-code (or pseudo-op) followed by a list of expressions representing the operands required by the op-code. But it

may be appropriate to represent some operands by a single character ("@" indicating the indirect bit, for example). Therefore a single parsing stratagy will process all assembly languages, provided minor changes to the syntax for operands are allowed.

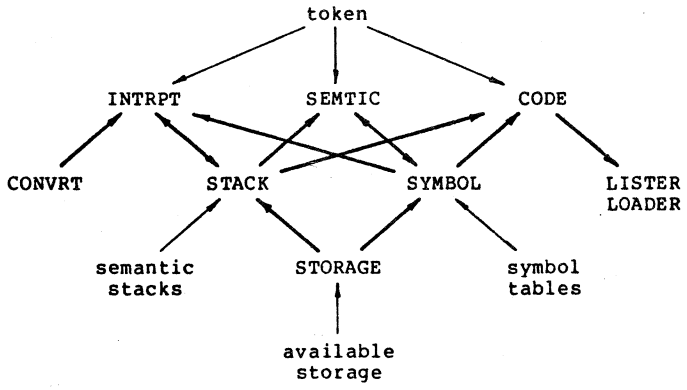
Numerous systems exist for generating parsers from formal language descriptions [Feldman67, Irons71, Altman74]; however, these systems are designed for high level languages with complex grammars, and they are too powerful for use in an assembler. The program used here [Wick75b] generates a deterministic top down parser without backup or lookahead [Conway63]. To achieve the necessary language variability, the grammar is represented as a syntax graph [Cohen70]. Links to semantic routines are embedded directly in the productions at the point where they are to be invoked.* Details of the syntax are outlined in appendix A.

2.4.3 Code Generation

The code generation phase can be further dissected into three modules: an assembly-time interpreter (INTRPT) for evaluating symbolic expressions, a semantic module (SEMTIC) for handling the definition of symbols, and a code module (CODE) for constructing instruction and data formats from

* In productions which follow, semantic routine names are enclosed in square brackets; for example, the syntax for parameter definitions is <stm> ::= nam [MNEMON] '=' <exp> [EQUATE].

their components.



INTRPT evaluates assembly-time expressions in Polish postfix order using an operand stack managed by the STACK module [Gries71]. Constants are translated to internal form by the CONVRT module, and values of names are supplied from the symbol table by the SYMBOL module. All other code generation routines obtain their operands from the interpreter stack; therefore an expression may appear anywhere a constant or symbol is allowed.

The SEMTIC module has jurisdiction over the definition of symbols (as well as a few bookkeeping steps); its functions follow directly from the axioms and theorems of section 2.2. It also contains routines for bootstrapping basic symbols, defining op-code mnemonics, and processing all pseudo-ops that do not produce binary output. The latter might include pseudo-ops that modify the location counter or determine the current base register and its value.

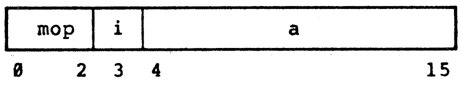
The CODE module assembles and outputs a format, obtaining field values from the symbol table, from the interpreter stack, and perhaps directly from the parser based on the presence or absence of an operator.* In contrast to other code generation modules, these routines are completely machine-dependent. However, they all follow the same basic pattern: 1) obtain operands from the stack, 2) perform some computation on the operands (and possibly also the location counter), and 3) assign results to fields or to the stack. When a format is completed, it is passed (with its location) to the output modules, and the location counter is updated.

2.4.4 An Assembler in a Nutshell

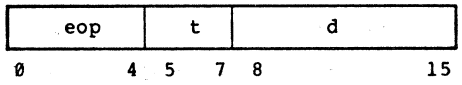
The following example illustrates the basic nature of the implementation model; it is a highly simplified version of the output which the assembler generator must eventually produce. It is included here only to show how the machine-independent model described above can be adapted to a specific computer. Assume a target machine with a 4K, 16-bit word memory and two instruction formats. The first

* For example, an indirect bit is often represented by an '@' preceding the address; it causes no symbol table access or expression evaluation, but it does cause a field to be assigned a value.

is composed of an op-code (mop), an indirect bit (i), and an address (a).



The second format contains an extended op-code (eop), a register selector (t), and a displacement field (d).



The first step is to define the syntax, including the points at which semantic routines are to be invoked.

```
<prg> ::= <stm> eol | <prg> <stm> eol
<stm> ::= lab [LABEL] <stm>
<stm> ::= mop [OP] <exp> [SETa] [PO]
<stm> ::= mop [OP] '@' [SETi] <exp> [SETa] [PO]
<stm> ::= eop [OP] <exp> [SETt] ',' <exp> [SETd] [PO]
<exp> ::= <atm> | <exp> '+' <atm> [ADD]
<atm> ::= nam [NAM] | '-' <atm> [NEG] | '(' <exp> ')'
```

Next, a brief description of each of the semantic routines is given. Recall that each subroutine has access to the current input token, as well as the interpreter stack and the symbol table.

Routine/Module	Action
LABEL/SEMTIC	Enters the token in the symbol table and defines its value and type.
OP/CODE	Obtains the value of the op-code field ("mop" or "eop") from the symbol table.
SETa/CODE	Sets the "a" field to the value popped off the top of the stack.
SETi/CODE	Sets the "i" field to one.
SETt/CODE	Sets the "t" field to the value popped off the top of the stack.
SETd/CODE	Sets the "d" field to the value popped off the top of the stack.
PO/CODE	Outputs the format and updates the location counter by the format length.
ADD/INTRPT	Pops two items off the stack, adds the first to the second, and pushes the result.
NEG/INTRPT	Negates the top of the stack.
NAM/INTRPT	Looks up the token in the symbol table, and pushes its value onto the stack.

From this example, it should be obvious, in principle at least, how to extend the assembler to process any set of formats. It remains to be seen how this set of formats is determined, and how sensitive the implementation model is to different machine organizations.

3. THE MACHINE-DESCRIPTION PROBLEM

As chapter 2 illustrates, it is reasonably straightforward to implement an assembler, once the language and the input/output translation have been defined. In particular, once the input representations (syntax), the output representations (formats), and the mapping (semantics) are known, the actual coding is not a difficult task. We must now investigate how these constructions are determined. Programming issues have a strong influence on the syntax of the assembly language, since that is the representation through which the user sees the machine. On the other hand, it is the machine which determines what must be represented, so we must have a way of describing it to the system. Two distinct approaches come to mind.

3.1 The Problem-Definition Language

One approach is to invent a specialized application language for defining assemblers; it would describe only those features of the machine which effect the assembly process. An example is the language appearing in [Miller71] that is used as a basis for implementing code generators. While such a language would represent a vast improvememt over existing methods for assembler generation, this approach has several drawbacks. First, it is difficult to introduce new concepts into a system based on a specialized application language. A great deal of foresight is required in its design, since the language must anticipate all possible machine organizations and their effect on the assembler design. Second, this approach does not address the automatic programming problem, since the user must still re-program the assembler for each new target machine. Our objective is to have this done automatically, with as little outside interaction as possible.

A second approach is to use a symbolic definition of the machine, written in some hardware-description language, as the input to the system. The generator must then be able to understand the machine description well enough to design its assembly language and implement its assembler. This is similar to a parser-generating system that understands a BNF language description well enough to choose the appropriate parsing method (operator precedence, LR(2), etc.) as well as generate the program. This approach has several advantages. First, a machine description has many other uses (simulation and documentation are among them), and it is likely to be developed in parallel with the machine design; therefore the assembler can be made available even before the machine is built. Second, the machine description must of necessity contain all of the information about the machine which might influence the assembler design. Finally, if the language is powerful enough, a wide class of existing machines (and their likely descendants) can be handled.

A desirable by-product of this approach is that it may lead to systems which can understand symbolic machine descriptions at a more complex level (for example, well enough to generate a FORTRAN compiler that produces code for the target machine); some of the possibilities are explored in [Barbacci74].

Unfortunately, there are some problems inherent in using a machine description as the problem-definition language. The information required to define the assembler may be difficult to identify and extract from a procedural definition; the goal of the machine description is to completely specify the machine's behavior, and therefore more detail is normally present than is required for our purposes. (It is not usually necessary to understand the semantics of an instruction in order to assemble its components, but the machine description must contain a complete definition of the semantics). In addition, as long as it is possible to write obscure (hard to understand) programs in any language, one can write obscure machine descriptions, and the assembler generator must be capable of unscrambling them. Finally, we have not yet demonstrated that machine properties alone will provide answers to all of the design questions facing the assembler generator. We will focus on these problems in remaining chapters.

3.2 Machine-Description Languages

The next task is to choose an appropriate machine-description language from the large number available; [Chu74] contains a survey of the notations currently in use. Rather than exploring each in turn, we can prune our search substantially by first considering the levels of detail at which a computer system can be described. Five distinct levels are defined in [Bell71].

(5) System level: The top level describes the machine in terms of its gross components: processors, memories, switches, peripherals, etc. It corresponds to the block diagram found near the front of most machine manuals.

(4) Programming level: The machine's instructions and its instruction interpreter are defined in terms of basic operations, data-types, and memories. This represents the information contained in the programming manual.

(3) Register Transfer (RT) level: Operations and sequencing are described in terms of a set of registers and functional or conditional transfers between registers. Timing begins to emerge as a major consideration at this level.

(2) Switching circuit level: The operations of the machine are defined in terms of gates, flip-flops, and boolean equations. Sequencing is determined by clocks or event-done signals, and timing is measured in gate delays.

(1) Circuit level: Gates and flip-flops are constructed from diodes, transistors, resistors, etc. Time is continuous, and transient behavior becomes an important consideration.

The nature of our desired machine-description language is immediately evident; we want to be able to ignore most of the structural details of the machine (how it is physically realized), without sacrificing an unambiguous definition of the operations it performs. Therefore, we do not need to consider any description above the programming level, or below the RT level.

Most RT languages have been developed primarily for machine simulation, and they have taken on a strong degree of similarity [Barbacci73]. For this reason they are closer to the the switching circuit level than the programming level; they represent a structural description of the machine, in which each variable corresponds to a hardware register or set of wires and all operations have physical counterparts. DDL (Digital system Design Language [Duley68]) and CDL (Computer Description Language [Chu72]) are examples of structural description languages. In general, these languages present too much detail to be useful for our purposes.

On the other hand, one notation, ISP (Instruction Set Processor [Bell17Ø, Bell171]), was developed as a semi-formal means for describing some properties of machines to computer professionals; it is intended to formalize and eventually replace much of the information found in programming reference or "princples of operation" manuals. In ISP, the description is behavioral, in terms of input/output relations; the physical hardware is treated as a black box, and structural details can be ignored. Instead of adders, state-indicators and clocks, ISP´s components are data-types, memories, and interpreters. Thus it is more suitable than other RT languages for our purposes, since it deals with the interface between the programming level and the RT level.*

3.3 Instruction Set Processor

The most complete description of ISP appears in [Bell171]; other references include [Bell172, Barbacci72, Goldman74, Siewiorek74, and Barbacci75]. Each of these works contains a different interpretation of the original definition. Since ISP was designed as a notational tool to be processed by humans (much like mathematical notation), its definition was necessarily informal and open-ended,

* An enlightening example of a machine described at three different RT levels -- structural, functional, and behavioral -- appears in [Barbacci73]; all three are given in ISP, demonstrating its ability to describe lower levels of design.

allowing for future development and evolution. Since our application requires a formally defined, machine-processable language, yet another interpretation of ISP is defined here (henceforth, this version is called ISP′ when the distinction is important).

Appendix B contains a formal definition of the syntax and semantics of ISP′; we present here the general structure common to all machine descriptions through a simple example (the AJP-21a computer, a modification of a machine described in [Perlis72]). As with all Algol siblings, ISP begins with a set of declarations, followed by functions and procedures. The declarations define the memories and registers of the machine, subdivided according to their relationship to the processor: the Mp-State (the primary memories in which instructions and data are stored), the Pc-State (registers and flags within the processor itself), and the External-Pc-State (memories associated with the console or other input/output devices or attached processors).

```
Mp_State

   M/Primary_Memory[Ø:1Ø23]<Ø:15>
   X/Index_Register<Ø:15> := M[Ø]

Pc_State

   AC/Accumulator<Ø:15>
   PC/Program_Counter<Ø:9>
   C/Condition_bit
   Run
```

```
External_Pc_State

   box/IO_port<Ø:15>
```

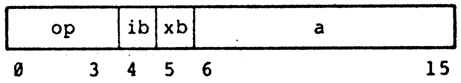
The slash operator (/) indicates abbreviation; the right-hand name is actually treated as a comment. Array and bit dimensions are enclosed in square and angle brackets. The colon-equal operator (:=) is used for definition, in this case to specify another name (X) for the first word of memory; presumably, it has a special role.*

The next section of the description defines the instruction format; unique names can be assigned to arbitrary fields within the instruction.

```
Instruction_Format

   I/Instruction<Ø:15>
      op/op_code<Ø:3> := I<Ø:3>
      ib/indirect_bit := I<4>
      xb/index_bit    := I<5>
      a/address<Ø:9>  := I<6:15>
```

This corresponds to the conventional instruction format diagram:



In general, it is possible to have several names for the same set of bits, and some fields may even overlap. The

* The role of X is impossible to determine, until we see how it is used. This is true of all the memories; the ISP processor does not understand English!

declaration merely indicates the naming; the actual formats depend on how the individual fields are combined and used. For this example, fifteen combinations are possible, since at least one field must represent an op-code, but all other fields might be optional.

Since computers process a variety of data, ISP includes a large set of data-types and data-operations. The term data-type refers to both the meaning of the information (boolean, digit, character, integer, real) and its encoding or representation (BCD, complement, true complement, sign-magnitude, floating-point). The encoding is defined in terms of a data-carrier which is some unit of memory (bit, nibble, byte, half-word, word, double-word). Data-operations are defined with the following precedence (the most binding groups are listed first).

selection operators	
bit selection	<>
address selection	[]
construction operator	
concatenation	.
unary operators	
complement	~
negate	-
absolute value	+
binary operators	
addition	+
subtraction	-
multiplication	*
division	/
remainder	\
exponentiation	^

relational operators	
less than	lt
less than or equal	le
equal	eq
not equal	ne
greater than or equal	ge
greater than	gt
logical operators	
and	&
inclusive or	|
equivalence	=
exclusive or	#
transfer operator	->

Operators within each group are of equal precedence and are performed left to right; parentheses are used for grouping in the usual way. Since most of these operators apply to more than one data-type, modifiers may appear in expressions, enclosed in braces. For example,

```
M[e]+AC -> AC {float}
```

indicates a floating-point addition. In this case, float applies to all the operands, but a modifier can also be used to indicate the operation type.

```
AC*(2^s) -> AC {rotate}
```

In the absence of modifiers, all data-types and operations are taken to be two's complement binary integer.

The procedural part of the description is divided into three sections; the first contains macro definitions of functions which are common to several machine instructions. The effective address calculation is the most typical of these, so it titles the section, but actually any common computation may be defined here.

```
            Effective_Address_Calculation

               e/effective_address<Ø:15> :=
                 (~ib & ~xb => a;
                  ~ib &  xb => a+X;
                   ib & ~xb => M[a];
                   ib &  xb => M[a+X])
```

The "=>" operator (read "implies") corresponds to the Algol conditional statement "if ... then ..."; if the left-hand expression is true, the right-hand expression is evaluated.

The next section defines the machine´s instruction interpreter: the well-known fetch-execute cycle; it is defined as a procedure, rather than a function, since it has no value (in ISP´, procedure names are enclosed in quotes to distinguish them easily from memories and functions).

```
          Instruction_Interpretation_Process

            ´Interpret´ :=
                Run =>
                  (M[PC] -> I; 1+PC -> PC next
                   ´Execute´ next ´Interpret´)
```

Actions separated by a semi-colon are performed in parallel; this is the normal mode of evaluation in ISP. When sequential execution is required, the "next" operator is used; transfers on the left of the "next" are performed simultaneously. Thus "A -> B; B -> A" interchanges the values of A and B, whereas "A -> B next B -> A" is essentially just "A -> B".

Finally, each instruction of the machine is defined by giving both the conditions under which it is executed and its action-sequence. A special form of the conditional statement allows a string to be associated with the conditions; it is interpreted as the op-code mnemonic.

```
     Instruction_Execution_Process

       ´Execute´ :=
         (´JMP´ (:= op eq  Ø) => e -> PC;
           ´JIF´ (:= op eq  1) => C => e -> PC;
           ´SIT´ (:= op eq  2) => (X-1 -> X next
                                    X ne Ø => e -> PC);
           ´HLT´ (:= op eq  3) => Ø -> Run;
           ´ADD´ (:= op eq  4) => AC+e -> AC;
           ´SUB´ (:= op eq  5) => AC-e -> AC;
           ´MPY´ (:= op eq  6) => AC*e -> AC;
           ´DIV´ (:= op eq  7) => AC/e -> AC;
           ´EQL´ (:= op eq  8) => AC eq e -> C;
           ´GTR´ (:= op eq  9) => AC gt e -> C;
           ´AND´ (:= op eq 1Ø) => (AC&e -> AC next
                                    ~AC eq Ø -> C);
           ´NOT´ (:= op eq 11) => (~AC -> AC next
                                    AC<15> -> C);
           ´LOD´ (:= op eq 12) => e -> AC;
           ´STO´ (:= op eq 13) => AC -> M[e];
           ´GET´ (:= op eq 14) => box -> M[e];
           ´PUT´ (:= op eq 15) => e -> box)
```

As this example demonstrates, an ISP description defines the machine with as little structural detail as possible. It does not specify the way in which the processor is organized (although it may imply some obvious choices), and it does not contain all of the memories, operations, and sequencing necessary to realize a physical implementation. For example, we might suspect, on examining the effective address calculation, that there is actually a register for this in the Pc-State, and that the calculation is really a two step process (first indexing the instruction address, then indirection). While the ISP could be modified to reflect this (see appendix C), such details are irrelevant to understanding the machine's instructions.

3.4 The Machine Space

Naturally the decision to use a machine-description language, and ISP in particular, has some important consequences; one is the class of machines that will be considered. Since ISP was designed to describe a specific set of data processing machines (namely stored-program computers with random access memories), we can assume that every ISP description contains a definition of an instruction interpreter, which uses a program counter to locate, fetch, and execute instructions. In general, we can assume that every description follows the sectioning outlined above, and that we know a good deal about the nature and organization of the information in each section.

As with all artificial languages, ISP represents a particular view of the subject matter it deals with, and as such it describes some machines better than others. For example, the description of the IBM-1401 [IBM60] contained in [Bell71] is highly simplified; most of the essential information necessary to understand the machine is contained in the (English) comments. In this case, the problem can be traced to the major data-type of the machine: the variable length character string. It is not a primitive in ISP, and

it is difficult to define in terms of the basic types without a specialized (recursive?) declaration.* While such a definition facility could have been provided, it was not, probably because such data-types (and operations) are uncommon as machine primitives. Thus we are justified in restricting the system to the more popular fixed length binary (and decimal) data-types, for which all the operations are well understood.

The point to be made here is that there are some simple machines whose complete ISP descriptions can be enormously complex. On the other hand, some machines, particularly large, general-purpose computers, are inherently complex, regardless of the language used to describe them. In general, if the machine is difficult to describe, we can expect any automatic programming system to have difficulty analyzing the description, and we should be prepared for lousy results.

* A more accurate description of the 1401 could be given in ISP, but it would be overloaded with details. It is not sufficient merely to define the data-type; specification of how all of the operators apply to that data-type must also be included. For example, if adding individual characters is taken as the primitive operation, the description of the process for character strings involves an eleven state FSM [Bell71, p. 229]. This amount of detail seems inappropriate for a machine which, from the programming point of view, is relatively simple; it has only four data-types (character, address, instruction, and string) and 24 to 30 op-codes (depending on special features).

There is another consequence arising from the choice of ISP: since it was designed primarily as a notational tool, the language was not defined formally. We must turn the ISP notation into a programming language, with a concrete syntax and well-defined semantics. In doing so, it is easy to become fascinated with the language for its own sake, implementing every feature and attempting to make sense out of every syntactically valid construct. I have taken the opposite view; wherever possible, the simplest interpretation is assumed, and the most straightforward solution is used.

Some of these decisions concern the representation of the publication language, and are purely syntactic. For example, the right-going assignment operator, while uncommon, simplifies the parsing; so does enclosing procedure names in quotes. Other decisions involve semantic issues: the general problem of equivalencing two memories with substantially different structures, for example. Appendix B contains a technical discussion of the semantics of the language. Other problems are related to implementation details: arbitrary precision arithmetic has not been included, so that some computers with long word lengths cannot be handled. Thus some machines are eliminated from consideration, but this is important to this research only if the computer demonstrates some new concept which influences the assembler (for example, byte

addressing). In this case we are free to invent a hypothetical machine which illustrates the concept but which also falls within the implementation limits.

Finally, the decision to use a machine-description language considerably enlarges the focus of this research from implementing assemblers to developing programs that understand machine descriptions. One can argue that this enlargement is not appropriate, since assembler design should concentrate on programming issues: relocation, binding time, macros, libraries and the like. This is true, but it is not the point. Assemblers were chosen as an interesting, useful, and hopefully achievable instance of the automatic programming problem, not the other way around.

4. THE ISP PROCESSOR

The problem now at hand is the analysis of an ISP description, keeping in mind that the ultimate goal is the automatic design and implementation of an assembler for the machine being described. The first step is to compile the description, by accumulating and storing declarative information in a symbol table, and by generating symbolic code for the procedures and functions, while at the same time filtering out syntactic and semantic errors (details are in appendix B).

The goal of the analysis is twofold. First, the concepts taken for granted in every assembler -- the notions of address, op-code, operand, and instruction format -- must be precisely defined in terms of ISP constructions. Second, based on these definitions, algorithms must be developed which recognize and extract the relevant instances of these constructs. We will begin with relatively simple algorithms, and extend them to handle more and more elaborate cases.

4.1 Basic Properties

One of the techniques used in analyzing a program is to attempt to determine the role of each of its variables.

Usually this can be done by examining all references to the variable, recording the types of operations performed on it, and noting the local context in which it is used. For example, it is easy to identify indexing variables by examining all of the array subscripts in the program; in this case the context is defined syntactically by the production for subscripting simple variables. We will call this technique local analysis, since it depends only on the local context, as determined by the syntax of the language.

In processing ISP, local analysis is used to determine the properties of the machine variables. Of course, some properties follow directly from the declarations, the most notable being primary-memory, processor-memory, and instruction variable. But more interesting properties depend on how the variables are used; these are assigned by procedures invoked in a syntax-directed manner. For example, the production for conditional statements "exp => action" invokes a procedure which assigns the boolean property to all variables in the expression. Likewise, the subscripting productions assign the address property to variables in the subscript expression, provided the subscripted variable is a primary memory. Other property routines may be invoked recursively when specific constructs are recognized. For example, the address routine invokes the index-register property routine whenever it encounters an addition or subtraction operation.

The union of a variable's properties determine what the variable represents (op-code, address, index-register, etc.); this information is required by the assembler, so that it can define an appropriate symbolic representation. It is possible that a variable is used in more than one way, as part of an op-code in one instruction, and as an address mode in another, for example. So properties are assigned both on a global basis (in the symbol table), and locally with each reference (in the generated code).

Since local analysis operates independently of the global context, it is often easily led astray; there is a tendency to try to infer too much information from a limited set of facts. For example, any construct of the form "M[e] -> I", where "M" is a primary memory and "I" is an instruction register, represents an instruction fetch operation; we would expect to be able to identify the program counter by analyzing the subscript expression. But this is true only if the global context is an instruction-fetch procedure of the machine's interpreter. If instead the statement appears in an effective address calculation, it might indicate a multi-level indirect addressing operation, which repeatedly fetches address words until a direct address is found. So in order to analyze any construct in detail, the complete context in which it occurs must be known.

4.2 Environments

Actions in the machine description do not execute in a vacuum, but rather under a well-defined set of conditions; we will call this set of conditions an environment.* To determine the environment at any particular point in the description, we record all of the conditions which must have been met in order for that point to be reached. For our purposes, we are especially interested in the environment at the time an op-code mnemonic is defined, so we will begin with that environment as the goal. In simple machines, such as the computer described in section 3.3, the environments associated with the mnemonics depend only on a single instruction field, so they are easily constructed.

However, in the general case, the environment associated with a mnemonic may depend on any combination of instruction bits, and it may involve an arbitrary number of nested conditions. Consider, for example, the PDP-8 [DEC67] instruction-execution process. (Appendix D contains the complete ISP description of this machine.)

```
'Execute' :=
   ('AND' (:= op eq 000b) => M[z]&AC -> AC;
    ... ;

    op eq 111b => 'Operate')
```

* This concept is similar to the environment defined in [Ruth74], which is used in analyzing sort programs.

```
'Operate' :=
   (~opr => ... ;
    opr & ~eae => ... ;
    opr & eae => 'EAE')

'EAE' :=
   (... ;

    'NMI' (:= mic eq 100b) => ... ;
    ... )
```

Local analysis would deduce that the op-code for 'NMI' is simply "mic eq 100b". But the environment at this point contains much more information, namely

```
op eq 111b & opr eq 1 & eae eq 1 & mic eq 100b
```

So it is appropriate to define the op-code corresponding to a mnemonic to be a copy of the environment at the point it is defined. In addition, the value of the op-code is simply the value of the instruction variables in that environment, with all the predicates taken to be true.

In a general programming language, an environment is difficult to maintain, since arbitrary branching is allowed; but for ISP the algorithm for determining the environment associated with any statement is straightforward. Environments are represented as a stack of predicates. Whenever a conditional statement "exp => action" is encountered, the expression is added to the environment and the action is processed; when it completes, the expression is deleted. Strictly speaking, the expression should be checked against the current environment before it is added, to make sure that it is currently undefined. For if it is defined, its value is known and there is no reason to re-test it (in the language of schemata theory, the schema defined by the ISP description is not free).

This process will completely define all of the op-codes of the machine mnemonics. (Environments could be attached to procedures and functions as well, but there is currently no need to do so). The next task is to analyze the procedure body of each instruction to determine its format, that is, the instruction fields that it references.

4.3 Instruction Formats

In machine manuals, a format is usually defined as a picture of the instruction; here they will be represented as a list of instruction fields, ordered by their absolute bit position. Included with each field is a list of its local properties.

To construct the format, the list is initialized with the instruction variables contained in the mnemonic's environment, and the op-code property is assigned to each variable; the code is then scanned in a top-down manner. Whenever a variable with the instruction property is encountered, it is added to the list (along with its local properties), checking to make sure it does not overlap existing fields. Function and procedure calls are handled by pushing the current state and performing the process recursively on their code bodies. However, in this case looping must be ignored; although ISP contains no "go-to" statement, procedures and functions can repeat themselves, as in the instruction interpretation process. Loops are ignored by constructing a stack of all active procedures and functions, and never re-scanning an active one.

If this process is carried out on the machine presented in the last chapter, it produces the format "op ib xb a" for most of the instructions. But the ´HLT´ instruction (op eq 3) and the ´NOT´ instruction (op eq 11) have the format "op", since neither of them reference the effective address function "e". Suppose this special case had been described differently.

```
´Interpret´ :=
    Run =>
      (M[PC] -> I; 1+PC -> PC next
       op ne 3 & op ne 11 => e -> E
       next ´Execute´ next ´Interpret´)

´Execute´ :=
  (´JMP´ (:= op eq 0) => E -> PC;
   ´JIF´ (:= op eq 1) => C => E -> PC;
   ... ;

   ´HLT´ (:= op eq 3) => 0 -> Run;
   ... )
```

Here "E" is used as a temporary register in the Pc-State, and all the instructions refer to it, rather than to the function "e". The algorithm as it stands would never examine "e", and all instructions would be analyzed as having the format "op".

The problem lies with the assumption that all of the actions performed by an instruction (other than instruction fetch) are defined in the body of its mnemonic; this is clearly not the case. Conditional actions like the one above might occur anywhere in the interpret or execute procedures, or in functions. But we are guaranteed that all the actions of an instruction are performed within a single cycle of the instruction interpreter. So the solution is to simulate the machine through one iteration of its interpretation process, recording the instruction fields as before. This technique is similar to the program debugging technique commonly called "single stepping".

Of course, the simulation is not done blindly; it takes place in the environment defined by the mnemonic currently at hand. That is, whenever a conditional statement is encountered, the condition is evaluated in the current environment, and the body is processed based on the outcome.

Strictly speaking, a mnemonic definition is just a conditional statement with a name attached, but it is handled in a special way. Normally all such conditions are false (except for the condition which defines the current mnemonic), since only one instruction can execute at a time. But if the condition is undefined -- that is, no assertion in the environment is sufficient to show that it is false -- this is an instance of a micro-coded operation. Consider another example from the PDP-8 [DEC67].

```
          op eq 111b =>
            (~opr =>
               ('CLA' (:= I<4>) => Ø -> AC;
                'CLL' (:= I<5>) => Ø -> L;
                ... );
             ... )
```

Both 'CLA' and 'CLL' may be specified simutaneously, since neither excludes the other. How this is represented in the assembly language is a separate issue (which will be taken up in the next chapter); for the moment we simply record those op-codes which can be combined.

Other undefined conditions, such as "AC>Ø => e -> PC", are treated as if they are true; since they cannot be evaluated at assembly time, it is assumed that there is some machine state for which the condition will be satisfied. In this sense the simulation is by no means a complete one; it need only include actions and conditions involving instruction variables.*

* There exist machines for which assemblers (as defined in section 2.2) cannot be written. As an example, suppose the machine has a mode bit M which indicates whether a long or short integer operation is to be performed. Suppose also that the machine has "immediate" operations (the data, rather than the address of the data, is specified in the instruction). The assembler, since it does not know the runtime value of M, cannot assemble any immediate instructions, since it cannot determine their lengths (axiom 2.2.5.3 has been violated). Of course, a MODE pseudo-op could be added to specify the value at assembly time. Or separate mnemonics could be defined for long and short operations (as in the PDP-11 [DEC71] floating-point option). But both solutions introduce a new way to create bugs: the assembly-time and run-time values of M may not agree.

This observation suggests that the environment should be expanded to include assignment actions when they involve instruction variables. However, statements of the form "M[PC] -> I[Ø]" cannot be handled in an obvious way, since "M[PC]" is undefined. Instead, a predicate is added to the environment of the form "exists I[Ø] & valid I[Ø]"; any subsequent assignment to the same instruction field would delete the "valid" predicate. This makes an instruction fetch trivial to locate without any pattern-matching (it is the first assignment to the instruction). It also handles the case where instruction registers are used as temporary registers in intermediate calculations, thereby destroying their original meaning. Consider the following (hypothetical) machine.

```
        I/Instruction<Ø:23>
           op<Ø:4> := I<Ø:4>
           xb      := I<5>
           t<Ø:3>  := I<6:9>
           d<Ø:13> := I<1Ø:23>
           a<Ø:17> := I<6:23>

        'Interpret' := Run =>
           (M[PC] -> I; 1+PC -> PC next
            xb => sign_extend(d)+X[t] -> a
            next 'Execute' next 'Interpret')

        'Execute' :=
           ('AD'  (:= op eq Ø & ~xb) => M[a]+AC -> AC;
            'ADX' (:= op eq Ø &  xb) => M[a]+AC -> AC;
             ... )
```

Even though both add instructions look identical, in the first "a" refers to the instruction field, but in the second it refers to the calculation "sign_extend(d)+X[t]". The second reference is ignored since "a" is no longer valid.

4.3.1 Multiple Formats

Since the two add instructions of the last example are so similar, one is tempted to combine them into a single mnemonic.

```
'ADD' (:= op eq 0) => M[a]+AC -> AC;
```

Unfortunately, this violates another assumption: that the format associated with a mnemonic is unique, and can therefore be determined knowing only the op-code. In this example, however, the 'ADD' instruction has two formats ("op xb a" and "op xb t d"), depending on the "xb" bit, which is unknown in the op-code environment.

The root of the problem can be traced to the handling of conditional statements during the simulation. It is apparent that undefined conditions, when they involve instruction variables, are potentially assembly-time calculations, even though they are not defined in the op-code environment. Different interpretations of these predicates may lead to different formats, although this is usually not the case.* However, it is impossible to predict whether multiple formats are present; the only general solution is to construct the format for each possible interpretation of the conditions.

* Of the forty or so machines described in [Bell71], only two exibit this behavior (the IBM-1401 [IBM60] and the IBM-1800 [IBM66]).

When, in the course of the simulation, a predicate is encountered which involves an instruction variable that is undefined in the current environment, new instances of the simulation are spawned for each possible value of the variable, with the environment updated to reflect that value. Of course, these new simulations may create still more instances involving other instruction variables. Each simulation will eventually produce its own version of the format (called a sub-format), together with the set of conditions that distinguish it from other cases (called the format environment); sub-formats are then combined if possible.

The following example, a simplified version of the IBM-1800 "modify-index" instruction [IBM66], will clarify the procedure.

```
I/Instruction[0:1]<0:15>
   op<0:4> := I[0]<0:4>
   f       := I[0]<5>
   t<0:1>  := I[0]<6:7>
   d<0:7>  := I[0]<8:15>
   ia      := I[0]<8>
   a<0:15> := I[1]<0:15>

'MDX' (:= op eq 01110b) =>
   (t eq 0 & ~f => d+PC -> PC;
    t eq 0 &  f => d+M[a] -> M[a];
    t ne 0 & ~f => d+X[t] -> X[t];
    t ne 0 &  f & ~ia => a+X[t] -> X[t];
    t ne 0 &  f &  ia => M[a]+X[t] -> X[t])
```

A total of five separate simulations are created, corresponding to the possible values of "t", "f", and "ia" (the case "t eq Ø" does not generate subcases for "ia"). They lead to the following environments and sub-formats:

```
1:  t eq Ø &  ~f        =>  op f t d
2:  t eq Ø &   f        =>  op f t d a
3:  t ne Ø &  ~f        =>  op f t d
4:  t ne Ø &   f & ~ia  =>  op f t ia a
5:  t ne Ø &   f &  ia  =>  op f t ia a
```

Identical formats are then merged by logically "or"-ing their conditions, yielding three distinct sub-formats for this op-code.

```
1+3:  ~f          =>  op f t d
  2:  t eq Ø & f  =>  op f t d a
4+5:  t ne Ø & f  =>  op f t ia a
```

Notice that the five cases correspond to three rather different actions: the first is a relative jump, the second increments memory, and the rest increment an index register. This suggests that each action should probably have been defined as a separate mnemonic.

Because of the considerable amount of work involved, it is worthwhile to do as much pre-processing as possible. Since functions are factored out because they are referenced frequently, this process should be applied to them individually. What is really needed is a technique which determines if multiple formats are possible. Some good heuristics can be defined (for example, if an instruction fetch is conditional on a variable not defined in some op-code environment, multiple formats are likely). However, there appears to be no general rule which can guarantee that multiple formats are not present.

4.3.2 Instruction Forms

To simplify the analysis, we have so far taken a simple definition of an instruction format: a list of instruction fields. In practice, a format element often represents (from the programmer's point of view) an assembly-time computation, rather than a field value. The computation usually involves the program counter and simple address operations (plus, minus, shift, or sign-extend), and it may represent the value of more than one field of the format. Computations of this nature will be called forms, as opposed to fields; when constructing formats, both are considered atomic elements.* For example, if "d<Ø:7>" is an instruction field, the following is a form:

```
rel<Ø:15> := PC+2*sign_extend(d)
```

Clearly, instructions which reference "rel" should be handled differently than those which reference "d" directly, since "rel" can be computed by the assembler. This will be the case if we treat "rel" as an instruction field different from "d" while constructing formats, provided forms can be defined in a rigorous way.

* Forms are similar to syntax macros [Leavenworth66].

Forms must satisfy several requirements. Obviously they must be computable at assembly time; that is, they may only reference constants, instruction fields, the PC, and other forms. Secondly, the form must have an inverse; given the value of the form (the programmer´s representation), the value of all the fields used in the form must be calculable. Finally, the form must be isolated as a separate function; this gives the user some control over what computations are performed implicitly by the assembler.

We can use the local analysis techniques described in the first section of this chapter to determine if a function is computable at assembly time; but the inverse property is the difficult one. Consider the following example from the PDP-8 [DEC67].

```
Effective_Address_Calculation
   PC´<0:11> := PC<0:11>-1
   page<0:4> := (~pb => 0; pb => PC´<0:4>)
   a<0:11> := page.pa
   z<0:11> := (~ib => a; ib => M[a])
```

Here "ib" (the indirect-bit), "pb" (the page-bit), and "pa" (the page-address) are instruction fields, and PC is the program counter. All but the last function are forms ("z" is not assembly-time computable since it references M). But it is not clear how to calculate "pb" given the value of

"page"; worse still, if the value of "page" is not zero and not equal to "PC´<0:4>", then "pb" is undefined. The conditional operator is suspect, but "page" can be defined without using an implication.

```
page<0:4> := sign_extend(pb) & PC´<0:4>
```

It is still unclear how to find its inverse symbolically.

To ensure that all forms have well-defined inverses, the set of operators which are recognized as assembly-time computable is restricted to addition, subtraction, multiplication and division by constants (shifting), sign-extention and truncation. These include practically all of the addressing schemes found in existing computers. In any case, failure to recognize a form is not a fatal error; at worst it leads to an awkward representation (in the above case, "pb" and "pa" would be specified separately, rather than as a single operand).

Note that the program counter is allowed to appear in forms; this implies that its value must be known whenever a form depending on it is referenced. This is accomplished by adding it to the environment as another variable to be simulated. However, the rules governing assignments to the PC are different than those for instruction variables; the behavior of the PC must be limited to operations involving instruction fetch. Jump, branch and skip instructions should be ignored, since they are not assembly-time operations. First, the left hand side of any assignment to the PC may only involve constants and the PC itself. Second, if the assignment is conditional (at any level of nesting), those conditions must be defined in the current environment. In all other cases, the PC is marked as invalid, and the assignment is not performed. The final requirement is that if a form involves the PC, its value must be identical for all references to the form.

Since the final value of the PC is now known for each format, the length of instructions can be measured in something other than bits. This is the subject of the next section.

4.4 Addressing Schemes

Formats provide us with a picture of the instructions. But they imply little about the addressing scheme of the computer. Knowing, for example, that all instructions are 32 bits long does not imply that addresses designate 32-bit words; they may refer to 8-bit units (as in a "byte-addressed" machine), or 64-bit units (if instructions are "packed", two to a word). What we are searching for is the underlying unit of measure of the machine -- which will be called a byte -- the smallest addressable unit of information. The assembler maintains the values of all the label symbols, and in particular the location counter, in terms of this unit, so it is vital for our purposes.

Because the assembler's location counter simulates the local behavior of the program counter, analysis of the operations on the PC is a likely place to start. The PC, after all, is the one register that always contains an address in every machine, so we expect it to represent the addressing scheme. In fact, the most common form of instruction fetch,

```
M[PC] -> I; 1+PC -> PC
```

defines the bytesize: it is the bit-width given in the declaration of "M". For a byte machine, with a memory defined by "M[Ø:2^16-1]<Ø:7>", the fetch would be written

```
M[PC:PC+1] -> I; 2+PC -> PC
```

Since 16 bits are fetched, and the PC is incremented by two, this construction defines an 8-bit bytesize.* On the other hand, a bit-stingy hardware designer might take advantage of the fact that instructions are always located at even addresses (that is, aligned on word boundaries) and write:

```
Mp_State

   M[Ø:2^16-1]<Ø:7>
   Mw[Ø:2^15-1]<Ø:15> := M[Ø:2^16-1]<Ø:7>
```

* Note that the structure of "I" is completely independent of this calculation, except for the total number of bits transfered; its structure is usually determined by the instruction fields, rather than by the addressing scheme.

```
Pc_State

   PC'<0:14>

  'Interpret' :=
    (Mw[PC'] -> I; 1+PC' -> PC'
     next ... )
```

This actually represents an entirely new (but related) addressing method. The original scheme selected words or bytes using 16-bit addresses, but the new scheme uses 15-bit addresses to designate words (and bytes can not be addressed at all). In a large general-purpose byte-addressed machine, there may be several address types, one each for bytes, half-words, words, double-words, etc.

The assembler normally uses the method with the smallest bytesize (largest address size), and all symbol values are assigned in those units, with conversions to other address types performed as necessary. This solution will always work, but it sometimes results in very unnatural representations, particularly for machines with packed instuctions. Consider a simplified version of the CDC-6600 [CDC67a]:

```
Mp_State

   M[0:2^18-1]<0:59>
   Mi[0:2^20-1]<0:14> := M[0:2^18-1]<0:59>

Pc_State

   PC<0:17>
   pc<0:1>

Instruction_Format

   I<0:29>
```

```
'Interpret' :=
   (Mi[PC.pc] -> I<0:14>; 1+PC.pc -> PC.pc next
    long => (Mi[PC.pc] -> I<15:29>; 1+PC.pc -> PC.pc)
    next 'Execute' next 'Interpret')
```

Here the "pc" is used as a two-bit extension of the "PC" to allow quarter-word addressing; they might have been combined and declared as a single register. This construction represents the dual of the byte-addressing case: instead of the instruction address size being shorter than the memory address size, it is longer. In this case, the assembler would interpret symbol values as 20-bit addresses. But this is unnatural from the programmer's point of view, since all operand addresses, including branch addresses, are 18 bits. This is typical for packed-instruction machines; one normally can not jump into the middle of a word.

From the assembler implementation standpoint, the choice of addressing scheme is somewhat arbitrary, but it may be important to the programmer. We will therefore assume that the primary memory declaration reflects the addressing scheme that the programmer wants to use, and that the program counter is consistent with this scheme (its bit length is the same as the address size of the memory). Note that in the case of packed instructions, this forces partial word selectors to be specified with bit-scripts, making this situation easy to recognize.

This completes the basic analysis of the machine. We now know, for each instruction, the value of its op-code, all of its formats, and their lengths in terms of the addressing scheme. The next task is to define a representation for each instruction (the syntax of the assembly language), and to produce the appropriate semantic routines.

5. THE ASSEMBLER GENERATOR

The basic structure of the assembler has already been described (section 2.4) in terms of its modules and their functions. The generation process follows the same organization, one generator for each module. While each module design was based on a general model of the task to be performed, the generator of the module is concerned with eliminating generality wherever possible, in order to achieve efficiency; this is done within the framework of the model.

The algorithms and major data structures used by each module have been predetermined (to within a small number of possibilities) by the requirements stated for the module in chapter 2. These requirements are defined in terms of specific subroutines which the module must provide. For example, the SYMBOL module must contain a function SEARCH(nam) which returns a pointer to the symbol table entry of the name. But the details of this function are not fixed. The generator is free to allocate the data structures of the module in any convenient manner, to rearrange code sequences for optimization, and even to employ a variety of search algorithms, so long as the interfaces between modules are not disturbed. In short, the generators handle implementation details.

The syntax of the assembly language will be developed first. Since the translation from source to binary is based on syntax-directed techniques, the syntax plays a crucial role in the overall design. Because semantic routine calls are embedded directly in the productions of the language, the syntax determines how the semantic routines are organized and when they are invoked. Fortunately, a good deal of the syntax is invariant across all assemblers; the general form of statements, label and parameter definitions, most pseudo-ops, and assembly-time expressions is fixed. Therefore we can concentrate on the representation of instruction and data formats.

Likewise, the discussion of the semantics will emphasize those routines which construct instruction and data formats from their components and output the results. While the purpose of these modules remains the same for all assemblers, the specific actions they perform are highly machine-dependent. The problem is to clearly identify all of these dependencies in every module.

5.1 Symbolic Representation

The general form of the syntax was outlined in section 2.2.2; we will begin with the productions defining statements of the language and tailor them to the specific set of formats defined by the machine.

```
<stm> ::= .nam  <exp> , <exp> , ... , <exp>
```

Here ".nam" is a lexical type standing for any op-code (or psuedo-op) and each expression represents an operand.* This representation is much too general; it does not distinguish one format from another, nor does it specify the number of operands allowed. Strictly, it is not necessary to do so. We could generate a single semantic routine for this production, although it would be a rather cumbersome one, due to the large amount of case analysis and error checking it would need to perform.

Instead, separate productions for each possible format will be generated; this approach has several advantages. First, it simplifies the semantics; several small, localized routines are used instead of one large one, and much of the error checking is handled by the syntax. Second, the syntax can be tailored to each format, and the representation of each operand can be specialized, without disturbing the rest of the language. Finally, the rules for acceptable inputs are defined in a form the user can readily understand, rather than being buried in the program. The syntax becomes part of the assembler documentation.

* In the meta-syntax, lexical types begin with a period to distinguish them from terminal symbols.

In general, inventing a syntax for a language is not a well-defined process; it depends heavily on personal preference and taste. Fortunately, if the problem is restricted to the set of assembly languages, there are some underlying principles which serve to limit the possible representations. The remainder of this section defines those principles. It should be noted, however, that the syntax of the language is not unique, and some choices will be made arbitrarily.

5.1.1 Op-codes

In most assembly languages, the op-code part of a format is represented by a mnemonic which begins the instruction and serves as a signal for what follows (the operands). Chapter 4 contains a definition of an op-code in terms of ISP: a copy of the environment when a mnemonic is defined. From a user´s point of view, however, the distinction between an op-code and an operand is not always clear; a mnemonic definition can be placed almost anywhere along the execution path of an instruction. Consider the IBM-360 "branch-condition" instruction [IBM70]:

```
´BC´ (:= op eq 47) => m[CC] => e -> PC
```

A 2-bit processor-state variable called the condition-code (CC) is combined with a 4-bit instruction mask field "m" to determine if a branch takes place. In this case, "m" is clearly an operand, since it is not part of the op-code

environment. But each value of "m" could be represented by a different instruction mnemonic:

```
´NOP´ (:= op eq 47 & m eq Ø) => e;
´BC1´ (:= op eq 47 & m eq 1) => CC eq 3 => e -> PC;
´BC2´ (:= op eq 47 & m eq 2) => CC eq 2 => e -> PC;
´BC3´ (:= op eq 47 & m eq 3) => CC eq 2|CC eq 3 => e -> PC;
...
´BR´  (:= op eq 47 & m eq 15) => e -> PC
```

Of course, more meaningful op-codes would probably be used, based on the interpretation of the condition-code bits. In any case, this example shows that the user has considerable flexibility when defining mnemonics. The generator can also use this technique (under conditions to be defined) when it is processing operands.

5.1.1.1 Extracting Op-codes

Mnemonics in assembly languages play the same role as keywords in ALGOL and FORTRAN; they indicate the type of statement to be processed, as well as the form of the input that follows. For our purposes, the mnemonic will determine the instruction format, as well as the value of the op-code portion of the format. A unique number is assigned to each format and stored along with the value in the mnemonic´s symbol table entry.

For subsequent processing, the op-code fields should be deleted from each format, leaving only the operands. But this may change the equality relationship within the format set; previously matching formats may not be equal when the op-code fields are ignored. In particular (for the example above), the formats ".fmt1: op | m e" and ".fmt2: op m | e" should not match.* We could reorganize the format set in a post-processing step, but it is easier to modify the format analyzer to check the op-code property of each field when it is comparing formats.

For similar reasons, two unequal formats may merge into a single case when their op-codes are deleted; the format "op | e" can be combined with "op m | e", but not with "op | m e". In general, two formats can be merged if their lengths are equal and their operands are identical; the structure of the op-code part can be ignored completely. This is because the assembler treats op-codes as bit-strings, and does not need to know the details of their sub-structure.

5.1.1.2 Micro-coded Instructions

As described above, the merging process is purely an optimization step which reduces the amount of syntax by reducing the number of formats. But when the machine contains micro-coded op-codes and several instructions can be combined, their formats must be merged, in all possible combinations, subject to some additional constraints. Syntactically, the op-code fields are represented by a list

of mnemonics, rather than a single symbol. However, the syntax should be defined so as to permit only the legal combinations. This will be accomplished by constructing a sub-grammar that translates lists of mnemonics into format names. The parser sees this translator (called the op-code machine) as a pre-processor, similar to the lexical analyzer.

During the construction of the formats, as a by-product of the machine simulation, each op-code environment was checked against every other, to determine which instructions could be combined (see section 4.3). If there are N op-codes, the legal combinations can be represented as an NxN bit matrix; it encodes an irreflexive, symmetric binary relation. First eliminate all zero rows and columns; these correspond to non-micro-coded instructions that are translated directly into their formats by a symbol table lookup. Proceed by "or"-ing together identical rows (and columns) of the matrix, ignoring the diagonal elements in the comparison. This process will define groups of op-codes (one for each row) which obey the same combining rules. The following example, a subset of the DEC PDP-8 instruction set (appendix D), will make this procedure clear.*

* In the notation, a colon ends the format name and a vertical bar separates the op-code and operand fields.

* The scope of the "next" operator following 'CLL' is somewhat unconventional. This example uses the description given in [Bell71]; appendix D defines this instruction set differently.

```
'Operate' :=
   ('CLA' (:= I<4>) => Ø -> AC;
    ~I<3> =>
     ('CLL' (:= I<5>) => Ø -> L next
      'CMA' (:= I<6>) => ~AC -> AC;
      'CML' (:= I<7>) => ~L -> L next
      'IAC' (:= I<11>) => 1+L.AC -> L.AC next
      'RAL' (:= I<8:1Ø> eq Ø1Øb) => L.AC*2 -> L.AC;
      'RTL' (:= I<8:1Ø> eq Ø11b) => L.AC*4 -> L.AC;
      'RAR' (:= I<8:1Ø> eq 1ØØb) => L.AC/2 -> L.AC;
      'RTR' (:= I<8:1Ø> eq 1Ø1b) => L.AC/4 -> L.AC);
    I<3> & ~I<11> =>
     ('OSR' (:= I<9>) => Data_switches|AC -> AC;
      'HLT' (:= I<1Ø>) => Ø -> Run) next
    I<3> & I<11> => 'EAE')
```

This machine uses a combination of horizontal and vertical op-code decoding. The following matrix is constructed by the format analyzer. Entry [i,j] is a one if op-codes "i" and "j" can be specified together in a single instruction word.

'CLA'	Ø	1	1	1	1	1	1	1	1	1	1
'CLL'	1	Ø	1	1	1	1	1	1	1	Ø	Ø
'CMA'	1	1	Ø	1	1	1	1	1	1	Ø	Ø
'CML'	1	1	1	Ø	1	1	1	1	1	Ø	Ø
'IAC'	1	1	1	1	Ø	1	1	1	1	Ø	Ø
'RAL'	1	1	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø
'RTL'	1	1	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø
'RAR'	1	1	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø
'RTR'	1	1	1	1	1	Ø	Ø	Ø	Ø	Ø	Ø
'OSR'	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1
'HLT'	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø

Using the first mnemonic in each group as the group name, the reduced matrix for this example, after like rows and columns have been combined, is as follows:

cla	Ø	1	1	1
cll	1	1	1	Ø
ral	1	1	Ø	Ø
osr	1	Ø	Ø	1

To construct the grammar, represent each group by a state in an FSM corresponding to a non-terminal. Since every group is a start state, generate a production of the form "<i> ::= i" for each group. Each one-bit in the matrix corresponds to a legal combination of op-codes, and hence to a legal transition between states. So for each non-zero entry [i,j], a production of the form "<k> ::= <i> j" is generated, where "k" is the intersection of groups "i" and "j". The intersection is used because the relation defined by the matrix is non-transitive. For example, if a "cla" and an "ral" op-code are combined, the intersection of these two rows of the matrix produces an entirely new group (cla.ral) which allows only op-codes of type "cll" in further combinations. The complete augmented matrix for this example is:

cla	Ø	1	1	1	Ø	Ø	Ø
cll	1	1	1	Ø	1	1	Ø
ral	1	1	Ø	Ø	1	Ø	Ø
osr	1	Ø	Ø	1	Ø	Ø	1
cla.cll	Ø	1	1	Ø	Ø	Ø	Ø
cla.ral	Ø	1	Ø	Ø	Ø	Ø	Ø
cla.osr	Ø	Ø	Ø	1	Ø	Ø	Ø

The additional columns do not generate any productions, since there are no op-codes corresponding to the augmenting rows. Finally, the grammar generated from this matrix appears below; each alternate is commented with the row and column number of element which produced it.

```
<cla> ::= cla ;
<cll> ::= cll | <cll> cll [2,2] ;
<ral> ::= ral | <cll> ral [2,3] |
                <ral> cll [3,2] ;
<osr> ::= osr | <osr> osr [4,4] ;
<cla.cll> ::=   <cla> cll [1,2] |
                <cll> cla [2,1] |
            <cla.cll> cll [5,2] ;
<cla.ral> ::=   <cla> ral [1,3] |
                <ral> cla [3,1] |
            <cla.cll> ral [5,3] |
            <cla.ral> cll [6,2] ;
<cla.osr> ::=   <cla> osr [1,4] |
                <osr> cla [4,1] |
            <cla.osr> osr [7,4] ;
```

Notice that in most groups, the grammar allows some op-codes to be combined with themselves; "IAC CLL IAC" is legal, for example. This occurs because the diagonal elements were ignored when combining rows. This substantially reduces the number of groups, and hence the machine size, so the semantic routine which processes the op-codes (OPCODE, section 5.2) must check for this error. It is invoked just before each state transition.

The original goal was to translate op-codes (and combinations of them) into formats; we must now determine the relation between formats and op-code groups. Unfortunately, there is no reason to suppose that, because two op-codes can be combined, their formats are the same. If the formats are not equal, they must be merged, subject to the same rules used in constructing them; an entirely new format may result. This must be done for every possible op-code combination.

The format information is encoded in a matrix similar to the first, where the [i,j] entry contains the format that results when the ith and jth formats are combined (and zero if the combination is illegal). Zero rows (and columns) are deleted and equal rows are combined to reduce the matrix. The result can also be viewed as an FSM (the format machine), where each state corresponds to a format and non-zero entries correspond to productions of the form "<new-format> ::= <old-format> op-code-format". Running this machine in parallel with the op-code machine will generate the format for any legal combination of mnemonics.

Since both machines encode regular grammars, it is appropriate to implement them in a separate module (called MICRO). Physically, it sits on the path between the lexical analyzer and the parser, acting as a filter. Symbols which are not op-codes pass through directly. An op-code causes state changes in the machines, based on its group and format obtained from the symbol table; it eventually emerges as a format, when combinations are no longer possible. The parser uses the format to determine the operand syntax, which will now be constructed.

5.1.2 Operands

Formats have now been reduced to their operand components. In generating the syntax for them, the existence and placement of semantic routines that process the operands must be established, but the details of these routines are left to the semantic generators. If the nth format is ".fmtn: a ... z", a first approximation of the syntax is:

```
<stm> ::= .fmtn <exp> [SETa] , ... , <exp> [SETz];
```

Of course ".fmtn" is the format produced by the op-code machine. Each operand is represented by an expression and has a semantic routine attached to process it. This is sufficient to construct a binary representation of the instruction, but additional bookkeeping actions are necessary to maintain the location counter and produce output. So standard prolog and epilog semantics are added:

```
<stm> ::= .fmtn [PROLOG] ... , <exp> [SETz][EPILOG];
```

This syntax is sufficient, but it can be improved upon considerably. Operands, after all, are more than just expressions (or possibly less); they represent objects, and in assembly languages usually physical objects, such as index registers, condition codes or memory addresses. In determining a representation, the generator should make use of the properties of the operand.

5.1.2.1 Operand Properties

To determine what kinds of properties influence the syntax of an operand, a substantial number of assembly languages were examined [IBM57(650), IBM60(1401), Honeywell62(800), IBM62(7094), Univac(1107), DEC67(PDP-8),

IBM67(360/370), CDC67b(6600), IBM68(1800), GE69(635), DEC71(PDP-11), DEC73(PDP-10), DGC74(Eclipse)]. The example first given in section 2.3 is typical, and it illustrates most of the techniques. The instruction contains an op-code (op), accumulator (ac), indirect bit (i), optional index register (xr), and address (a); a typical instruction might be:

```
MOVE  AC5,1,R2,POINTER+1
```

The representation proposed in chapter 2 used the '@' operator for the indirect bit, and the index register (if any) was enclosed in brackets.

```
MOVE  AC5,@POINTER+1[R2]
```

This suggests that the addressing calculations are the constructs to examine, and that the properties of interest are indirect, relative, indexed, immediate and the like. This presents two problems.

First, the inherent limitations of local analysis techniques make these properties difficult to determine; this was discussed in section 4.1. It is easy enough to conclude, for example, that the "xr" field is used to select one of a number of registers in the Pc-State. But the term "optional index register" implies considerably more information than that: the register must contain an address (or small increment) which is added to (or subtracted from) the effective address (or the operand itself), and so on. This illustrates the second problem: the terms must be rigorously defined, in the light of current machine architectures. This is unacceptable if our techniques are to apply to future machine designs, which are likely to contain new concepts not built into our system.

To determine exactly what these constructs have in common, in a machine-independent sense, we examine the ISP for this example:

```
e<0:17> :=
  (~i & xr eq 0 => a;
   ~i & xr ne 0 => a+R[xr];
   i =>
     (xr eq 0 => M[a] -> i.xr.a;
      xr ne 0 => M[a+R[xr]] -> i.xr.a
      next e))

'MOVE' (:= op eq 200) => M[e] -> R[ac]
```

The example involves changes in the representation of the indirect bit and the index register. Each of these fields is used as a boolean variable in some predicate of the program. It seems apparent that one use of syntax is to select options and make them more readable. So the generator will concentrate on the boolean variables in determining the operand syntax. It will be useful to define additional properties which indicate the type of test made on the field, whether it is bound to a value (as in "xr eq 0") or unbound ("xr ne 0"), and whether the field is also used as an operand (as in R[xr]). The following

heuristics are used in developing the syntax. (To simplify the example, the "ac" field will be ignored.)

If a field is used exclusively as a boolean variable (and not as an operand), it can be represented by a unique operator for each possible value (the null operator is allowed). These operators are obtained from the user interactively, with the restriction that they may not also begin expressions.* A semantic routine called SETfn is attached that sets the field "f" to the value "n". So if the user specifies '@' for the indirect bit in the example above, and '' (null) for its absence, the productions generated (ignoring prolog and epilog semantics) are:

```
.fmt '' [SETi0] <exp> [SETxr] , <exp> [SETa]
.fmt '@'[SETi1] <exp> [SETxr] , <exp> [SETa]
```

Usually the null string represents a zero value, but this is not required. Also, the field is normally a single bit, but multi-bit fields can be handled in this way, although a large number of productions will result.

On the other hand, if a boolean field is also used as an operand, and at least one condition is unbound, the value of the field has a double meaning, as with the index register field in the example. The user supplies operators

* Strictly speaking, they are not in FIRST+(<exp>); see [GRIES71]. This avoids any local ambiguities in the language.

for each of the cases as before, but the unbound cases are treated differently. The field is represented by an operator and an expression, and the semantic routine checks that the unbound condition is met. If the user specifies ´´ (null) for the bound case "xr eq Ø", and ´[´ for "xr ne Ø", the final syntax will be:

```
    .fmt ´´ [SETiØ] ´´[SETxrØ] <exp> [SETa];
    .fmt ´@´[SETiØ] ´´[SETxrØ] <exp> [SETa];
    .fmt ´´ [SETiØ] ´[´ <exp> [SETxrneØ] , <exp> [SETa];
    .fmt ´@´[SETiØ] ´[´ <exp> [SETxrneØ] , <exp> [SETa];
```

Note that the generator does not attach any special significance to the user supplied operators (for instance, that ´[´ is a bracketing character which requires a matching ´]´), nor does it attempt to reorder components of the production, since it has no way of telling when one ordering "looks nicer" than another. These modifications are considered to be "syntactic sugar",* and they should be made directly by the user in the generated syntax file.

The last possibility is to make the boolean field part of the op-code, as outlined in section 5.1. Instead of an operator, the generator asks for an alpha-numeric string for each case, which is used to create a new op-code, deleting the old one. If the variable is bound, the condition is made part of the op-code environment; otherwise,

* Here "syntactic sugar" means any change to the syntax which does not require modification of the semantic routines.

appropriate semantics are generated. Finally, the field is deleted from the format. Note that this will probably require reconstruction of the op-code processors.

All of these techniques are applied optionally, at the discretion of the user. They have been developed primarily for the multiple format case, where their application is required.

5.1.2.2 Multiple Formats

As we discovered in chapter 4, the op-code environment may not be sufficient to uniquely determine the instruction format; multiple sub-formats are constructed in this case, each with a format environment that distinguishes among them. The techniques of the last section can be applied to the conditions in the format environment in order to derive a syntax that is unique for each sub-format. The simplest solution is to assign a different op-code to each sub-format, appending the format environment to the op-code. But this will be unworkable if the format appears often and has a large number of sub-formats (the number of PDP-11 binary op-code mnemonics, for example, would increase from 14 to 5Ø4 [DEC71]). In addition, the sub-formats usually contain compound conditions on several variables, and the user may want to apply different techniques to each field, building up the cases in a tree-like fashion. The following example from the IBM-18ØØ [IBM66] (appendix E) illustrates this.

```
I/Instruction[0:1]<0:15>
   op<0:4>    := I[0]<0:4>
   f          := I[0]<5>
   t<0:1>     := I[0]<6:7>
   d<0:7>     := I[0]<8:15>
   ia         := I[0]<8>
   a<0:15>    := I[1]<0:15>

disp<0:15> := sign_extend(d)

rel<0:15> := PC+disp

z<0:15> :=
  (~f & t eq 0 => rel;
   ~f & t ne 0 => X[t]+disp;
   f & t eq 0 & ~ia => a;
   f & t ne 0 & ~ia => a+X[t];
   f & t eq 0 & ia => M[a];
   f & t ne 0 & ia => M[a+X[t]])

´S´ (:= op eq 10010b) => A-M[z] -> C.A
```

The format analyzer produces the following set of sub-formats, after combining identical cases and extracting the op-code fields:

```
~f & t eq 0  =>  .fmt.0: f t rel
~f & t ne 0  =>  .fmt.1: f t disp
 f           =>  .fmt.2: f t ia a
```

The idea is to split the problem into subcases until a single format remains. Clearly, some variable contained in the environments must partition the sub-formats into at least two distinct groups, since the conditions are mutually exclusive by construction. So first the generator finds the set of variables referenced in all of the format environments;* if there is more than a single variable in

* Strictly, the variable must be referenced in each conjunct of every environment. When formats are merged, their environments are "or"-ed together, so the conditions are in disjunctive normal form.

the set, the user chooses the one to be processed first. He then has the option of adding that field to the op-code or converting it to syntax, as described in the last section. In any case, that variable is used to split the sub-formats, and it is then deleted.

For this example, "f" is the first field processed. The last format will be split off into a group by itself, so it can be processed as before. The following group remains:

```
t eq 0  =>  .fmt.0: t rel
t ne 0  =>  .fmt.1: t disp
```

One more iteration of the process will reduce the group to single format sub-groups, partitioned by the value of "t".

The syntax (or the new op-codes) supplied by the user may be sufficient to distinguish all of the formats in the group, even though the variable chosen does not partition the set into single format sub-groups. Suppose the user decides to convert "f" to syntax, and responds as follows:

```
Operator for f=0 in ~f & t eq 0 => f t rel  ?  #
Operator for f=0 in ~f & t ne 0 => f t disp ?  [
Operator for f=1 in  f          => f t ia a ?  (null)
```

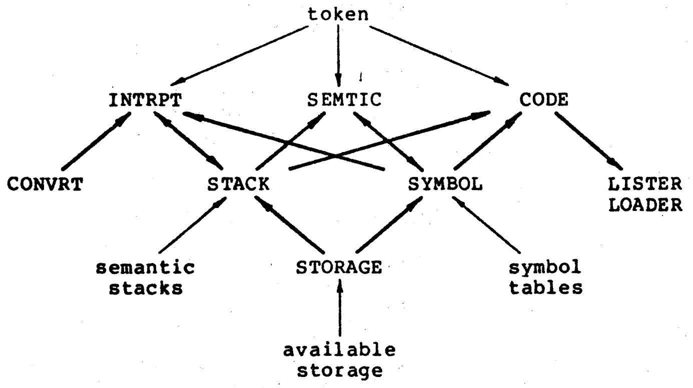
The user has flattened the format tree by combining two levels; the generator must realize that this has happened, and not ask for a decision about "t". It does this by first partitioning the sub-formats based on syntax alone, deleting "f" as before. It then checks to see if the predicate on any variable is the same in every environment of the group. If so, that predicate is subsumed by the syntax, and it can be deleted from the group after generating appropriate semantics. In this example, the syntax supplied for "f" also distinguishes the conditions on "t", so the productions are:

```
    .fmt ´#´ [SETf0] [SETt0] <exp> [SETrel]
    .fmt ´[´ [SETf0] <exp> [SETtne0] , <exp> [SETdisp]
    .fmt ´´  [SETf1] <exp> [SETt] , <exp> [SETia] , ...
```

To complete the syntax, the generator outputs the (constant) productions for statement lists, label and parameter definitions, machine-independent pseudo-ops and expressions. The syntax and semantics for these constructs are defined in appendix A.

5.2 Code Generation Semantics

The organization of the code generation phase of the assembler was given in section 2.4.3 and is reproduced here.



In this section, the modules which construct and produce output will be described; this includes the CODE module, which constructs binary representations of formats from symbolic ones, and the output module LOADER, which organizes the binary information in a form that can be loaded into the target machine. The semantic routines for processing instructions are described first; the techniques are then extended to arbitrary data structures.

5.2.1 Instruction Semantics

The syntax for instruction formats was constructed to include three basic functions: a prolog routine, common to all instructions, invoked at op-code time, a routine to process each field (or form) of the format, and an epilog routine, also common to all instructions, called when all components of the format have been processed. The basic structure of each of these subroutines is outlined below; "I" represents the instruction variable, "DOT" is the location counter, "DEL" the instruction length, and "PASS" is true if the assembler is in pass-2 state.

```
subroutine PROLOG() is
  (length(format) -> DEL;
   PASS => list(DOT));

subroutine OPCODE() is
   PASS => op-code|I -> I;

subroutine SETf() is
   PASS => field -> I;
```

```
subroutine EPILOG() is
  (PASS =>
     (list(I[0:DEL-1]);
      load(I[0:DEL-1]);
      0 -> I[0:DEL-1]);
   DEL+DOT -> DOT)
```

The field semantic routines, including the OPCODE subroutine, are responsible for constructing the binary representation of the instruction from its components. The prolog and epilog routines maintain the value of the location counter and output the final result. All of these actions are completely target machine dependent; these dependencies are now investigated.

5.2.1.1 Data Structure

The first task is to derive the data structure used to represent an instruction. In general, it is a (bit-string) vector, but the size and number of elements depend both on the length properties of the formats and the addressing scheme of the target machine. Unfortunately, the structure declared in the ISP description need not reflect either of these properties. For example, the IBM-360 [IBM70] instruction register might be declared as I<0:47>, or I[0:2]<0:15>, or I[0:5]<0:7>. The second structure is the appropriate one, since all instructions are composed of one, two, or three 16-bit halfwords.

More formally, the unit of allocation (the instruction-size) is defined to be the greatest common divisor of the format lengths.* Actually, any common divisor could be used (including a single bit). But for efficiency reasons, it is desirable to use the largest possible unit, thereby increasing the amount of information that can be processed in a single operation. Instruction lengths are measured in terms of this unit, so they are guaranteed to be integer values.

5.2.1.2 Location Counter

The instruction-size must now be reconciled with the bytesize and the addressing scheme of the machine. (In section 4.4, a byte was defined to be the smallest addressable unit of information.) The location counter is incremented in bytesize units, which may not agree with the instruction-size. There are three cases, corresponding to the three addressing schemes noted in chapter 4.

1). instruction-size = bytesize: This is the normal case, found in most word-organized machines. Since instructions are allocated in bytes, no conversions are necessary between instruction lengths and the location counter unit.

* The length calculation is based on the amount of information fetched during construction of the format, rather than on its fields. Formats can have "holes" corresponding to unused bits.

2). instruction-size > bytesize: The machine is "byte-addressed" (the IBM-360 [IBM70] and the PDP-11 [DEC71] are examples).* Usually the instruction size is an integer multiple of the bytesize, and on binary machines, the multiple is a power of two. Instruction and data lengths must be converted to bytes before they are used in address calculations. Alignment of instructions and data (to even byte addresses, for example) may be required.

3). instruction-size < bytesize: Instructions are "packed", several to a word (for example, the CDC-6600 [CDC67a]). The bytesize is usually an integer multiple of the instruction-size. Since the location counter must be an integer value, an auxiliary counter (called the position counter) keeps track of the bits remaining in the current byte. Label symbols, as well as most data items, must be aligned on byte boundaries.

A major variation in these schemes is the requirement for a bit position counter in the last case. Actually, all versions of the assembler contain a position counter, since it is useful for assembling packed data structures with components of arbitrary size. The above cases do serve to identify optimizations that should be applied to the code

* Actually, a machine is byte-addressed if any common (single precision) information-unit is more than one byte long. So this condition is sufficient, but not necessary.

being produced. While these would constitute global optimizations at compile time, they are much easier to recognize at program generation time. Even optimization across procedures is feasible.

5.2.1.3 Alignmemt

The concept of alignment is a general one, which can be useful no matter what kind of addressing scheme is employed in the target machine. For example, the storage allocation software may require that all program and data segments begin on boundaries determined by their size. So two pseudo-ops are always provided to align the position and location counters. The problem here is to determine when alignment is required, what specific alignment rules must be followed, and what action should be taken when the rules are violated.

5.2.1.3.1 Data Alignment

The assembler requires that all alignment of data items be done explicitly with the pseudo-ops; no automatic alignment is performed, except to addressable boundaries.* There are two reasons for this approach. First, the proper alignment of a data item may depend on its positicn in an encompassing data structure (a control block, for example), rather than on the type of the data. This is a global

* A "variable-field data" pseudo-op [CDC67b] provides a method of overriding the alignment of the position counter as well.

consideration which is beyond the scope of the assembler. Second, the declaration of the data may not reflect the way it is referenced; a character string might be allocated in bytes, but processed as a word vector.

This suggests that the proper time to check data alignment is when the data is referenced, that is, when its address appears as an operand in some instruction. After all, that is when the hardware performs the check. In fact, because operand alignment checks are predicates on instruction fields, they will appear as conditions on the field in the format environment; these conditions can be verified by the semantic routine that processes the field. However, the format constructor must be careful not to eliminate these conditions when it is combining equal formats.

5.2.1.3.2 Instruction Alignment

The same principles apply to instruction alignment. Instructions are referenced when they are fetched by the machine's interpreter. This implies that any alignment requirements must appear as conditions on the program counter in the op-code environment of the instruction. Clearly, these conditions must be identical for all instructions of the same format, since alignment depends on the length of an item.

These observations imply that the prolog routine can check the alignment conditions based on the value of the location counter and the length of the current format. The only remaining problem is to determine what action should be taken when the conditions are not met. The general rule is to output "nop" instructions until the location counter satisfies the alignment predicate. But an instruction of the required length may not exist; in the IBM-360, for example, instructions must be aligned on even byte addresses, but there are no one-byte instructions. In this case an error message is generated, and nulls are supplied.

5.2.1.4 Field and Form Semantics

The OPCODE routine is called by the op-code machine responsible for combining micro-instructions; it "or"s the value of each mnemonic into the instruction, checking first that all of the selected bits are zero. To simplify this routine, all op-codes are assumed to have uniform structure and length, even though op-code fields may differ among formats. This is accomplished by representing each op-code value as if it were a complete instruction in itself, with operands unspecified (zero by convention). Finally, if there are no micro-coded instructions, this routine is merged with the prolog semantics.

For operands, the syntax generator created two distinct types of field semantic routines. If a bound conditional field is represented by syntax, the value of the field is known at generation time, and the code produced is:

```
subroutine SETfn() is PASS => n -> f;
```

Of course, the field reference must be mapped into an equivalent access on the instruction vector "I". In general, a field may cross addressable or allocation boundaries, and several assignment statements may be required.

For operands represented by an expression, the semantic routine obtains the value of the field from the interpreter stack. Since the value is computed at runtime, an error check must be included to ensure that no significant bits are lost when it is truncated to the field size; the specifics depend on the properties of the field. Additional error checking is generated if the field has an unbound condition attached, such as "f ne Ø" (see section 5.1.2.1).

```
subroutine SETf() is
  (POP() -> VAL;
   PASS =>
     (check(VAL);
      VAL -> f));
```

If the operand represents a form, the inverse of the form is computed from the stack value and assigned to the field. For example, if d<Ø:7> is an instruction field and the form is:

```
rel<Ø:15> := PC+2*sign_extend(d)
```

then the code generated is:

```
subroutine SETrel() is
  (POP() -> VAL;
   PASS =>
     (VAL-(DOT+PC) -> VAL;
      VAL\2 => error;
      VAL/2 -> VAL;
      trunc(VAL) => error;
      VAL<8:15> -> d));
```

Additional error checks are generated based on the operators used to define the form; the general rule is that no information should be lost. The inverse of "sign_extend", for example, checks that all truncated bits are equal to the sign bit. The generator also makes use of the value of the PC recorded by the format analyzer when the form was referenced (it was required to be a constant).

5.2.1.5 Binary Output

The final step is to produce a binary image of the instructions for subsequent loading into the target machine. The keyword here is loading. The output of an assembler is not just a continuous stream of data bits; it must include information about the location at which the data is to be loaded. This observation suggests that the output should be structured according to the bytesize of the machine, as blocks of contiguous data bytes.*

* Historically there are additional reasons for this organization. To insure accuracy of the loading process, the program is split into blocks so that a checksum can be associated with each block [Barron69].

The structure of the load file usually depends on software design considerations not entirely determined by either the assembler or the hardware; the format must be compatible with the output of other processors that produce executable code (compilers, linking loaders, and debuggers, for example). These processors require more complex information in the load file, such as relocation information, global symbol definitions, and "fixups". The following structure provides for compatability with other software systems:*

filler	any number of null (zero) bytes
block-id	unique number identifying the block type
count	number of bytes in the block
origin	load address for the first data byte
data	first data byte
.	
.	
data	last data byte
checksum	one-byte checksum

Every block is assumed to contain a block-id, count, and checksum. The assembler produced by the generator outputs only one block type: absolute binary code. Other block types can be defined (possibly with different internal structure) to satisfy the needs of other software systems, without affecting the assembler implementation.

* This structure was inspired by the PDP-1Ø [DEC73] and PDP-11 [DEC7Ø] relocatable binary formats.

The details of the block structure are determined by the relation between the bytesize and the address-size. It may be possible to pack the block-id, count, and origin fields into a single byte, for example; or several bytes may be required for each field. The details need not concern us here. From the code generator's point of view, a subroutine is provided (BYTLOD) which accepts a single byte, together with its address, and handles all of the formatting details.

Unfortunately, the code generation routines may not produce information in bytesize units; the instruction-size, for example, can be either larger or smaller than the bytesize. In general, data items can be of any size, ranging from a single bit to a variable length character string.

The solution to this problem is to view the path between the code generation semantics and the load file as a pair of pipes joined together by a filter.* The filter is constructed so that the in-coming pipe may be of arbitrary width, but the out-going pipe is fixed at the bytesize. The problem is then a simple case of buffering and packing (or unpacking) the data, were it not for the fact that addresses must also be generated and passed through the out-going

* The terms "pipe" and "filter" are as described in UNIX [Ritchie74], although here they apply to modules rather than processes.

pipe. For this reason, both the position and location counters are controlled by the filter; they are updated as a side-effect when information is passed through the pipes. No other routine may modify the counters, without first ensuring that the two pipes are "synchronized", that is, aligned on a byte boundary.

This completes the developement of the prolog, field, and epilog semantics for processing instructions. The assembler must also provide pseudo-ops for producing data; these routines are described below.

5.2.2 Data Semantics

This section will demonstrate that the techniques used to process instructions are sufficient to handle data formats as well. Instructions are just one example of data structures, so this should be expected. Consider the declaration of the PDP-10 block transfer word [DEC73]:

```
xwd/block_transfer_word<0:35>
   src/source<0:17>      := xwd<0:17>
   dst/destination<0:17> := xwd<18:35>
```

A format can be constructed directly from the declaration; it is processed exactly like an instruction format, except that it has no op-code component. In this case, the syntax (ignoring prolog and epilog) is:

```
<stm> ::=  .xwd <exp> [SETsrc] , <exp> [SETdst];
```

The semantic routines required for the fields are obvious.

The notion of forms also applies to data formats, when a field is defined functionally, rather than as an equivalence. The input-output word first appearing in section 2.3 would be defined as follows:

```
iowd/input_output_word<0:35>
   cnt/count<0:17>    := -iowd<0:17>
   adr/address<0:17> := iowd<18:35>+1
```

The syntax is the same as for ".xwd" (with the names changed), but the semantic routines use the inverse of the forms to calculate the value of the fields.

Finally, for simple data-types with only one component, the processing can be consolidated into a single routine. For example, if the ISP contains a definition of "byte<0:7>" in the Data-Type section, a pseudo-op is invented with the following syntax:

```
<stm> ::= .byte <exp> [BYTE];
```

By merging the prolog, epilog, and field semantics (and given the filter defined in the previous section), the code generated is:

```
subroutine BYTE() is
  (align(BYTESIZE); POP() -> VAL;
   filter(8,VAL); PASS => list(VAL))
```

Note that the BYTESIZE used to align the position counter is determined by the addressing scheme and may not bear any relation to the "byte" declared by the user.

The term "data-type" has been used loosely here to mean a data structure with a single component. This is only part of the definition. A data type (according to ISP) "specifies the encoding of a meaning into an information medium" [Bell71, p. 629]; it includes a referent (character, integer, real number, ...), a carrier (byte, halfword, word, ...), a format (a list of components), and an encoding or representation scheme (one's complement binary, ten's complement decimal, sign-magnitude, ...). The discussion above assumes that any data type can be constructed from a single basic unit: the uninterpreted bit-string. For non-trivial data-types, this may lead to awkward representations. For example, a floating point number might be defined as follows:

```
float<Ø:35>
   s/sign            := float<Ø>
   e/exponent<Ø:7>   := float<1:8>
   f/fraction<Ø:26> := float<9:35>
```

The generator could represent this format as an optional sign followed by two (unrelated) expressions. This representation is not entirely unusable, but it is far short of convenient. The problem is that the declaration defines only the data carrier and format; it does not provide any information about the referent (a real number) or the encoding scheme (two's complement normalized binary fraction, and a one's complement excess-128 binary exponent, for example). ISP provides no mechanism for defining new

data-types (and the legal operations on them), so the system must struggle along with limited pre-defined notions of a few encoding schemes.* It is important to determine how this problem affects the assembler implementation, and why it is at the root of the machine-independent coding problem.

5.3 Machine Dependence

The semantic routines developed in the previous section were coded in a variant of ISP, augmented by additional control structures such as subroutine calls, WHILE-loops, and the like. This represents a programming language (which we will call IMP, to avoid any confusion); it was used primarily to avoid having to introduce another notation. Of course a machine that executes IMP does not exist, so the code must be compiled into some executable form that runs on some specific machine.

A complete solution of the assembler generation problem would produce an assembler written in the machine language that executes on the computer defined by the ISP description. This is a difficult task. At the very least, it presumes the existence of an IMP compiler that generates code for the target machine. But because the machine is not fixed, both the IMP language and the code generator for that language must be modified to reflect the properties of the

* In [Bell71], data-types are defined in an entirely different notation (PMS?). For other views on defining data-types and representations, see [Flon74] or [Iverson62].

target computer. Although ISP (and therefore IMP) contains a wide variety of data-types and operations, the target machine is likely to implement only a few of these with instructions. And the machine may include single instructions that combine several operations for efficiency reasons. The program generators must take these facts into account.

This means that the module generators described in the above section must produce programs in a variant of IMP which has been tailored to the new computer. An additional subsystem is required which determines exactly what that version of IMP should contain. And the generators require additional functions which map the high-level (but machine specific) IMP constructs into instruction sequences that execute on the target machine. These functions might be distributed throughout the system in each individual program generator, or centralized in a single module usually called a code generator.

Producing a code generator for any high-level language from a machine description is clearly orders of magnitude more difficult than producing an assembler. This is because the assembler does not need to "understand" the semantics of an instruction in order to produce an encoded representation of it from a symbolic one. A code generator, on the other hand, is primarily concerned with the effects of an instruction (what it does), and the representation is

incidental. The construction of machine-specific code generators has been the subject of previous research [Miller71, Weingart73, Donegan73], and work is currently going on in the field [Barbacci74, Newcomer]. But that problem is beyond the scope of this thesis.

The approach used here is to generate a cross-assembler, running on some (fixed) host machine, that produces code for the target machine. This has the advantage that all of the programming systems of the host environment are available for use in developing the system; in particular, the mechanics of code generation can be ignored by implementing the cross-assembler in some fixed high-level language.* This approach does not eliminate all of the problems associated with machine-dependence, but it does allow the module generators to concentrate on only those issues that are involved in the assembly process. As we shall see, these issues revolve around the representation and accessing of information. Since the output of the assembler must be in a form acceptable to the target machine, some properties of the target computer must be simulated on the host machine.

* The language used is IMP10 [Irons70, Bilofsky73]. By coincidence, it is very similar in notation and power to ISP. This is convenient, but certainly not required; any systems programming language could be used.

First, the instructions and data processed by the assembler must be allocated in terms of the basic information-units (i-units) of the target machine, and accessed according to its addressing scheme. This amounts to defining equivalences (bit mappings) between the primary memories of the host and target computers, and constructing conversion rules for addresses based on the bytesize of those memories. This is exactly the problem considered in section 5.2.1, where the data structure used for allocating instructions was determined, and the behavior of the location counter was defined in terms of that structure. The same techniques are used in determining the mapping between host and target memories.

Second, the representation of the target machine's data-types must be simulated on the host; this includes all properties of the data-type: referent, carrier, format and encoding. The referents are pre-defined, and built into the system: boolean, digit, character, integer, and real number. The carrier is an i-unit that determines the allocation and addressing of the data-type, as described above. In ISP, the methods of encoding values in bit-strings are assumed to be understood in terms of current machine architectures, and the operations on a data-type are assumed to be defined by its representation. For example, the following declaration defines a floating point data-type:

```
float<Ø:35> {referent: real; value: s.f*(2^e)}
   s/sign            := float<Ø>       {two's complement}
   e/exponent<Ø:7>   := float<1:8>+128 {one's complement}
   f/fraction<Ø:26>  := float<9:35>    {two's complement}
```

As a result of this declaration, the following operation is assumed to be well-defined:

```
AC-abs(M[e]) -> AC,MQ {unnormalized float}
```

Understanding this definition requires a considerable amount of world knowledge about the implementation of floating point arithmetic on current generation binary computers.

The assembler generator faces two problems when a new data-type is defined. First, it must define a pseudo-op that serves to create instances of that data type from its components, and generate semantics to transform an external (character-string) representation of the data-type into its internal form. These problems were addressed in section 5.2.2. Second, if the data-type can appear in assembly-time expressions, the interpreter must be modified to process the data-type, and conversion rules must be established between it and all other data-types.* In addition, if the data-type is used to represent addresses, then computations on

* In most assemblers, expressions are always computed as integers of some fixed maximum precision (determined by the maximum field size of the machine, usually called a word); any conversions are performed by the field semantic routine that processes the value. Hence the INTRPT module need be modified only if the integer data-type does not use the default encoding (two's complement binary).

addresses, and in particular on the position and location counters, must be implemented by calls on the interpreter. Even the seemingly innocent statement "1+DOT -> DOT" must be replaced by:

```
push(number('1')); push(DOT);
add(); pop() -> DOT;
```

Of course, the location counter is just a special case. Any variable whose value appears as output of the assembler must be handled in this way. Thus all operations carried out by the field semantic routines (including error checking) must be performed by the interpreter.

The key point here is that the assembly process is understood well enough so that the generator is able to anticipate all the ramifications of a perturbation in the data-types. Furthermore, the organization of the assembler is such that the required changes are localized to specific modules and can be made without disturbing other processing. In short, the generator knows when a change is necessary and where it should be made. However, unless all the data-types (and operations on them) are built into the system, the generator does not know how to effect the necessary changes. This marks the limit of the system's understanding.

6. CONCLUSIONS

6.1 The Implementation

The reader should now be convinced that a program actually exists which produces assemblers for a large class of computers; the appendices contain a number of examples. The program does not implement all of the features described in the text, and therefore it represents an existence proof, rather than a production piece of software. But it does contain enough features to produce usable assemblers; the system has been particularly helpful in hardware courses and microprogramming environments, where the instruction set is a moving target.

The generator is not especially large, and it is surprisingly fast. Initially it occupies about 18K 36-bit words, expanding as necessary to accommodate the ISP description of the target machine, reaching about 34K for a mid-sized computer like the IBM-1800 (appendix E). The running time is governed by user response during the interactive syntax dialog. Assuming an active response rate (with no pauses for meditation between answers), the total elapsed time to produce an assembler for this machine is less than two minutes. (More accurate timing measurements were not considered necessary.) Processing is about equally divided between ISP compilation, machine analysis, and program generation. Because of its speed, it is possible to use the system interactively to check the ISP description before the assembler is actually produced.

It is somewhat disappointing that the assembler is the only program generated by the system. An obvious extension would be to also produce a linking loader, and modify the assembler to generate relocatable output. This would be desirable in order to allow an incremental and orderly transition from assembly code to a higher-level language. Since the linker requires considerably less machine-dependent knowledge than an assembler, this would not be a difficult task. In fact, the analyzer as it stands produces all of the required parameters involving address fields, and several of the program modules could be used with minor modifications (the symbol table routine, for example). The most difficult (and most rewarding) task is in defining the linking process axiomatically and developing an implementation model.

6.2 Other Contributions

One contribution of this research is a precise yet flexible model of assembly, both in terms of what assemblers do (the axiomatic definition) and how they do it (the implementation model). This is a necessary prerequisite for the rest of the development. But it also represents a major limitation; the system has a preconceived notion of the

assembly process that is not easily modified. A better approach might be to include a dialog with the user that determines what kind of assembler he would like to have generated. The user might specify one or two passes, or relocatable or absolute output, for example. All of these variations are independent of the target machine, so they were not considered here. But it is important to realize that an assembler is actually a family of programs, and we have displayed only a few members of the community.

Another important contribution is in showing that the use of a formal description of a computer by an automatic programming system is a feasible approach. The fact that the assembler generator obtains all of the required machine-dependent knowledge from the machine description demonstrates this. It is clearly not appropriate (or even desirable) to expect every automatic programming system to accept a machine description as one of its inputs. But for systems programs -- those that serve as the interface between the user and the computer system -- a thorough understanding of the machine is obviously required, and the machine description approach has several advantages:

First, the machine description is precise. Although a description can be quite large, it is a great improvement over its English counterpart, and it does not suffer from the inherent ambiguities of a natural language description.

Second, the machine description by default contains all of the machine-dependent information that the implementor requires. While this information may be difficult to extract, the number of errors customarily made by human programmers in both the reading and writing of machine reference manuals is substantially reduced.

Third, this approach leads to general characterizations of the machine space. Because ISP descriptions are reasonably compact compared to machine manuals, it is possible to examine a large number of them. And because ISP is a formal language, much of the examination can be performed by the machines themselves. As more machines are inspected, constants are replaced by variables and special cases by general concepts.

Finally, if a sufficiently powerful machine analyzer can be built -- one that is capable of understanding the semantics of instructions and data-types as well as their structure -- then a systems programmer can depend on the analyzer to handle the machine-dependent details, while he concentrates on the user side of the man-machine interface. This is clearly where the emphasis belongs.

6.3 Further Research

We have demonstrated that a formal machine description can serve as an almost complete specification of the machine-dependent aspects of at least one systems

programming problem. The obvious questions to ask are 1) what must be added to machine description languages to make them more useful from the software point of view, and 2) how can this approach be extended to other automatic programming problems.

6.3.1 Backtracking

By far the most troublesome aspect of ISP (and therefore also of the assembler generator) is the scheme for defining data-types (see sections 5.2.2 and 5.3). ISP defines the representation of values in bit-strings, and the operations that apply to those representations, by an enumeration of existing schemes.* For example, the terms "two´s complement" and "sign-magnitude" are keywords in ISP, and they are assumed to be well understood.

There are two difficulties with this approach. First, it is impossible to introduce new data-encoding schemes into the language. While the set of keywords is adequate for current schemes, it may not be sufficient in the future. Second, the keywords themselves are not precise enough to completely define the representation. The expression "X[0:15]<0:3> {sign-magnitude}" does not provide enough information to determine the bit pattern of a value or to perform an addition of two such values. We need a more

* In most other hardware description languages, the operations are defined in terms of the hardware itself (the primitive operations "nand" and "nor"), and the question of data-type (referent) never arises.

specific definition mechanism, but something considerably less detailed than the circuit diagram of the arithmetic unit.

If we extend ISP to include some of the APL operators [Iverson62], the following might serve as a (parameterized) definition for the two's complement encoding of an integer in a bit-string (assuming zero-origin indexing):

```
                fixed<Ø:N-1> {ref:int} :=
                  (int≥Ø => (Nρ2)⊤|int;
                   int<Ø => ¨(Nρ2)⊤|int+1)
```

Here "int" is assumed to be a keyword, built into the language, standing for the abstract data-type "integer", the referent of the (concrete) data-type "fixed". The definition shows how to encode an integer value in an N-bit register.

On a six-bit character machine, a sign-magnitude encoding might be defined as follows:

```
    decimal[Ø:N]<Ø:5> {ref:int}
       sign[Ø]<Ø:5> := ´+-´[int<Ø] {bcd}
       mag[1:N]<Ø:5> := ´Ø123456789´[(Nρ1Ø)⊤|int] {bcd}
```

This definition assumes that some more primitive encoding (bcd<Ø:5>) has been defined for characters.

In contrast to programming languages, the data-types in ISP appear as operation modifiers, rather than as part of the declaration of a "variable" (register or memory). This is consistent with the hardware view that a bit-string remains uninterpreted until an operation is performed on it. So if "X<Ø:15>" is a register, then "X+1 -> X {fixed}" is shorthand for

```
      X{fixed<Ø:15>}+1{fixed<Ø:15>} -> X{fixed<Ø:15>}
```

ISP does not define the rules for distributing a modifier over an expression; it is often unclear exactly how this should be done. For example, the assignment operation in the statement above does not depend on the data-type; but contrast this with "X{fixed} -> X{float}".

These definitions by themselves do not solve all of our problems. We need to be able to decode values as well as encode them; this is very much like inverting an instruction form (section 4.3.2). We are also assuming that all of the ISP operators can be understood in terms of APL operations and the encoding scheme. APL is being used as a meta-language, and it is not clear that it is an appropriate one. It may be better to define all the ISP operators on a data-type in terms of some more primitive operators and data-types. The examples here are intended only to illustrate a possible approach to the problem.

Some of these problems are addressed by the current research on abstract data-types in higher level languages [Wegbreit74, Wulf74, Liskov74, Flon74]. The context of this research is different, of course; the primitive types of a programming language (booleans, characters, integers and reals) correspond to abstractions of the hardware's primitive type (bit). But both areas deal with abstractions of more primitive concepts, so the same principles should apply.

This discussion is obviously not an exhaustive analysis of the ISP notation; further difficulties are pointed out in appendix B. But from the point of view of generating software for the machine, the definition of data-types is of critical importance in every automatic programming system, and it merits further study.

6.3.2 Look-ahead

To examine extensions of this research to other problems, we will proceed by wishful thinking. Suppose that a machine understanding system -- an extension of the analyzer described in chapter 4 -- were available. Besides the obvious extension to linking loaders described above, the following applications come to mind:

1). A program that finds a simulation of the BCPL INTCODE pseudo-machine [Richards74].

2). A system that generates LISP interpreters written in machine language [McCarthy6Ø].

3). A program that produces locally-optimizing code generators for a fixed class of (ALGOL-like) languages [Newcomer].

4). A program that generates single-user operating systems.

Note that none of these applications (with the probable exception of the operating system) require significant extensions to the machine-description language (other than the data-type definition facilities described above). In particular, we need to add timing information for use in determining optimal code sequences; but this is easily done. The major difficulties are in formally defining each of these problems, and in extracting the required machine-dependent information from the computer description. Each problem requires at least the information derived by the assembler generator, so the analyzer can be used as a base to build from.

The principal extension required is that each of these applications needs an understanding of the semantics of the machine's instructions and data-types, whereas the assembler generator could ignore all but the structure and the field semantics (addressing operations, primarily). It is no small step to add this kind of understanding. To allow gradual development of the analyzer, I would begin with the LISP interpreter, for the following reasons: 1) the definition of the problem can be stated in LISP itself, 2) the implementation model is well established and there are numerous examples, and 3) the number of implementation decisions and corresponding optimizations is reasonably small. This problem is also flexible; one can begin by implementing only pure LISP, and then building the rest of the language on top of it.

The next step is a LISP compiler, with emphasis on efficiency of the generated code. This could be used as the ground work for a code generator generator, designed to span procedural languages of the ALGOL variety. Finally, this system should be integrated into a compiler-compiler environment. Thus a machine description becomes another input to a translator-writing system, a goal of long standing [Feldman67], but one which has been sparsely researched [Miller71, Weingart73].

Regardless of the application, it is clear that any system that reduces the complexity of dealing with the machine at the instruction-set level is of significant value in the development of production software and programming tools. Attacking these problems by way of a symbolic machine description serves to guarantee the generality of our solutions across a large machine space. The assembler generator demonstrates the feasability of this approach; further research is needed to verify its usefulness in more complex applications.

> "Writing this sort of report is like building a big software system. When you´ve done one you think you know all the answers and when you start another you realize you don´t even know all the questions."
> Brian Randell [NATO7Ø]

APPENDIX A
THE BASE ASSEMBLER

The general flavor of the assembly language and the organization of the implementation were described in chapter 2. This appendix serves to present the syntax and semantics of the base language in detail. The machine dependent constructions for instruction and data formats are not described fully, as they are adequately dealt with in chapter 5, and in the examples in appendices following. When it is necessary to assume some particular computer to illustrate a point, the AJP-21a machine described in chapter 3 (and appendix C) will be used.

The development proceeds in a bottom-up fashion, beginning with the basic symbols of the language, and expanding to expressions, statements, and programs. The remainder of the discussion involves the pseudo-ops of the assembler and their effects on the internal state variables of the assembly process, notably the location counter. We conclude the appendix with a simple example.

A.1 Lexical Conventions

A.1.1 Alphabet

To allow for variations in the character set, symbols of the language are defined in terms of character types rather than the characters themselves. Types required by the assembler include digits, letters, marks (special characters), and additional types used to distinguish symbols. For the ASCII character set, the type definitions are as follows (non-printing characters are represented by their numeric equivalents):

```
   mrk ::= ´!´ | ´"´ | ´#´ | ´$´ | ´%´ | ´&´ | ´(´ | ´)´ |
           ´*´ | ´+´ | ´,´ | ´-´ | ´/´ | ´:´ | ´;´ | ´<´ |
           ´=´ | ´>´ | ´@´ | ´[´ | ´\´ | ´]´ | ´^´ | ´_´ |
           ´`´ | ´{´ | ´|´ | ´}´ | ´~´ ;

   let ::= ´.´ | ´A´ | ´B´ | ... | ´Z´ | ´a´ | ... | ´z´ ;

   dig ::= ´Ø´ | ´1´ | ´2´ | ´3´ | ´4´ | ´5´ | ... | ´9´ ;

   blk ::= ´ ´;         qot ::= ´´´´;         qtm ::= ´?´;
   eol ::= 12b;         eof ::= 177b;
```

Not all of the mark characters are actually used in the language. They are included to allow flexibility in the definition of machine-dependent syntax, without requiring modification of the lexical analyzer.

A.1.2 Symbols

Symbols are of four varieties: operators, names, numbers, and strings. Comments may also be included in programs; they are ignored. The following regular grammar defines operators, names, and numbers.

```
        <opr> ::= mrk | eol | eof ;

        <nam> ::= let | <nam> let | <nam> dig ;

        <num> ::= dig | <num> let | <num> dig ;
```

Names are constructed in the usual way from letters and digits (note that a period is a letter, so the "compound" names TOK.TYP and TOK.LEN are valid). The symbols "." and ".." have special significance, described below. Numbers can contain letters as well as digits (they must begin with a digit, of course). This is to allow bases greater than ten, as in hexidecimal notation.

Strings (literal text) are distinguished from other symbols by enclosing them in single quote marks: ´Foo´. A quote itself is represented by two quotes, as in ´Don´´t´. The null string (value zero) is denoted by ´´.

```
    <stg> ::= <str> qot | <str> ;

    <str> ::= qot | <str> blk | <str> mrk |
        <str> let | <str> dig | <stg> qot | <str> qtm ;
```

The special characters "eol" and "eof" are not allowed in strings; in the absence of a closing quote mark, they serve as string terminators (but are not part of the string).

Comments begin with a question mark, and can contain any character except "eol" and "eof". Comments are completely ignored in the rest of the processing (except that they appear in the output listing).

```
    <cmt> ::= qtm | <cmt> ign | <cmt> blk | <cmt> mrk |
        <cmt> let | <cmt> dig | <cmt> qot | <cmt> qtm ;
```

In the syntax given below, the notation ".nam" will be used to denote any symbol of type <nam>; similarly ".num", ".stg", and ".opr" correspond to numbers, strings, and operators.

A.2 Expressions

Expressions are formed from basic symbols, unary and binary operators, and parentheses. As a general rule, an expression is allowed wherever a symbol may appear, with the obvious exception of the label and op-code fields. All expressions are computed as full-word quantities (however long a word happens to be), and truncated if necessary by the semantic routines that process the value.

A.2.1 Values

The value of a symbol depends on its type. Names are used to represent labels, parameters, or op-codes (see section A.3). Numbers and strings represent constants.

```
   <atm> ::= .nam [NAME] | .num [NUMBER] | .stg [STRING];
```

If a number contains a period, it is assumed to be decimal; otherwise it is interpreted according to the binary radix derived for the particular machine (usually octal or hexadecimal). No other significance is attached to the period, so the numbers "5.Ø5" and "5Ø5." both represent the same bit pattern.

The value of a string is the integer equivalent of the binary code for the first (left-most) character. Since ISP provides no information about the character set or the method of packing characters, strings appearing in expressions are restricted to a single character. Pseudo-ops are used to enter longer strings as data (see A.5.2).

A.2.2 Unary Operators

Unary (monadic) operators include complement (~), absolute value (|), negation (-), and the null operator (+). Multiple unary operators are evaluated from right to left, so that "|~X" means "|(~X)". Parentheses are used for grouping in the usual way.

```
      <atm> ::= ´~´ <atm> [NOT] | ´|´ <atm> [ABV] |
                ´-´ <atm> [NEG] | ´+´ <atm> |
                ´(´ <exp> ´)´ ;
```

A.2.3 Binary Operators

Binary (dyadic) operators include the usual arithmetic operations (addition, subtraction, multiplication, division, remainder) and logical operations (and, inclusive-or, equivalence, exclusive-or). Logical shift (~) shifts left or right depending on whether the second operand is positive or negative.

```
   <exp> ::= <atm> |
      <exp> ´+´ <atm> [ADD] | <exp> ´-´ <atm> [SUB] |
      <exp> ´*´ <atm> [MUL] | <exp> ´/´ <atm> [DIV] |
      <exp> ´\´ <atm> [REM] | <exp> ´&´ <atm> [AND] |
      <exp> ´|´ <atm> [IOR] | <exp> ´=´ <atm> [EQV] |
      <exp> ´#´ <atm> [XOR] | <exp> ´~´ <atm> [SHF] ;
```

Note that all binary operators have equal precedence; in the absence of parentheses, evaluation proceeds from left to right.

A.3 Statements

Statements are used to produce code and data, or to direct the internals of the assembly process. Usually statements contain an optional label, an op-code or pseudo-op, and a list of operands, separated by commas. The pseudo-ops are described in the section A.4; labels, parameters, and op-codes are described below.

A.3.1 Labels

Every statement, regardless of its type, may be labeled. The value of the label is set equal to the current value of the location counter. The position counter is advanced to an addressable boundary, if necessary, before the label is defined.

```
<stm> ::= [HASH] .nam [MNEMON] ´:´ [LABEL] |
          [HASH] .nam [MNEMON] ´:´ [LABEL] <stm> ;
```

Note that a statement may be null, containing only a label definition, and that any number of labels may appear. Of course, a label can only be defined once; if multiple definitions appear, only the first is used.

It is important to note that the processing of a label is carried out independently of the statement in which it appears. In particular, if the statement represents an

instruction which requires alignment of the location counter, the location of the instruction may differ from the value of the label (see also section A.3.3 and A.5.1).

A.3.2 Parameters

Symbols that are given a value by a direct assignment statement, rather than by copying the location counter value, are called parameters. The form of an assignment statement is as follows:

```
<stm> ::= [HASH] .nam [MNEMON] ´=´ <exp> [EQUATE];
```

Unlike labels, the value of a parameter is not necessarily constant throughout the program; it may be changed at any time by an assignment statement ("X=X+1", for example). This leads to the restriction that the value of the right-hand side must be defined in both pass one and pass two; the expression may not contain forward references.

Two special parameters are predefined by the assembler: the location counter ".", which counts upward from zero in bytes, and the position counter "..", which counts downward toward zero in bits. These parameters are automatically updated by the assembler. If they are to be modified explicitly, pseudo-ops are used, rather than direct assignment statements (see A.5.1).

A.3.3 Mnemonics

All remaining statements begin with a mnemonic, followed by a list of operands. Mnemonics are either machine op-codes as they appear in the ISP description, or pseudo-ops invented by the generators. In any case, they are reserved keywords, and may not be used as labels or parameters. The format of the operands depends on the particular mnemonic; pseudo-ops are described individually in the next section. Op-code statements are necessarily machine dependent, and are fully described in chapter 5. The AJP-21a instruction formats might be represented using the following syntax (the machine-dependent semantic routines are not shown):

```
<stm> ::= [HASH] .fmt2Ø <exp> ;
<stm> ::= [HASH] .fmt2Ø ´*´ <exp> ;
<stm> ::= [HASH] .fmt2Ø ´@´ <exp> ;
<stm> ::= [HASH] .fmt2Ø ´@´ ´*´ <exp> ;

<stm> ::= [HASH] .fmt21;
```

This definition reflects the two types of instructions found on the machine. The first contains an op-code (.fmt2Ø), an indirect bit (@), an index bit (*), and an address (<exp>), while the second contains only an op-code field (.fmt21). The symbols ".fmt2Ø" and ".fmt21" actually stand for a set of op-codes, just as the symbol ".nam" represents any user-defined variable. All mnemonics, including pseudo-ops, are represented in this way, rather than using the mnemonics themselves. The HASH function

sorts out the meaning of the symbols according to their type. This allows the choice of a mnemonic to be determined by the user, without affecting the internal processing of the assembler (see A.5.4).

In addition to assembling and outputing an instruction, op-code statements cause the location and/or position counters to be updated, according to the length of the code. The actual updating does not take place until after all of the operands have been processed, so the values of the symbols "." and ".." always refer to the beginning of the current instruction (this rule does not apply to all pseudo-ops; see section A.5.2). Note that on some machines, both counters may require alignment before the instruction is processed.

A.4 Programs

A program is composed from a series of lines terminated by an end-of-file character (this character is supplied automatically by the lexical analyzer and is not entered as part of the program). Each line is terminated by an end-of-line character.

```
<prg> ::= <stl> eof [END];

<stl> ::=  eol       [LINE] |
    <stm>  eol       [LINE] |
    <stl>  eol       [LINE] |
    <stl> <stm> eol  [LINE] ;
```

Blank lines may be included in the program to indicate sectioning and improve readability. A special pseudo-op is not required for this purpose.

A.5 Pseudo-Ops

Pseudo-ops are so called because they do not produce machine code; they may, however, produce data. Those that produce no output, but merely change some internal variable of the assembler, are sometimes called assembler directives. As mentioned above, mnemonics in the syntax are represented using the dot notation (.org, for example); the actual (default) symbol is indicated in the text.

A.5.1 Location and Position Counter

This group of psuedo-ops serves to manipulate the values of the location (byte) counter and the position (bit) counter. They either set the value directly, or align the value on a boundary.

```
<stm> ::=
    [HASH] .org ``    [ORIGIN] |
    [HASH] .org <exp> [ORIGIN] |
    [HASH] .aln <exp> [ALIGN]  |
    [HASH] .pad <exp> [PAD]    |
    [HASH] .fup <exp> [FORCUP] ;
```

The ORG (.org) pseudo-op first aligns the position counter on a byte boundary. It then saves the current value of the location counter, and sets the LC to the value of the expression. If no expression appears, the last value saved is used, effectively swapping it with the current LC. This enables a program to bounce back and forth between two areas of memory, perhaps allocating instructions in one area and data in the other.

The ALIGN (.aln) pseudo-op first forces the position counter to an addressable boundary, and then increments the location counter (if necessary) until it is an integer multiple of the operand. On a byte-addressed machine like the IBM-360, for example, the statement "ALIGN 8." advances the LC to a double-word boundary. This operation does not disturb the data in the locations that are passed over.

The PAD (.pad) pseudo-op serves to align the position counter so that it is an integer multiple of the operand; unlike the ALIGN pseudo-op, bit alignment does produce data, setting all bypassed bits to zero. Note that the position counter counts downward from the bytesize, rather than upward toward it. Naturally the value of the operand cannot exceed the bytesize, and must be greater than zero.

The FORCUP (.fup) pseudo-op forces the position counter to its upper limit (bytesize) by padding the current byte with no-op instructions. Except for the use of no-ops, it is equivalent to the statement "PAD bytesize". This type of alignment is required on a machine that packs several instructions into a single byte (the CDC-6600, for example).

A.5.2 Entering Data

Two pseudo-ops are provided for generating data; a simple one is defined for the most frequently used data-type, and a more complex construction allows arbitrary data structures.

```
        <stm> ::= [HASH] .dat <exp> [DATA] |
                  [HASH] .vfd <vfd> ;

        <vfd> ::= ´<´ <exp> ´>´ [SIZE] <exp> [FIELD] |
            <vfd> ´<´ <exp> ´>´ [SIZE] <exp> [FIELD] |
            <vfd> ´,´ <exp> [FIELD];
```

The DATA (.dat) pseudo-op allocates a single word containing the value of the operand; the position counter is aligned to an addressable boundary before processing, if necessary. Note that a word may be larger than a byte (depending on the machine), but it is never smaller. It usually corresponds to the size of an accumulator.

The FIELD (.vfd) pseudo-op allocates lists of data items in fields of arbitrary bit-width. As the syntax indicates, the field size is enclosed in angle brackets, followed by a list of values for that field size. A size specification may appear anywhere in the list, and applies to all values up to the next size specification or the end of the list. Note that a field size always begins the list.

Unlike most statements, this pseudo-op pays absolutely no attention to addressable boundaries; if a value does not fit in the current byte, it is split so that the low-order bits appear in the high-order end of the next byte. The position and location counters are updated dynamically as the list is processed; hence they usually do not refer to

the beginning of the current line.

A.5.3 Miscellaneous

Two pseudo-ops are provided to control the format of the listing file produced by the assembler. A third pseudo is used to inform the loader of the starting address of the program.

```
        <stm> ::=
            [HASH] .pag         [PAGE]  |
            [HASH] .til .stg    [TITLE] |
            [HASH] .beg <exp>   [BEGIN] ;
```

The PAGE (.pag) pseudo-op causes the statements following it to appear on a new page in the assembly listing. The TITLE (.til) pseudo-op sets the title line for subsequent pages, replacing the current title. If a TITLE pseudo immediately follows a PAGE, the title will appear on the new page.

The BEGIN (.beg) pseudo-op is used to indicate the starting address of the program, which is passed on to the loader. If no BEGIN statement appears in the program, a start address of zero is assumed. The value of the operand is not checked against the allocation limits of program.

A.5.4 Bootstrapping

Two pseudo-ops are provided to assist in the bootstrapping process. One is used exclusively for machine op-codes; the other defines any arbitrary symbol. Numerous examples of these pseudo-ops are contained in the bootstrap file BOOT produced by the assembler generator.

```
  <stm> ::= [HASH] .def .stg [MNEMON]
                  '(' .nam [FORMAT] ')' '=' <exp> [OPDEF];

  <stm> ::= .stg [MNEMON] <exp> <exp> <exp> <exp> [BOOT];
```

The OPDEF (.def) pseudo-op is used to bootstrap op-codes or to extend (but not redefine) existing mnemonics. The first operand is the op-code itself, enclosed in quotes. The second operand contains the format name (.fmt27, for example), or the name of some other op-code with the proper format. The format determines the operand syntax and semantics. The last operand specifies the value of the new op-code.

The exact configuration of the value depends on the maximum op-code size of all the instructions of the machine. The bit length of the opcode field is this maximum, rounded upward to an integer number of bytes (say N bytes). These N bytes are always assigned to the first N bytes of the instruction (even if the instruction is actually shorter). So if the new op-code value is less than N bytes long, it should be left-justified (bytewise) in an N-byte field. An examination of the examples in the bootstrap file will make this clear.

The last pseudo-op serves as the bootstrapping operator; it is used to define the location and position counters, the pseudo-ops, and the instruction format names.

The symbol to be defined begins the statement, enclosed in quotes. The four operands represent the symbol type, format code, interpreter type, and the value. Each field is implementation-dependent, and is fully described in the semantic modules. Normally the only user modification involves changing the symbol, if some other pseudo-op names are preferred.

A.6 Example

The following sample program illustrates the basic features of the language. The machine is the AJP-21a, and the program is Knuth's maximum finder.

```
TITLE 'Algorithm 1.2.10-M  Find the maximum [Knuth68]'

? Given n elements A[1], A[2], ..., A[n], find m and j
? such that m = A[j] = max 0<k<n+1 A[k], and for which
? j is as large as possible.

?   1. [Initialize.]  Assign n -> j, n-1 -> k, A[n] -> m.
?
?   2. [All tested?]  If k=0, the algorithm terminates.
?
?   3. [Compare.]  Go to 5 unless A[k]>m.
?
?   4. [Change m.]  Assign k -> j, A[k] -> m.
?
?   5. [Decrease k.]  Decrease k by one, return to 2.

? Assumption: n and A[1:n] are already in memory.
? Comment: m := AC and k := X {index register}.
? Comment: A[1:n] is stored in Mp[A+1:A+n].

K=0      ? k is index register X := Mp[0]

         ORG  100
         BEGIN here
here:    LOD  @N
         STO  J          ? 1. n -> j
         STO  K          ?    n -> k
         LOD  @*A        ?    A[n] -> m

loop:    SIT  done       ? 2. k-1 -> k=0 => stop
         EQL  @*A
         JIF  loop       ? 3. m=A[k] => go to 2
         GTR  @*A
         JIF  loop       ?    m>A[k] => go to 2
         LOD  @K
         STO  J          ? 4. k -> j
         LOD  @*A        ?    A[k] -> m
         JMP  loop       ? 5. go to 2

done:    STO  M
         HLT

J:       DATA  0         ? local, output
M:       DATA  0         ? local, output
N: A:                    ? global, input
```

APPENDIX B

THE ISP' IMPLEMENTATION

Section 3.3 contains a general description of ISP and a simple example of a machine description. This appendix contains a more thorough definition of the subset of the language (ISP') used by the assembler generator. The description is presented in two major sections. The first defines lexemes: the terminal symbols of the language. The second defines the declarative and procedural constructs of ISP'. The development assumes a working knowledge of ISP as defined in [Bell71].

B.1 Lexical Conventions

B.1.1 Alphabet

For efficiency reasons, the lexical analyzer processes characters according to their type (letters, digits, etc.) rather than the actual character codes themselves; this also allows for greater machine independence. The first step is to partition the character set according to type; for the ASCII set, the definitions are as follows:

```
mrk ::= '!' | '"' | '#' | '$' | '%' | '&' | '(' | ')' |
        '*' | '+' | ',' | '.' | '/' | ';' | '<' | '?' |
        '@' | '[' | '\' | ']' | '^' | '`' | '|' | '~' ;

let ::= 'A' | 'B' | 'C' | ... | 'Z' | 'a' | ... | 'z' ;

dig ::= '0' | '1' | '2' | '3' | '4' | ... | '8' | '9' ;
```

```
lcb ::= '{';          rcb ::= '}';          dsh ::= '_';
qot ::= '''';         min ::= '-';          col ::= ':';
eql ::= '=';          rab ::= '>';          eof ::= 177b;
```

In addition to the above, there is a built-in character type called "ign"; it is assigned to all characters not appearing in any type definition. These serve as break characters (separators), but they are not part of tokens; blank is a break character, for example.

B.1.2 Symbols

Operator symbols are defined first; these include all of the special characters in the set (even though not all are used in ISP'). In addition, some special combinations are recognized as single tokens (":=", "->", and "=>'); this simplifies the parsing.

```
<min> ::= min;    <col> ::= col;    <eql> ::= eql;

<opr> ::= mrk | rcb | dsh | rab | eof | <col> |
    <col> eql | <min> | <min> rab | <eql> | <eql> rab;
```

Numbers are defined to include letters as well as digits (they must begin with a digit, of course); this permits "flexidecimal" numbers [Bilofsky73]. The number "nnnBrr" is interpreted in base "rr". If "rr" is omitted, base two is assumed, and if "Brr" is omitted, base ten is assumed. The base "rr" is always expressed in decimal. Thus "4Ø95" can also be written as "111111111111B" or "7777b8" or "ØFFFb16" (the "B" may appear in either upper or lower case).

```
<num> ::= dig | <num> dig | <num> let;
```

Identifiers begin with a letter and contain letters, digits, and dashes (underlines). They may also be "primed" any number of times (as in z', z'', ...). This usually indicates that a reference to the variable causes a side-effect.

```
<nam> ::= <idf> | <idf> qot | <nam> qot;

<idf> ::= let | <idf> let | <idf> dig | <idf> dsh;
```

Strings are used for special purposes in ISP' (see below); they do not represent character constants. Hence they are restricted to the rules used for names, but they appear with enclosing quotes. Embedded quotes are represented by two quotes; a quote itself is '''', and '' is the null string.

```
<stg> ::= <str> qot | <str> ;

<str> ::= qot |
    <str> let | <str> dig | <str> dsh | <stg> qot;
```

Finally, comments are enclosed in curly brackets; they can contain any character at all (including break characters and end-of-line markers). All text after a left bracket is ignored until a right bracket is found, which terminates the comment.

```
<cmt> ::= <com> rcb | <com> ;

<com> ::= lcb | <com> ign | <com> mrk | <com> let |
    <com> dig | <com> dsh | <com> got | <com> min |
    <com> col | <com> eql | <com> rab | <com> lcb ;
```

As in appendix A, we use the dot notation to represent an arbitrary symbol of some particular lexical type. In the syntax which follows, ".opr", ".nam", ".num", and ".stg" stand for operators, names, numbers, and strings respectively.

B.2 Syntax

An ISP description is a concatenation of declarative and procedural sections, followed by the end-of-file character (in this case, an ASCII rubout).

```
<isp> ::= <prg> 177b;             {an ISP program}

<prg> ::= <sct> | <prg> <sct> ;   {a section list}
```

Each section begins with a section name, which is a reserved keyword of the language. The order of the sections is not specified by the syntax; with one exception, the only restriction is that variables must be defined before they are referenced.

B.2.1 Declarations

Declarative sections are of five varieties, usually occuring in the order shown below. Each section name denotes some special properties of the variables declared therein.

```
<sct> ::=
   ´Mp_State´ <dcl>              |  {primary memory}
   ´External_Mp_State´ <dcl>     |  {external to Mp}
   ´Pc_State´ <dcl>              |  {processor state}
   ´External_Pc_State´ <dcl>     |  {external to Pc}
   ´Instruction_Format´ <dcl> ;  {instructions}
```

The primary memory must be the one from which instructions are fetched. External memories may include low-speed bulk memory (LCS or ECS, for example), input/output devices, or console switches and lights. Mp-State variables are usually called memories, while Pc-State variables are called registers.

B.2.1.1 Memory Definitions

Each section contains a list of declarations, optionally separated by semi-colons. The semi-colon is usually used to improve readability when several declarations are given on the same line.

```
<dcl> ::= <dec> | <dcl> <dec> | <dcl> ´;´ <dec> ;
```

A declaration is either a free-standing memory or an equivalence to a previously declared memory. Equivalences to equivalences are not allowed.

```
<dec> ::= <mry>  |                {a memory declaration}
          <mry> ´:=´ <equ> ;      {a memory equivalence}
```

A memory (register) may be a scalar or a vector, and it may or may not be bit dimensioned. Unlike ISP, the subscripts (and bitscripts) must be numeric; names are not allowed, and the comma operator is not implemented.

```
<mry> ::=
   <nam>  |                      {a single bit}
   <nam> ´<´ <rng> ´>´   |  {a register}
   <nam> ´[´ <rng> ´]´   |  {a bit vector}
   <nam> ´[´ <rng> ´]´ ´<´ <rng> ´>´ ;

<rng> ::= .num ´:´ .num;   {a sub-range}
```

A <nam> may include abbreviations. Only the first name is processed; other names are treated as comments (alias´s and "canons" are not implemented). Subscripts and bitscripts (if any) are attached to the last name in the list.

```
<nam> ::= .nam | <nam> / .nam;   {a name list}
```

B.2.1.2 Memory Equivalences

In equivalences, the syntax for the right-hand side is almost identical to the left. A single subscript or bitscript may be specified, rather than a range of values. But the total number of bits in each operand must be equal.

```
<equ> ::= .nam  |                  {a single bit}
          .nam ´<´ <fld> ´>´   |  {a register}
          .nam ´[´ <fld> ´]´   |  {a bit vector}
          .nam ´[´ <fld> ´]´ ´<´ <fld> ´>´ ;

<fld> ::= .num | .num ´:´ .num;  {a field selector}
```

ISP´ does not allow arbitrary equivalences of memories with sustantially different structure. The implementation assumes that any equivalence can be represented using an absolute bit position and a total bit length. This implies that equivalences representing vertical slices along the bit dimension of a memory are not allowed. For example, if the memory is declared as M[0:1023]<0:31>, the following equivalence is illegal:

```
signs[0:1023] := M[0:1023]<0>
```

The rule used is that if the right-hand side contains a subscript range, rather than a single expression, the bitscripts must specify the entire word of the memory. This restriction is consistent with the behavior of the hardware; while it is easy to access all bits of a single word, it is difficult to reference a single bit in all words. Memories do not behave like two-dimensional bit matrices. ([Darringer69] contains a complete discussion of this problem.)

B.2.2 Procedures

Procedural sections are of four types, usually appearing in the order shown below. In general, a "process" is defined using a procedure (or a list of procedures), whereas a "calculation" can be defined using either a procedure or a macro (macros have bit structure and value, while procedures do not).

```
<sct> ::=
   ´Instruction_Interpretation_Process´ <prc> |
   ´Instruction_Fetch_Process´ <prl>           |
   ´Effective_Address_Calculation´ <def>       |
   ´Instruction_Execution_Process´ <prl> ;
```

```
<prl> ::= <prc> | <prl> <prc> ;    {a list of prc's}

<def> ::= <prc> | <def> <prc> |    {a list of prc's}
          <mac> | <def> <mac> ;    {or mac's mixed}
```

Since it is awkward to require procedures to be declared before they are referenced, procedure names are represented by strings (enclosed in quotes) to distinguish them from memories and macros. All other variables must be declared before they are referenced.

```
        <prc> ::= .stg ':=' <act> ;  {a procedure}
```

The value of a procedure is always undefined. Any actions it performs are realized through side effects.

A macro is similar to a memory definition, except that it has code attached. The current implementation does not allow macros with parameters.

```
        <mac> ::= <mry> ':=' <act> ;  {a macro}
```

The body of a macro is often written in LISP style, as a list of condition-expression pairs. In ISP, however, the pairs may be evaluated in parallel, in which case the conditions must be mutually exclusive.

B.2.2.1 Statements

Statements are strung together with the semi-colon and the "next" operators (instead of "; next", as in ISP). The former indicates parallel evaluation, and is more binding.

```
<stl> ::= <stp> | <stl> next <stp> ;  {a step list}

<stp> ::= <act> | <stp> ';' <act> ;   {an action list}
```

Because all actions have values, both "next" and ";" are treated as binary operators, with left and right operands. The value of "X;Y" (or of "X next Y") is either X or Y, whichever is defined. If both are defined, an error results. If neither is defined, the result is undefined; this is perfectly acceptable in some situations. The value of a procedure, for example, must be undefined.

The "next" operator causes assignments to its left to be performed in parallel. The scope of a "next" is determined by its level of nesting, as well as its physical position. For example, in "A; B; (C next D)", the "next" operator applies only to C, not to A or B. In macros, an implicit "next" operation is assumed at the end of the code, applying only to assignments within it. For example, if "nw' := (M[PC]; 1+PC -> PC)" is defined as a macro, a dangling "next" is supplied, with no right operand; this is equivalent to

```
        nw' := (M[PC]; 1+PC -> PC next undefined)
```

Note the use of a quote in the name of a macro to indicate that it has a side-effect; this is a convention, not a rule.

B.2.2.2 Actions

Actions are either procedure calls (string references), register transfers (using a right-going arrow in this implementation), or conditional statements (implications). A free-standing expression is usually used to represent the value of a macro when using the LISP style mentioned above.

```
<act> ::=
   .stg   |   <exp> |   {procedure, value}
   <exp> '->' <mex> |   {register transfer}
   <exp> '=>' <act> ;   {implication (if-then)}
```

Like a procedure, the value of a register transfer is undefined. We avoid the problem of whether the left or right operand should be used. Instead, the result must be stated explicitly, using an expression following the assignment, as in

```
(PC+1 -> M[z']; PC+1)  or  (PC+1 -> M[z']; M[z'])
```

Note that if z' is a macro with a side-effect, the distinction is crucial.

The value of an implication is the value of the action if the expression is true, and it is undefined otherwise. The expression must evaluate to a single bit, so that there is no question about the meaning of "true" (1B) and "false" (ØB) in this context.

A glorified version of the "=>" operator allows a name to be attached to the conditional expression; it is interpreted as an instruction mnemonic. To avoid a local ambiguity, the mnemonic is represented by a string (enclosed in quotes). Mnemonics can only be defined in "execute" processes.

```
<act> ::= .stg ( ':=' <exp> ) '=>' <act> ;
```

B.2.2.3 Expressions

Expressions are constructed using three groups of operators: logical, relational, and arithmetic. Within each group, operators are evaluated from left to right, without individual precedence.

```
<exp> ::=                 {logical expression}
   <rex>  |               {relational}
   <exp>  &   <rex> |     {logical and}
   <exp> '|'  <rex> |     {inclusive-or}
   <exp> '='  <rex> |     {equivalence}
   <exp>  #   <rex> ;     {exclusive-or}

<rex> ::=                 {relational expression}
   <aex>  |               {arithmetic}
   <aex> lt   <aex> |     {less-than}
   <aex> le   <aex> |     {less-than or equal}
   <aex> eq   <aex> |     {equal}
   <aex> ne   <aex> |     {not-equal}
   <aex> ge   <aex> |     {greater-than or equal}
   <aex> gt   <aex> ;     {greater-than}

<aex> ::=                 {arithmetic expression}
   <atm>  |               {atomic expression}
   <aex>  +   <atm> |     {addition}
   <aex>  -   <atm> |     {subtraction}
   <aex>  *   <atm> |     {multiplication}
   <aex>  /   <atm> |     {division}
   <aex>  \   <atm> |     {remainder}
   <aex>  ^   <atm> ;     {exponentiation}
```

All of these operators have the usual meaning in integer and boolean domains; most are defined only if both operands are defined. But logical-and and inclusive-or might be defined even if one of their operands is undefined.

Atomic expressions are either numeric constants, memories, unary operations on atoms (performed right-to-left), or parenthesized step-lists. Note that any construction can appear within parentheses (including transfers, implications, and procedure calls); if the atom is used in an arithmetic expression, the value of the step-list should not be undefined.

```
<atm> ::=          {atomic expression}
    .num     |     {numeric constant}
    <mex>    |     {memory expression}
    + <atm>  |     {absolute value}
    - <atm>  |     {negation}
    ˜ <atm>  |     {complement}
    ( <stl> ) ;    {grouping}
```

The following "built-in" functions are defined for those unary operations without convenient single-character representations. The two functions "error" and "undefined" are provided so that incomplete ISP descriptions are syntactically valid, but at the same time the fact of the omission is made clear.

```
<atm> ::= {atomic expression}
    ´nop´  |  ´undefined´   |
    ´error´        ( <arg> ) |
    ´carry´        ( <arg> ) |
    ´overflow´     ( <arg> ) |
    ´zero_extend´  ( <arg> ) |
    ´sign_extend´  ( <arg> ) |
    ´normalize´    ( <arg> ) |
```

```
    ´norm_count´  ( <arg> ) |
    ´undefined´   ( <arg> ) ;

<arg> ::= <exp> | <arg> , <exp> ;
```

B.2.2.4 Registers and Memories

Memory references are similar to declarations in format, except that subscripts and bitscripts may be arbitrary arithmetic expressions. A concatenation of memories, using the dot operator in place of a box, forms a memory expression.

```
<mex> ::= <mem> | <mex> ´.´ <mem> ;  {concatenation}

<mem> ::= .nam  |                    {memory or macro}
          .nam ´<´ <sel> ´>´  |      {with bitscripts}
          .nam ´[´ <sel> ´]´  |      {with subscripts}
          .nam ´[´ <sel> ´]´ ´<´ <sel> ´>´ ;

<sel> ::= <aex> | <aex> ´:´ <aex> ;  {selector/range}
```

A variable refererence normally does not require subscripts and bitscripts; if they are omitted, the entire range as specified in the declaration of the variable is used. If a range is given, rather than a single subscript expression, it is interpreted as a concatenation; that is, M[a:b] means

M[a].M[a+1]. ... .M[b]   if a<b,

and a corresponding decreasing sequence if a>b. Note that if the two expressions are equal, the variable is subscripted as though a single expression had been written: M[a:a] := M[a]. These rules apply to bitscripting as well.

B.3 Summary

Numerous examples of ISP´ descriptions can be found in the preceding chapters and in the appendices which follow. ISP´ has been designed to correspond as closely as possible to the reference language described in [Bell71]. Some minor syntactic differences exist, largely because of a restrictive character set. More significant resrictions have been imposed on the semantics of variable definitions; this was necessary to achieve a running implementation in a short time. But the overall flavor of ISP´ is very much like ISP, and machine descriptions given in [Bell71] and elsewhere are easily adapted to the language defined here.

APPENDIX C

CASE STUDY: THE AJP-21a

This appendix contains a complete listing of the assembler which the system generated for the AJP-21a computer [Perlis72] first described in chapter 3. This machine uses a sixteen-bit word, a ten-bit address, and only two instruction formats. A single index register is synonymous with the first word of memory. The effective address calculation is unique, in that the indirect bit is treated differently in load and store operations. Load instructions use either immediate operands or memory contents, whereas store operands are either memory or indirect addresses. Both forms may be indexed. The indirect (immediate) bit and the index bit are represented by operators in the syntax (see section 5.1.2); I have chosen ´@´ and ´*´ respectively.

The appendix begins with the ISP description of the computer; note that the effective addresss calculation is defined differently than in the ISP description of chapter 3. Next, a list of the modules which make up the assembler is given; this list almost corresponds to the assembler organization outlined in section 2.4 (and 5.2). Five additional routines appear here: a main driver program

(ASM) and four input/output modules (GETLIN, GETFNM, LIST, and LOAD); these serve as interfaces to the operating system of the host computer (a DEC PDP-1Ø [DEC73]).

Following this are the modules themselves, in the order specified in the list. All of them have been included, but files of particular interest are the syntax of the assembly language (SYNTAX.SYN), the machine-dependent code generation module (CODE.I1Ø), and the bootstrapping file (BOOT). After the boot file, there is an example of an assembly listing, and an octal dump of the matching binary file; the sample program used is the maximum finder presented in appendix A. Note that these files are output of the generated assembler, not the assembler generator.

At this point, it will be useful to include a few details about the IMP programming language [Irons7Ø], and to describe the dialect of the language in which all of the assembler modules are written (IMP1Ø). IMP provides facilities roughly at the level of FORTRAN II, with some machine-oriented systems programming features added. However, IMP's style more closely resembles Algol, without block structure. All variables are global to all programs compiled together in a single module. Unlike Algol, IMP is an expression language: almost all statements, as well as expressions, have a useful value, and the statement separator ";" is a binary operator. With these preliminaries established, we illustrate the basic features of IMP1Ø with a series of short examples. This section is intended only as a quick reference guide; details of the language may be found in [Bilofsky73].

<u>Constants</u>

1976	A decimal integer constant.
77ØØB	An octal integer constant.
2AFb16	A "flexadecimal" integer constant.
'String'	An ASCII string, with terminating null.
R'c'	The integer equivalent of a character.

<u>Variables</u>

9R, 12R	PDP-1Ø hardware registers.
V, FOO9	Variable names, beginning with a letter.
V[E]	Subscripting: the Eth word of array V, counting from zero. E is an arbitrary expression.
[E]	Indirect addressing: the Eth word of memory, Mp[E]; in other words, the contents of the location whose address is E. (Again, E is an arbitrary expression.)
LOC(V)	Dereferencing: The memory address of variable V, which may be subscripted. [LOC(V)] is the same as V.
V<S,P>	A bit-field of V called a "byte"; the field is S bits long, and begins at bit position P, counting from the rightmost bit of V (bit zero). In addition, V<L> (and V<R>) refers to the left (right) halfword of V. Other byte operations peculiar to the PDP-1Ø are included in IMP1Ø, but they are not described here.

Expressions

```
(E)
```

Parentheses are used for grouping statements (instead of BEGIN-END) as well as expressions. Dyadic operators are performed right to left (as in APL), without individual precedence.

```
-E   NOT E
```

Monadic (unary) operations are performed first, before dyadic ones (but after function evaluation -- see below). The operators available are "-" (twos complement), and NOT (ones complement).

```
E+F  E-F
E*F  E/F
E//F
```

The usual integer arithmetic operations are provided; the remainder operator is "//" (E mod F).

```
E AND F  E OR F
E XOR F  E EQV F
```

The bitwise logical operations include AND, OR (inclusive or), XOR (exclusive or), and EQV (equivalence).

```
E LS F    E RS F
E ALS F   E ARS F
E LROT F  E RROT F
```

The shifting operators provided are left and right logical shift (LS, RS), arithmetic shift (ALS, ARS), and circular shift, also called rotate (LROT, RROT). Note that if the second operand is negative, the direction of the shift is reversed.

```
V_E
```

The assignment operator, originally a left-arrow, is now an underscore. It is treated as a binary operator, whose value is the quantity assigned to the variable V. Hence, expressions of the form "V_E+W_F" are legal.

```
E<F  E=F  E>F
E LT F  E LE F
E EQ F  E NE F
E GT F  E GE F
```

Relational operators are performed after assignments; they normally appear in conditional statements (they must be enclosed in parentheses elsewhere, as in V_(E>F)). The value of the expression is -1 if the relation is true and zero otherwise.

Statements

```
GO TO L
GO TO (LØ,L1,..) E
L:S
```

The infamous GO TO statement, and the equally offensive computed GO TO (as in FORTRAN, but with zero origin). The label L may be attached to any expression, using a colon.

```
E => S
E => S ELSE T
```

Conditional statements: if the value of E is non-zero, S is evaluated, otherwise T is evaluated (if it is present). The statement reads "E implies S {else T}".

```
WHILE E DO S
S UNTIL E
```

Basic loops: the statements are repeated while (or until) expression E is true. As the constructions indicate, WHILE (UNTIL) performs the test before (after) executing the statement S.

```
S FOR V IN E,F,G
S FOR V FROM E
S FOR V TO G
```

These FOR-loops execute statement S repeatedly with values of V beginning at E, incrementing by F, and terminating at G (inclusive). The FROM construction is shorthand for "S FOR V IN E,-1,Ø", and the "TO" format is short for "S FOR V IN Ø,+1,G". Note that S is always performed at least once (as in FORTRAN).

```
READ V,W,...
PRINT E,F,...
```

These statements cause files to be read and written. Formatting directives may also be included in the variable (expression) list.

Programs and Subprograms

```
S;T;...;U %%
```

A program is a list of statements separated by semicolons and terminated by double percent signs. As mentioned, ";" is a binary operator; the value of "S;T" is T. Unlike other dyadic operators, semicolons are evaluated left to right. For example, the value of "(A_X; B_3)+9" is twelve.

```
SUBR X(A,B) IS E
SUBR Y() IS E
```

A subroutine X is defined this way, with formal parameters A and B (subroutine Y has no parameters). In IMP, there is no distinction between functions and subroutines; the value of E is always returned (but perhaps ignored). IMP1Ø uses call by address.

```
RETURN E
```

RETURN interrupts the sequential execution of a subroutine, returning the value of expression E.

```
X(E,F)
Y()
```

These constructions cause execution of the function X (or Y), with arguments E and F (with no arguments). The arguments may be arbitrary expressions.

Declarations and Data Allocation

LET X=3,Y=BF[2]

The LET statement declares synonyms using the form N=V, where N is a name and V is a constant or a simple or subscripted variable. Whenever N appears in the program, the effect is as if V had been written.

X,Y ARE COMMON
Z IS 3 LONG

In general, a declaration contains a list of names, followed by a list of properties to be associated with all of the names. COMMON, for example, makes the names known outside the module.

DATA(E,F,...)
REMOTE S

DATA causes initialized storage to be allocated; the expressions must evaluate to constants at compile time. If a variable name appears in the list, its address is used as the value of the data. REMOTE causes the code for the statement S to be inserted at the end of the program, rather than in-line. Thus variables may be initiallized lexically near to their first reference (by using REMOTE X:DATA(3), for example).

Syntax Macros

Finally, IMP is an extensible language. There are several mechanisms provided for defining extensions to the language; we will describe only one technique, called "quoted semantics" in IMP1Ø. This feature is similar to the syntax macros described in [Leavenworth66]. A modified BNF notation (without alternation) is used to describe the syntax of an extension; the semantic part is enclosed in quotes and added on the right. For example,

```
    <ATOM> ::= ABS <ATOM,A> ::= "A<Ø => -A ELSE A"
```

Notice that a name has been attached to the non-terminal ATOM so that it can be referenced in the semantic part. The

quoted semantics can contain any valid IMP expression, including a statement list. There is also a facility for declaring variables local to the semantics. The following is a simplified definition of the WHILE statement (<STM> is the syntactic type representing statements, and <EXP> corresponds to expressions):

```
      <STM> ::= WHILE <EXP,A> DO <STM,B>
            ::= LOCAL L IN "L: A => (B; GO TO L)"
```

The extension mechanisms can also be used to implement basic data structures. For example, a three word record representing a node in a binary tree (containing a type, a left pointer, and a right pointer) might be defined as follows (<VBL> is the syntactic type of variables):

```
        <VBL> ::= <VBL,A>.TYP ::= "[A]";
        <VBL> ::= <VBL,A>.CAR ::= "[A+1]";
        <VBL> ::= <VBL,A>.CDR ::= "[A+2]";
```

If PTR is a pointer to a record with the above format, the notation PTR.TYP, PTR.CAR, and PTR.CDR can be used to refer to its fields. This technique is used frequently in the assembler modules which follow.

```
{ISP 3.6                                               16-Sep-75}

                     {AJP21A LISTING}


{AJP21A.LST[22,54]=AJP21A.ISP[22,54]      16-Sep-75    00:10}

Mp_State

   M/Primary_Memory[0:1023]<0:15>
   X/Index_Register<0:15> := M[0]


Pc_State

   PC/Program_Counter<0:9>
   C/Carry; AC/Accumulator<0:15>
   E/Effective_Operand<0:15>
   EA/Effective_Address<0:9> := E<6:15>
   Run

External_Pc_State

   box/IO_Port<0:15>


Instruction_Format

   I/Instruction<0:15>
      op/op_code<0:3> := I<0:3>
      ib/indirect_bit := I<4>
      xb/index_bit    := I<5>
      a/address<0:9>  := I<6:15>


Effective_Address_Calculation

  'Effaddr' :=
     (a -> E next
      xb => X+E -> E next
      ib => M[EA] -> E)


Instruction_Interpretation_Process

  'Interpret' :=
      Run =>
        (M[PC] -> I; 1+PC -> PC next
         op ne 3 & op ne 11 => 'Effaddr'
         next 'Execute' next 'Interpret')


Instruction_Execution_Process
```

```
'Execute' :=
  ('JMP' (:= op eq  0) => EA -> PC;
   'JIF' (:= op eq  1) => C => EA -> PC;
   'SIT' (:= op eq  2) => (X-1 -> X next X eq 0 => EA -> PC);
   'HLT' (:= op eq  3) => 0 -> Run;
   'ADD' (:= op eq  4) => AC+E -> AC;
   'SUB' (:= op eq  5) => AC-E -> AC;
   'MPY' (:= op eq  6) => AC*E -> AC;
   'DIV' (:= op eq  7) => AC/E -> AC;
   'EQL' (:= op eq  8) => AC eq E -> C;
   'GTR' (:= op eq  9) => AC gt E -> C;
   'AND' (:= op eq 10) => (AC&E -> AC next AC eq -1 -> C);
   'NOT' (:= op eq 11) => (~AC -> AC next AC<15> -> C);
   'LOD' (:= op eq 12) => E -> AC;
   'STO' (:= op eq 13) => AC -> M[EA];
   'GET' (:= op eq 14) => box -> M[EA];
   'PUT' (:= op eq 15) => E -> box)
```

```
ASM(YL)             ;$Main driver
GETLIN(YL)          ;$Reads a TTY command line
GETFNM(YL)          ;$Parses DEV:FIL.EXT[PRJ,PRG]
INPUT(YL)           ;$Buffered ASCII mode input
; LEX.SYN(LAG)      ; Syntax for symbols in BNF
LEX(YL)             ;#Lexical analyser
CONVRT(Y)           ;*Conversion routines
PARSER(YL)          ; Conway's Parser
; SYNTAX.SYN(PAG)   ;*Syntax for statements in BNF
SYNTAX(Y)%"NOSTART" ;#Cohen & Gotlieb's syntax graphs
SEMTIC(Y)           ;*Machine-independent semantics
CODE(Y)             ;*Machine-dependent semantics
INTRPT(YL)          ; Assembly time interpreter & stack
SYMBOL(Y)           ;*Symbol table search
LISTER(Y)           ;*List file formatter
LIST(YL)            ;$Buffered ASCII mode output
LOADER(Y)           ;*Binary file formatter
LOAD(YL)            ;$Buffered binary mode output
STORAG(YL)          ;$Dynamic storage allocation
F40:FORTIO/LIB      ;$FORTRAN UUO interface
;
; ASM.CMD   20-Oct-73   10-Sep-75   Basic assembler
;
; *'ed files are produced by ISP assembler generator.
; #'ed files are produced by LAG or PAG generators.
; $'ed files are machine dependent (provided by user).
; all other files remain constant across assemblers.
```

```
#************************************************************
*                                                           *
*                  Assembler Main Program                   *
*                                                           *
************************************************************#

# ASM.I10   20-Oct-73   10-Sep-75 #   CALL ME ASM;

!.JBSA!,!.JBFF! ARE COMMON;
SWITCH,ERRFLG ARE COMMON,1 LONG;

LET CS=7, WS=36, CRLF=064240000000B;

<ST> ::= RESET ::= "DATA(047000000000B)";
<ST> ::= EXIT  ::= "DATA(047000000012B)";
<ST> ::= OUTSTR <EXP,A> ::= DEWOP(051B,AREG(23B),A);

   LINBUF IS 16 LONG; DEV,FIL,EXT ARE 2 LONG;
   RESET;
     (LP_BYTEP LINBUF<CS,WS>; GETLIN(LP);
      GETFNM(LP,DEV,FIL,EXT,PRJ,PRG,SWITCH);
      DEV=0 => DEV_'DSK';
      ERR_OPENI(DEV,FIL,EXT,PRJ,PRG) =>
        (OUTSTR('%Input open failure'); OUTSTR(CRLF)))
   UNTIL ERR=0;
   OPENL('DSK',FIL,'LST',0,0) =>
      SYSERR('?List output open failure');
   OUTSTR('Pass 1');
   ISEM(); ILST(); ILEX(); LEX();
   WHILE ERR=0 DO PARSE() => ERR_ERROR() ELSE ERR_NOT 0;
   ERRFLG => OUTSTR(' errors'); OUTSTR(CRLF);
   REOPNI(FIL,EXT,PRJ,PRG);
   OPENO('DSK',FIL,'BIN',0,0) =>
      SYSERR('?Binary output open failure');
   OUTSTR('Pass 2');
   ERR_0; ISEM(); ILST(); BEGLOD(); ILEX(); LEX();
   WHILE ERR=0 DO PARSE() => ERR_ERROR() ELSE ERR_NOT 0;
   CLOSEI(); CLOSEL(); ENDLOD(); CLOSEO();
   ERRFLG => OUTSTR(' errors')
   ELSE !.JBSA!<L>_!.JBFF!<R>; EXIT;

SUBR SYSERR(M) IS (OUTSTR(M); OUTSTR(CRLF); EXIT) %%
```

```
#***********************************************************
*                                                         *
*                      Get a TTY Line                     *
*                                                         *
***********************************************************

GETLIN(BP)
   Effect: Inputs a line from the user Teletype.  An '*' is
      output before input is accepted.  CR (15B), LF (12B),
      ESC (33B) and EOF (32B) terminate the input line (but
      do not appear in the output buffer).  CRLF is echoed
      (if not done by the monitor) and the buffer is padded
      to 80 characters with zeros if necessary (at least one
      null always terminates the buffer).
   Parameters:
      BP - Byte pointer to 80 entry input line buffer.
   Value: Returns the line break char (CR, LF, ESC or EOF).
   Errors: Input characters past the 80th (not counting
      those deleted by rubout or cursor left) are ignored,
      except for the line terminator. #

# GETLIN.I10   07-Dec-71   09-Sep-75 #    CALL ME GETLIN;

 <ST>    ::= IF <EXP,A> <RELOP,EQ> <EXP,B> THEN <ST,C>
         ::= EQ/"A=B => C";
 <ST>    ::= IF <EXP,A> THEN <ST,B>
         ::= "A => B";
 <ST>    ::= IF <EXP,A> <RELOP,EQ> <EXP,B>
             THEN <ST,C> ELSE <ST,D>
         ::= LOCAL EL IN EQ/"A=B => (C; GO TO EL); D; EL:0";
 <ST>    ::= IF <EXP,A> THEN <ST,B> ELSE <ST,C>
         ::= "IF A NE 0 THEN B ELSE C";
 <EXP>   ::= <VBL,A>_INCHWL ::= DEWOP(051B,AREG(4),A);
 <ATOM>  ::= OUTCHR <EXP,A> ::= DEWOP(051B,AREG(1),A);
 <ATOM>  ::= OUTSTR <EXP,A> ::= DEWOP(051B,AREG(3),A);

 LET LIM=80,CRLF=064240000000B;

SUBR GETLIN(BP) IS
  (C,I,OP ARE REGISTER;
   OUTCHR(R'*'); C_0; I_0; OP_BP;
 L:C_INCHWL;
   IF C NE 32B THEN
     (IF C NE 12B THEN
        (IF C NE 15B THEN
           (IF C NE 33B THEN
              (IF I<LIM-1 THEN (<+OP>_C; I_I+1); GO TO L);
            OUTSTR(CRLF))
         ELSE 0R_INCHWL));
   <+OP>_0 UNTIL I_I+1 GE LIM;
   I,OP ARE RELEASED;
   C) %%
```

```
#***********************************************************
*                                                         *
*                File Specification Parser                *
*                                                         *
***********************************************************

GETFNM(BP,DEV,FIL,EXT,PRJ,PRG,SWS)
   Effect: Parses an ASCIZ string (converting to upper-case)
      like dev:filename.extension[proj,prog](abc)/def/h/i/j
      into its constituents, advancing BP past the string.
   Parameters:
      BP  - Byte pointer to buffer containing input text.
      DEV - Set to device name, 1 to 9 characters.
      FIL - Set to file name, 1 to 9 characters.
      EXT - Set to file extension, 1 to 5 characters.
      PRJ - Set to project number, 0 to 777777777777B.
      PRG - Set to programmer number, 0 to 777777777777B.
      SWS - Set to switches, A-Z, 0-9 being bits 0-35.
   Value: Break char which terminated the file spec.
   Errors: Any char not valid for the current item, or not
      the delimiter of the current item ":.[,]()/", is taken
      as the break character and terminates the parse.  Any
      argument not found in the input is set to zero.     #

# GETFNM.I10   07-Dec-71   12-Sep-75 #   CALL ME GETFNM;

LET LIM=10;
IP,OP,C,T,I,BF,BF1,O,S ARE REGISTER,RESERVED,SCRATCH;

SUBR GETFNM(BP,DEV,FIL,EXT,PRJ,PRG,SWS) IS
  (IP_BP; DEV_0; DEV[1]_0; FIL_0; FIL[1]_0;
   EXT_0; PRJ_0; PRG_0; SWS_0;
   !.TOKE.!();
   C=R':' => (DEV_BF; DEV[1]_BF1; !.TOKE.!());
   FIL_BF; FIL[1]_BF1;
   C=R'.' => (!.TOKE.!(); EXT_BF);
   C=R'[' =>
     (!.TOKE.!(); PRJ_O;
      C=R',' => (!.TOKE.!(); PRG_O);
      C=R']' => !.TOKE.!());
   C=R'(' =>
     (!.TOKE.!(); SWS_SWS OR S;
      C=R')' => !.TOKE.!());
   WHILE C=R'/' DO (!.TOKE.!(); SWS_SWS OR S);
   BP_IP; C);

SUBR !.TOKE.!() IS
  (BF_0; BF1_0; O_0; S_0;
   OP_BYTEP BF<7,36>; I_0;
   WHILE 1 DO
     (C_<+IP>; C>R'_' => C_C-40B; T_0;
      C GE R'0' => C LE R'9' => T_1;
      C GE R'A' => C LE R'Z' => T_1;
      T=0 => RETURN(0R);
   I_I+1<LIM => <+OP>_C;
   C<R'8' => O_(O_O LS 3) OR C AND 7B;
   C<R'A' => C_C + (R'Z'+1) - R'Ø';
   S_S OR 1 LS (35+R'A') - C);
ØR) %%
```

```
#**************************************************************
*                                                             *
*                        Input Module                         *
*                                                             *
***************************************************************

OPENI(DEV,FIL,EXT,PRJ,PRG)
   Effect: Initialize a file for input.
   Parameters:
      DEV - Device name terminated by a null byte.
      FIL - File name terminated by a null byte.
      EXT - File extension terminated by a null byte.
      PRJ - Project number (octal); if zero, this UFD.
      PRG - Programmer number (octal); if zero, this UFD.
   Value: Error code (usually file-not-found), zero if ok.
   Errors: None.

GET(NXT)
   Effect: Advances the current file to the next character.
   Parameters:
      NXT - Set to the next byte from the current file.
   Value: Error code (usually end-of-file), zero if ok.
   Errors: None.

REOPNI(FIL,EXT,PRJ,PRG)
   Effect: Closes the current file and opens a new one,
      re-using the current device and the current buffers.
   Parameters: As in OPENI.
   Value: Error code (usually file-not-found), zero if ok.
   Errors: None.

CLOSEI()
   Effect: Terminates input processing.
   Parameters: None.
   Value: None.
   Errors: None.   #

# INPUT.I1Ø   16-Nov-72   Ø9-Sep-75 #   CALL ME INPUT;

LET MODE=Ø, BYTSIZ=Ø, LEN=128, NBUF=2, IGN=1;
IBUFS IS NBUF*LEN+3 LONG; !.JBFF! IS COMMON;

SUBR OPENI(DEV,FIL,EXT,PRJ,PRG) IS
  (CH_GETCHA(Ø)<Ø => RETURN CH;
   E_INIT(CH,MODE,DEV,'Ø',IBUF) => RETURN E;
   E_LOOKUP(CH,FIL,EXT,PRJ,PRG) => (RELEA(CH); RETURN E);
   SAVJBF_!.JBFF!<R>; !.JBFF!<R>_LOC(IBUFS);
   INBUF(CH,NBUF); !.JBFF!<R>_SAVJBF;
   BYTSIZ => IBUF[1]<6,24>_BYTSIZ;
   Ø);   IBUF:DATA(Ø,Ø,Ø);

SUBR REOPNI(FIL,EXT,PRJ,PRG) IS
  (CLOSE(CH); LOOKUP(CH,FIL,EXT,PRJ,PRG));
```

```
SUBR GET(NXT) IS
  (LP:WHILE IBUF[2]_IBUF[2]-1<Ø DO
        (E_INPUT(CH) => RETURN E);
      IGN => (NXT_<+IBUF[1]>=Ø => GO TO LP)
      ELSE NXT_<+IBUF[1]>;
   Ø);

SUBR CLOSEI() IS RELEA(CH) %%
```

```
{LEX.SYN                                                    1Ø-Sep-75}

                          {Syntax for Symbols}

                'CHARSIZE'=7;          'TOKSIZE'=8Ø;

mrk ::= '!' | '"' | '#' | '$' | '%' | '&' | '(' | ')' |
        '*' | '+' | ',' | '-' | '/' | ':' | ';' | '<' |
        '=' | '>' | '@' | '[' | '\' | ']' | '^' | '_' |
        '`' | '{' | '|' | '}' | '~';

let ::= '.' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
        'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' |
        'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' |
        'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' |
        'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' |
        'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
        'v' | 'w' | 'x' | 'y' | 'z';

dig ::= 'Ø' | '1' | '2' | '3' | '4' | '5' | '6' | '7' |
        '8' | '9';

blk ::= ' ';           qot ::= '''';          qtm ::= '?';
eol ::= 12b;           eof ::= 177b;

<opr> ::= mrk | eol | eof ;

<nam> ::= let | <nam> let | <nam> dig ;

<num> ::= dig | <num> let | <num> dig ;

<stg> ::= <str> qot [OMIT] | <str> ;

<str> ::= qot [OMIT] |
          <str> blk | <str> mrk | <str> let |
          <str> dig | <stg> qot | <str> qtm ;

<cmt> [IGNORE] ::= qtm [OMIT] | <cmt> ign [OMIT] |
               <cmt> blk [OMIT] | <cmt> mrk [OMIT] |
               <cmt> let [OMIT] | <cmt> dig [OMIT] |
               <cmt> qot [OMIT] | <cmt> qtm [OMIT] ;
```

```
# LGS 2.3                                              1-Jun-74 #

                     # Lexical Analyzer #
                        CALL ME LEX;


# LEX.I10[22,54]=LEX.SYN[22,54]          10-Sep-75   05:31 #


LET TOKSIZ=80,EOF=177B;
LET NT=9,MASK=4,RETURN=10B,CLEAR=4B,PACK=2B,READ=1B;

LET LTYP=TOK,LADR=TOK[1],LLEN=TOK[2],LVAL=TOK[3];
TOK IS 3+TOKSIZ+1 LONG,COMMON;

<ST> ::= ENTER ::= "GO TO [RT]; GO:0";
<ST> ::= LEAVE <EXP,A>
     ::= LOCAL L IN "RT_LOC(L); RETURN A; L:0";

SUBR ILEX() IS
  (STATE_0; LEN_0; ARC_READ; RT_LOC(GO); 0);

SUBR LEX() IS
  (ENTER;
   WHILE 1 DO
     (ARC AND READ =>
        (GET(CHR) => CHR_EOF; TYPE_TYPES[CHR]);
      ARC_STATES[TYPE+STATE*NT];
      ARC AND RETURN =>
        (LTYP_STATE; LADR_0; LLEN_LEN;
         LVAL[LEN]_0; LEAVE 0);
      ARC AND CLEAR => LEN_0;
      STATE_ARC RS MASK;
      ARC AND PACK =>
        (LEN<TOKSIZ => (LVAL[LEN]_CHR; LEN_LEN+1));
      ARC AND READ => PUTCHR(CHR));
   0);

# LET init=0,opr=1,nam=2,num=3,stg=4,str=5,cmt=6; #

STATES:
 DATA(001B,023B,043B,063B,001B,121B,141B,023B,022B);
 DATA(015B,037B,057B,077B,015B,135B,155B,037B,036B);
 DATA(015B,037B,043B,043B,015B,135B,155B,037B,036B);
 DATA(015B,037B,063B,063B,015B,135B,155B,037B,036B);
 DATA(015B,037B,057B,077B,015B,123B,155B,037B,036B);
 DATA(100B,123B,123B,123B,123B,101B,123B,100B,100B);
 DATA(141B,141B,141B,141B,141B,141B,141B,027B,026B);

LET ign=0,mrk=1,let=2,dig=3,blk=4,qot=5,qtm=6,eol=7,eof=8;

TYPES:
 DATA(ign,ign,ign,ign,ign,ign,ign,ign,ign,ign,eol,ign,ign);
 DATA(ign,ign,ign,ign,ign,ign,ign,ign,ign,ign,ign,ign,ign);
```

```
DATA(ign,ign,ign,ign,ign,ign,blk,mrk,mrk,mrk,mrk,mrk,mrk);
DATA(qot,mrk,mrk,mrk,mrk,mrk,mrk,let,mrk,dig,dig,dig,dig);
DATA(dig,dig,dig,dig,dig,dig,mrk,mrk,mrk,mrk,mrk,qtm,mrk);
DATA(let,let,let,let,let,let,let,let,let,let,let,let,let);
DATA(let,let,let,let,let,let,let,let,let,let,let,let,let);
DATA(mrk,mrk,mrk,mrk,mrk,mrk,let,let,let,let,let,let,let);
DATA(let,let,let,let,let,let,let,let,let,let,let,let,let);
DATA(let,let,let,let,let,let,mrk,mrk,mrk,mrk,eof) %%
```

```
# ISP 3.6                                                16-Sep-75 #

                       # AJP21A #
                    CALL ME CONVRT;


# CONVRT.I1Ø[22,54]=AJP21A.ISP[22,54]      16-Sep-75    ØØ:1Ø #

# number bases #
 LET BASE=1Ø, RADIX=4, POWER=2;

# ASCII: character set functions #

 <ATOM> ::= NUM <ATOM,A> ::= LOCAL C,L IN
   "C IS REGISTER; C_A>R' ' => C_C-4ØB;
    C GE R'Ø' => C LE R'9' =>
      (C_C-R'Ø'; GO TO L);
    RADIX>1Ø =>
       C GE R'A' => C LE R'A'+(RADIX-1)-1Ø =>
         (C_C-R'A'-1Ø; GO TO L);
    C_-1; L:C";

 <ATOM> ::= REP <ATOM,A> ::= LOCAL C,L IN
   "C IS REGISTER; C_A GE Ø =>
      (C LE 9 => (C_C+R'Ø'; GO TO L);
       RADIX>1Ø => C LE RADIX-1 =>
         (C_C+R'A'-1Ø; GO TO L));
    C_-1; L:C";

# IMP: Absolute value function #
 <ATOM> ::= ABS <ATOM,A>
        ::= DEWOP(214B,AREG1(1,15B),A);

# convert binary word to its bit length #

SUBR BITLEN(V) IS
  (ANS,VAL ARE REGISTER;
   ANS_Ø; VAL_V;
   ANS_ANS+1 UNTIL VAL_VAL RS 1=Ø;
   ANS); ANS,VAL ARE RELEASED;

# convert char array to char length #

SUBR STGLEN(S) IS
  (PTR IS REGISTER; PTR_Ø;
   WHILE S[PTR] NE Ø DO PTR_PTR+1;
   PTR); PTR IS RELEASED;

# convert char array to integer value #

SUBR STGCHR(S,W) IS (W_S; S[1] => NOT Ø ELSE Ø);

# Convert char array to its integer value #
```

```
SUBR STGIGR(S,W) IS
  (ERR,ANS,PTR,FLG,CHR ARE REGISTER;
   ERR_Ø; PTR_Ø; ANS_Ø; FLG_Ø;
   # see if there's a '.' anywhere in S #
   WHILE (CHR_S[PTR] NE Ø) AND (FLG=Ø) DO
     (CHR=R'.' => FLG_NOT Ø; PTR_PTR+1);
   # if so it's decimal (use BASE) #
   PTR_Ø; FLG =>
     (WHILE CHR_S[PTR] DO
        (CHR NE R'.' =>
           (CHR_NUM(CHR)<Ø => (CHR_Ø; ERR_NOT Ø);
            CHR GE BASE => (CHR_Ø; ERR_NOT Ø);
            ANS_ANS*BASE; ANS_ANS+CHR);
         PTR_PTR+1);
      W_ANS; RETURN ERR);
   # otherwise it's binary (use RADIX) #
   WHILE CHR_S[PTR] DO
     (CHR_NUM(CHR)<Ø => (CHR_Ø; ERR_NOT Ø);
      CHR GE RADIX => (CHR_Ø; ERR_NOT Ø);
      ANS_ANS LS POWER; ANS_ANS OR CHR; PTR_PTR+1);
   W_ANS; ERR); ERR,ANS,PTR,FLG,CHR ARE RELEASED;

# Convert integer to character array
  (in binary), unsigned zero fill #

SUBR BINSTG(V,D,L) IS
  (PTR,TMP ARE REGISTER; TMP_V;
     (D[PTR]_REP(TMP AND RADIX-1);
      TMP_TMP RS POWER)
   FOR PTR FROM L-1;
   Ø); PTR,TMP ARE RELEASED;

# Convert integer to character array
  (in decimal), signed blank fill #

SUBR DECSTG(V,D,L) IS
  (PTR,TMP ARE REGISTER;
   PTR_L-1; TMP_ABS(V);
     (PTR GE Ø => (D[PTR]_REP(TMP//BASE); PTR_PTR-1))
   UNTIL TMP_TMP/BASE=Ø;
   V<Ø => PTR GE Ø => (D[PTR]_R'-'; PTR_PTR-1);
   WHILE PTR GE Ø DO (D[PTR]_R' '; PTR_PTR-1);
   Ø); PTR,TMP ARE RELEASED %%
```

```
# PAG 2.3                                    18-Aug-74 #

             # Top-down No-backup Parser #
                   CALL ME PARSER;


# PARSER.I10[22,54]                    10-Sep-75   05:36 #

ROOT,NODE ARE COMMON;

# LEX: lexeme format # LET
 LTYP=TOK,      # lexical type #
 LADR=TOK[1],   # symbol table pointer #
 LLEN=TOK[2],   # character length #
 LVAL=TOK[3];   # value (char-array) #
 TOK IS COMMON; # owned by LEX #

# PARSER: syntax graph format # LET
 PTYP=[P],   # lex type or "ntm"   #
 PVAL=[P+1], # char or ptr to ntm  #
 PSEM=[P+2], # semantic routine    #
 PALT=[P+3], # alternate down ptr  #
 PSUC=[P+4]; # successor right ptr #

<ATOM> ::= CALL <VBL,A> ::= LOCAL C IN "C<R>_A; C:PARSE()";

SUBR PARSE() IS
  (IPS(); CNT_0; P_ROOT;
   WHILE 1 DO
      PTYP=NODE =>
        (PON(CNT,P) => RETURN NOT 0; CNT_0; P_PVAL)
      ELSE
         (PTYP=0) OR (PTYP=LTYP) AND
         (PVAL=0) OR (PVAL=LVAL) =>
           (PSEM => CALL PSEM;
            PTYP => (CNT_CNT+1; LEX());
            WHILE P_PSUC=0 DO
              (POF(TMP,P) => RETURN 0;
               CNT_CNT+TMP; PSEM => CALL PSEM))
         ELSE
           (WHILE PALT=0 DO
              (CNT>0 => RETURN NOT 0;
               POF(CNT,P) => RETURN NOT 0);
            P_PALT);
   0);

# parser non-terminal stack routines #

LET PSL=20, PSN=2;   PSTK IS PSL*PSN LONG;

SUBR IPS() IS PSP_-1;

SUBR PON(A,B) IS
  (PSP GE (PSL*PSN)-1 => RETURN NOT 0;
```

```
   PSP_PSP+1; PSTK[PSP]_A; PSP_PSP+1; PSTK[PSP]_B; 0);

SUBR POF(A,B) IS
  (PSP<0 => RETURN NOT 0;
   B_PSTK[PSP]; PSP_PSP-1; A_PSTK[PSP]; PSP_PSP-1; 0) %%
```

{AJP21A SYNTAX}

```
{SYNTAX.SYN[22,54]=AJP21A.ISP[22,54]       16-Sep-75   ØØ:1Ø}

.opr=1;         .nam=2;         .num=3;         .stg=4;
.pag=5;         .til=6;         .beg=7;         .aln=8;
.org=9;         .def=1Ø;        .vfd=11;        .pad=12;
.fup=13;        .dat=14;        .fmt2Ø=2Ø;      .fmt21=21;

         opr=.opr;        num=.opr;        stg=.opr;

<prg> ::= <stl> 177b [END];

<stl> ::= 12b              [LINE] |
          <stm> 12b        [LINE] |
          <stl> 12b        [LINE] |
          <stl> <stm> 12b  [LINE] ;

<stm> ::=
   [HASH] .nam [MNEMON] ´:´ [LABEL] |
   [HASH] .nam [MNEMON] ´:´ [LABEL] <stm> |
   [HASH] .nam [MNEMON] ´=´ <exp> [EQUATE];

<vfd> ::= ´<´ <exp> ´>´ [SIZE] <exp> [FIELD] |
    <vfd> ´<´ <exp> ´>´ [SIZE] <exp> [FIELD] |
    <vfd> ´,´ <exp> [FIELD];

<exp> ::= <atm> |
   <exp> ´+´ <atm> [ADD] | <exp> ´-´ <atm> [SUB] |
   <exp> ´*´ <atm> [MUL] | <exp> ´/´ <atm> [DIV] |
   <exp> ´\´ <atm> [REM] | <exp> ´&´ <atm> [AND] |
   <exp> ´|´ <atm> [IOR] | <exp> ´=´ <atm> [EQV] |
   <exp> ´#´ <atm> [XOR] | <exp> ´~´ <atm> [SHF] ;

<atm> ::=
   .nam [NAME] | .num [NUMBER] | .stg [STRING] |
   ´~´ <atm> [NOT] | ´|´ <atm> [ABV] |
   ´-´ <atm> [NEG] | ´+´ <atm> |
   ´(´ <exp> ´)´ ;

<stm> ::= [HASH] .fmt2Ø [PROLOG]
          <exp> [SETa] [EPILOG];
<stm> ::= [HASH] .fmt2Ø [PROLOG]
          ´*´ [SET1xb] <exp> [SETa] [EPILOG];
<stm> ::= [HASH] .fmt2Ø [PROLOG]
          ´@´ [SET1ib] <exp> [SETa] [EPILOG];
<stm> ::= [HASH] .fmt2Ø [PROLOG]
          ´@´ [SET1ib] ´*´ [SET1xb] <exp> [SETa] [EPILOG];

<stm> ::= [HASH] .fmt21 [PROLOG] [EPILOG];
```

```
<stm> ::=
   [HASH] .org ´´    [ORIGIN] |
   [HASH] .org <exp> [ORIGIN] |
   [HASH] .dat <exp> [DATA]   |
   [HASH] .vfd <vfd> |
   [HASH] .aln <exp> [ALIGN]  |
   [HASH] .pad <exp> [PAD]    |
   [HASH] .pag       [PAGE]   |
   [HASH] .til .stg  [TITLE]  |
   [HASH] .beg <exp> [BEGIN]  |
   [HASH] .def .stg  [MNEMON]
     ´(´.nam [FORMAT]´)´ ´=´ <exp> [OPDEF] |
   .stg [MNEMON] <exp> <exp> <exp> <exp> [BOOT];
```

```
            # Parser and Syntax Graph #
                CALL ME SYNTAX;


# SYNTAX.I1Ø[22,54]=SYNTAX.SYN[22,54]     16-Sep-75    ØØ:15 #

ROOT,NODE ARE COMMON;   ROOT:DATA(prg1);   NODE:DATA(22);

LET ntm=22,opr=1,nam=2,num=3,stg=4,pag=5,til=6,beg=7,aln=8,
 org=9,def=1Ø,vfd=11,pad=12,fup=13,dat=14,fmt2Ø=2Ø,fmt21=21;

END,LINE,HASH,MNEMON,LABEL,EQUATE,SIZE,FIELD,ADD,SUB,MUL,
DIV,REM,AND,IOR,EQV,XOR,SHF,NAME,NUMBER,STRING,NOT,ABV,NEG,
PROLOG,SETa,EPILOG,SET1xb,SET1ib,ORIGIN,DATA,ALIGN,PAD,PAGE,
TITLE,BEGIN,FORMAT,OPDEF,BOOT ARE COMMON;

prg1:DATA(ntm,st11,Ø,Ø,prg2);
prg2:DATA(opr,177B,END,Ø,Ø);

st11:DATA(opr,12B,LINE,st16,st12);
st12:DATA(opr,12B,LINE,st13,st12);
st13:DATA(ntm,stm1,Ø,st15,st14);
st14:DATA(opr,12B,LINE,Ø,st12);
st15:DATA(Ø,Ø,Ø,Ø,Ø);
st16:DATA(ntm,stm1,Ø,Ø,st17);
st17:DATA(opr,12B,LINE,Ø,st12);

stm1:DATA(stg,Ø,MNEMON,stm6,stm2);
stm2:DATA(ntm,exp1,Ø,Ø,stm3);
stm3:DATA(ntm,exp1,Ø,Ø,stm4);
stm4:DATA(ntm,exp1,Ø,Ø,stm5);
stm5:DATA(ntm,exp1,BOOT,Ø,Ø);
stm6:DATA(Ø,Ø,HASH,Ø,stm7);
stm7:DATA(nam,Ø,MNEMON,stm13,stm8);
stm8:DATA(opr,R´:´,LABEL,stm11,stm9);
stm9:DATA(ntm,stm1,Ø,stm1Ø,Ø);
stm1Ø:DATA(Ø,Ø,Ø,Ø,Ø);
stm11:DATA(opr,R´=´,Ø,Ø,stm12);
stm12:DATA(ntm,exp1,EQUATE,Ø,Ø);
stm13:DATA(fmt2Ø,Ø,PROLOG,stm25,stm14);
stm14:DATA(ntm,exp1,SETa,stm16,stm15);
stm15:DATA(Ø,Ø,EPILOG,Ø,Ø);
stm16:DATA(opr,R´*´,SET1xb,stm19,stm17);
stm17:DATA(ntm,exp1,SETa,Ø,stm18);
stm18:DATA(Ø,Ø,EPILOG,Ø,Ø);
stm19:DATA(opr,R´@´,SET1ib,Ø,stm2Ø);
stm2Ø:DATA(ntm,exp1,SETa,stm22,stm21);
stm21:DATA(Ø,Ø,EPILOG,Ø,Ø);
stm22:DATA(opr,R´*´,SET1xb,Ø,stm23);
stm23:DATA(ntm,exp1,SETa,Ø,stm24);
stm24:DATA(Ø,Ø,EPILOG,Ø,Ø);
```

```
stm25:DATA(fmt21,Ø,PROLOG,stm27,stm26);
stm26:DATA(Ø,Ø,EPILOG,Ø,Ø);
stm27:DATA(org,Ø,Ø,stm3Ø,stm28);
stm28:DATA(ntm,exp1,ORIGIN,stm29,Ø);
stm29:DATA(Ø,Ø,ORIGIN,Ø,Ø);
stm3Ø:DATA(dat,Ø,Ø,stm32,stm31);
stm31:DATA(ntm,exp1,DATA,Ø,Ø);
stm32:DATA(vfd,Ø,Ø,stm34,stm33);
stm33:DATA(ntm,vfd1,Ø,Ø,Ø);
stm34:DATA(aln,Ø,Ø,stm36,stm35);
stm35:DATA(ntm,exp1,ALIGN,Ø,Ø);
stm36:DATA(pad,Ø,Ø,stm38,stm37);
stm37:DATA(ntm,exp1,PAD,Ø,Ø);
stm38:DATA(pag,Ø,PAGE,stm39,Ø);
stm39:DATA(til,Ø,Ø,stm41,stm4Ø);
stm4Ø:DATA(stg,Ø,TITLE,Ø,Ø);
stm41:DATA(beg,Ø,Ø,stm43,stm42);
stm42:DATA(ntm,exp1,BEGIN,Ø,Ø);
stm43:DATA(def,Ø,Ø,Ø,stm44);
stm44:DATA(stg,Ø,MNEMON,Ø,stm45);
stm45:DATA(opr,R´(´,Ø,Ø,stm46);
stm46:DATA(nam,Ø,FORMAT,Ø,stm47);
stm47:DATA(opr,R´)´,Ø,Ø,stm48);
stm48:DATA(opr,R´=´,Ø,Ø,stm49);
stm49:DATA(ntm,exp1,OPDEF,Ø,Ø);

exp1:DATA(ntm,atm1,Ø,Ø,exp2);
exp2:DATA(opr,R´+´,Ø,exp4,exp3);
exp3:DATA(ntm,atm1,ADD,Ø,exp2);
exp4:DATA(opr,R´-´,Ø,exp6,exp5);
exp5:DATA(ntm,atm1,SUB,Ø,exp2);
exp6:DATA(opr,R´*´,Ø,exp8,exp7);
exp7:DATA(ntm,atm1,MUL,Ø,exp2);
exp8:DATA(opr,R´/´,Ø,exp1Ø,exp9);
exp9:DATA(ntm,atm1,DIV,Ø,exp2);
exp1Ø:DATA(opr,R´\´,Ø,exp12,exp11);
exp11:DATA(ntm,atm1,REM,Ø,exp2);
exp12:DATA(opr,R´&´,Ø,exp14,exp13);
exp13:DATA(ntm,atm1,AND,Ø,exp2);
exp14:DATA(opr,R´|´,Ø,exp16,exp15);
exp15:DATA(ntm,atm1,IOR,Ø,exp2);
exp16:DATA(opr,R´=´,Ø,exp18,exp17);
exp17:DATA(ntm,atm1,EQV,Ø,exp2);
exp18:DATA(opr,R´#´,Ø,exp2Ø,exp19);
exp19:DATA(ntm,atm1,XOR,Ø,exp2);
exp2Ø:DATA(opr,R´~´,Ø,exp22,exp21);
exp21:DATA(ntm,atm1,SHF,Ø,exp2);
exp22:DATA(Ø,Ø,Ø,Ø,Ø);

vfd1:DATA(opr,R´<´,Ø,Ø,vfd2);
vfd2:DATA(ntm,exp1,Ø,Ø,vfd3);
vfd3:DATA(opr,R´>´,SIZE,Ø,vfd4);
vfd4:DATA(ntm,exp1,FIELD,Ø,vfd5);
vfd5:DATA(opr,R´<´,Ø,vfd9,vfd6);
```

```
vfd6:DATA(ntm,expl,0,0,vfd7);
vfd7:DATA(opr,R´>´,SIZE,0,vfd8);
vfd8:DATA(ntm,expl,FIELD,0,vfd5);
vfd9:DATA(opr,R´,´,0,vfd11,vfd10);
vfd10:DATA(ntm,expl,FIELD,0,vfd5);
vfd11:DATA(0,0,0,0,0);

atm1:DATA(nam,0,NAME,atm2,0);
atm2:DATA(num,0,NUMBER,atm3,0);
atm3:DATA(stg,0,STRING,atm4,0);
atm4:DATA(opr,R´~´,0,atm6,atm5);
atm5:DATA(ntm,atm1,NOT,0,0);
atm6:DATA(opr,R´|´,0,atm8,atm7);
atm7:DATA(ntm,atm1,ABV,0,0);
atm8:DATA(opr,R´-´,0,atm10,atm9);
atm9:DATA(ntm,atm1,NEG,0,0);
atm10:DATA(opr,R´+´,0,atm12,atm11);
atm11:DATA(ntm,atm1,0,0,0);
atm12:DATA(opr,R´(´,0,0,atm13);
atm13:DATA(ntm,expl,0,0,atm14);
atm14:DATA(opr,R´)´,0,0,0) %%%
```

```
# ISP 3.6                                                  16-Sep-75 #

                               # AJP21A #
                            CALL ME SEMTIC;


# SEMTIC.I10[22,54]=AJP21A.ISP[22,54]    16-Sep-75   00:10 #

LET OPCDSIZE=16, INSTSIZE=16, BYTESIZE=16;

# CHR: special character codes #
 LET CR=15B, LF=12B, FF=14B, EOL=12B, EOF=177B;

# LEX: lexeme format # LET
 LTYP=TOK,       # lexical type #
 LADR=TOK[1],    # symbol table pointer #
 LLEN=TOK[2],    # character length #
 LVAL=TOK[3];    # value (char-array) #
 TOK IS COMMON;  # owned by LEX #

# LEX: lexical types #
 LET OPR=1, NAM=2, NUM=3, STG=4;

# SEMTIC: directory types #
 LET MOP=1, SOP=2, LCA=3, LCB=4, PRM=5, VBL=6;

# INTRPT: directory modes #
 LET UDF=0, ABL=1, REL=2, EXT=3;

# SYMBOL: directory access functions #
 <VBL> ::= <VBL,A>.TYP ::= "[1][A]"; # type   #
 <VBL> ::= <VBL,A>.FMT ::= "[2][A]"; # format #
 <VBL> ::= <VBL,A>.MOD ::= "[3][A]"; # mode   #
 <VBL> ::= <VBL,A>.VAL ::= "[4][A]"; # value  #
 <VBL> ::= <VBL,A>.LEN ::= "[5][A]"; # length #
 <VBL> ::= <VBL,A>.SYM ::= "[6][A]"; # symbol #

PASS,DOT,DIT ARE COMMON;

SUBR ISEM() IS
  (DOT => (DOT.MOD_ABL; DOT.VAL_IDOT);
   DIT => (DIT.MOD_ABL; DIT.VAL_IDIT);
   ISTK(); SAVDOT_0; 0);

PASS:DATA(0); DOT:DATA(0); DIT:DATA(0);

SUBR HASH() IS
  (LTYP=NAM =>
     (LADR_SEARCH(LVAL);
      KEY AND 1 LS LADR.TYP => LTYP_LADR.FMT);
   0);

KEY:DATA((1 LS MOP) OR 1 LS SOP);
```

```
SUBR MNEMON() IS (PTR_SEARCH(LVAL); Ø);

SUBR BOOT() IS
  (POP(TMP,VAL); POP(TMP,MOD);
   POP(TMP,FMT); POP(TMP,TYP);
   PASS=Ø =>
     (TYP=LCA => (DOT_PTR; IDOT_VAL);
      TYP=LCB => (DIT_PTR; IDIT_VAL);
      PTR.TYP_TYP; PTR.FMT_FMT;
      PTR.MOD_MOD; PTR.VAL_VAL);
   Ø);

SUBR FORMAT() IS (FMT_SEARCH(LVAL); Ø);

SUBR OPDEF() IS
  (POP(MOD,VAL);
   PASS=Ø =>
     (TMP_1; MOD=UDF => (PUTERR(R´U´); TMP_Ø);
      FMT.TYP NE MOP => (PUTERR(R´F´); TMP_Ø);
      PTR.TYP NE UDF => (PUTERR(R´M´); TMP_Ø);
      VAL RS OPCDSIZE => PUTERR(R´T´);
      TMP =>
        (PTR.TYP_MOP; PTR.FMT_FMT.FMT; PTR.MOD_ABL;
         PTR.VAL_VAL AND (1 LS OPCDSIZE)-1))
   ELSE PUTDAT(Ø,PTR.VAL); Ø);

SUBR LABEL() IS
  (BLINE(BYTESIZE);
   PASS=Ø =>
     (PTR.TYP=UDF =>
        (PTR.TYP_VBL;
         PTR.MOD_DOT.MOD;
         PTR.VAL_DOT.VAL)
      ELSE PUTERR(R´M´))
   ELSE
     (PTR.VAL NE DOT.VAL =>
         PUTERR(R´P´);
      PUTADR(DOT.VAL));
   Ø);

SUBR EQUATE() IS
  (POP(MOD,VAL);
   PASS=Ø => PTR.TYP=UDF => PTR.TYP_PRM;
   PTR.TYP=PRM =>
     (PTR.MOD_MOD; PTR.VAL_VAL;
      MOD=UDF => PUTERR(R´U´);
      PUTDAT(Ø,VAL))
   ELSE PASS => PUTERR(R´M´);
   Ø);

SUBR ORIGIN() IS
  (POP(MOD,VAL) => (MOD_ABL; VAL_SAVDOT);
   BLINE(BYTESIZE);
   MOD=UDF => PUTERR(R´U´)
```

```
   ELSE (SAVDOT_DOT.VAL; DOT.VAL_VAL);
   Ø);

SUBR ALIGN() IS
  (POP(MOD,VAL);
   BLINE(BYTESIZE);
   PASS => PUTDAT(Ø,VAL);
   MOD=ABL =>
     (VAL>Ø => ALINE(VAL)
      ELSE PUTERR(R´T´))
   ELSE PUTERR(R´U´);
   Ø);

SUBR PAD() IS
  (POP(MOD,VAL);
   PASS => PUTDAT(Ø,VAL);
   MOD=ABL =>
     (VAL>BYTESIZE =>
         (VAL_BYTESIZE; PUTERR(R´T´));
      VAL>Ø => BLINE(VAL)
      ELSE PUTERR(R´T´))
   ELSE PUTERR(R´U´);
   Ø);

SUBR SIZE() IS
  (POP(MOD,VAL); MOD=UDF => PUTERR(R´U´);
   VAL<Ø => (VAL_Ø; PUTERR(R´V´));
   SIZ_VAL; Ø);

SUBR FIELD() IS
  (POP(MOD,VAL);
   PASS =>
     (DIT.VAL=BYTESIZE => PUTADR(DOT.VAL);
      MOD=UDF => PUTERR(R´U´);
      TMP_VAL ARS SIZ => NOT TMP => PUTERR(R´T´);
      PUTDAT(SIZ,VAL));
   FILTER(SIZ,VAL); Ø);

SUBR BEGIN() IS
  (POP(MOD,VAL);
   PASS =>
     (PUTDAT(Ø,VAL);
      MOD=ABL => GOALOD(VAL)
      ELSE PUTERR(R´U´));
   Ø);

SUBR TITLE() IS (PUTITL(LVAL); Ø);

SUBR PAGE() IS (PASS => PUTPAG(); Ø);

SUBR LINE() IS (PUTLIN(); Ø);

SUBR END() IS
  (DIT.VAL NE BYTESIZE =>
    BLINE(BYTESIZE);
  PASS_NOT PASS; 0);

SUBR ERROR() IS
  (WHILE (LTYP NE OPR) OR
        (LVAL NE EOL) AND
        (LVAL NE EOF) DO LEX();
  PASS => PUTERR(R'S'); ISTK();
  LINE(); LVAL NE EOF => LEX();
  TMP_(LVAL=EOF) => END(); TMP) %%
```

```
                        # AJP21A #
                      CALL ME CODE;


# CODE.I10[22,54]=AJP21A.ISP[22,54]        16-Sep-75   00:10 #

# IMP: absolute value function #
 <ATOM> ::= ABS <ATOM,A>
        ::= DEWOP(214B,AREG1(1,15B),A);

# LEX: lexeme format # LET
 LTYP=TOK,        # lexical type #
 LADR=TOK[1],     # symbol table pointer #
 LLEN=TOK[2],     # character length #
 LVAL=TOK[3];     # value (char-array) #
 TOK IS COMMON; # owned by LEX #

# INTRPT: directory modes #
 LET UDF=0, ABL=1, REL=2, EXT=3;

# SYMBOL: directory access functions #
 <VBL> ::= <VBL,A>.TYP ::= "[1][A]"; # type   #
 <VBL> ::= <VBL,A>.FMT ::= "[2][A]"; # format #
 <VBL> ::= <VBL,A>.MOD ::= "[3][A]"; # mode   #
 <VBL> ::= <VBL,A>.VAL ::= "[4][A]"; # value  #
 <VBL> ::= <VBL,A>.LEN ::= "[5][A]"; # length #
 <VBL> ::= <VBL,A>.SYM ::= "[6][A]"; # symbol #

# SEM: instruction field sizes # LET
 NOP=0B, BYTES=1, BYTESIZE=16, INSTSIZE=16, WORDSIZE=16;

PASS,DOT,DIT ARE COMMON;

SUBR PROLOG() IS
  (BLINE(BYTESIZE);
   PASS =>
     (PUTADR(DOT.VAL);
      I_LADR.VAL);
   0);

I:DATA(0);

SUBR SETa() IS
  (POP(M,V); PASS =>
     (M=UDF => PUTERR(R'U');
      T V ARS 10 => NOT T => PUTERR(R'T');
      I[0]<10,0>_V);
   0);

SUBR SET1xb() IS
  (PASS => (I[0]<1,10>_1); 0);
```

```
SUBR SETlib() IS
  (PASS => (I[Ø]<1,11>_1); Ø);

SUBR EPILOG() IS
  (PASS =>
     (PUTDAT(INSTSIZE,I);
      BYTLOD(DOT.VAL,I);
      I_Ø);
   DOT.VAL_DOT.VAL+1; Ø);

SUBR DATA() IS
  (BLINE(BYTESIZE);
   POP(M,V); PASS =>
     (PUTADR(DOT.VAL);
      M=UDF => PUTERR(R'U');
      T_V ARS WORDSIZE =>
        NOT T => PUTERR(R'T');
      V_V AND (1 LS WORDSIZE)-1;
      PUTDAT(WORDSIZE,V);
      BYTLOD(DOT.VAL,V));
   DOT.VAL_DOT.VAL+1; Ø);

SUBR ALINE(N) IS
  (FIL_DOT.VAL//N =>
      DOT.VAL_DOT.VAL+FIL_N-FIL;
   FIL);

SUBR BLINE(N) IS
  (FIL_DIT.VAL//N => FILTER(FIL,Ø); FIL);

SUBR FILTER(S,V) IS
  (POS_DIT.VAL; SIZ_S; VAL_V;
   WHILE SIZ>Ø DO
     (BTS_SIZ>POS => BTS_POS;
      PASS =>
        (MSK_(1 LS SIZ)-1; VAL_VAL AND MSK;
         II_II OR VAL LS POS-SIZ);
      POS_POS-BTS=Ø =>
        (PASS => (BYTLOD(DOT.VAL,II); II_Ø);
         DOT.VAL_DOT.VAL+1; POS_BYTESIZE);
      SIZ_SIZ-BTS);
   DIT.VAL_POS; Ø);

II:DATA(Ø) %%
```

```
#************************************************************
*                                                           *
*                Assembly Time Interpreter                  *
*                                                           *
************************************************************#

# INTRPT.I1Ø   Ø3-Nov-73   1Ø-Sep-75 #    CALL ME INTRPT;

# IMP: absolute value function #
 <ATOM> ::= ABS <ATOM,A> ::= DEWOP(214B,AREG1(1,15B),A);

# LEX: lexeme format # LET
 LTYP=TOK,       # lexical type #
 LADR=TOK[1],    # symbol table pointer #
 LLEN=TOK[2],    # character length #
 LVAL=TOK[3];    # value (char-array) #
 TOK IS COMMON;  # owned by LEX #

# INTRPT: semantic stack types (modes) #
 LET UDF=Ø, ABL=1, REL=2, EXT=3;

# SYMBOL: directory access functions #
 <VBL> ::= <VBL,A>.TYP ::= "[1][A]";    # type   #
 <VBL> ::= <VBL,A>.FMT ::= "[2][A]";    # format #
 <VBL> ::= <VBL,A>.MOD ::= "[3][A]";    # mode   #
 <VBL> ::= <VBL,A>.VAL ::= "[4][A]";    # value  #
 <VBL> ::= <VBL,A>.LEN ::= "[5][A]";    # length #
 <VBL> ::= <VBL,A>.SYM ::= "[6][A]";    # symbol #

PASS IS COMMON;

# routines to get a value on the stack #

SUBR NAME() IS
  (LADR_SEARCH(LVAL);
   PUSH(LADR.MOD,LADR.VAL); Ø);

SUBR NUMBER() IS
  (STGIGR(LVAL,VAL) =>
      PASS => PUTERR(R'N');
   PUSH(ABL,VAL); Ø);

SUBR STRING() IS
  (STGCHR(LVAL,VAL) =>
      PASS => PUTERR(R'T');
   PUSH(ABL,VAL); Ø);

# routines to do unary and binary operations #

SUBR ABV() IS
  (POP(TYP,VAL); VAL_ABS VAL; PUSH(TYP,VAL); Ø);

SUBR NEG() IS
  (POP(TYP,VAL); VAL_-VAL; PUSH(TYP,VAL); Ø);
```

```
SUBR NOT() IS
  (POP(TYP,VAL);  VAL_NOT VAL; PUSH(TYP,VAL); 0);

SUBR ADD() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1+VAL2;
   PUSH(TYP,VAL); 0);

SUBR SUB() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1-VAL2;
   PUSH(TYP,VAL); 0);

SUBR MUL() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1*VAL2;
   PUSH(TYP,VAL); 0);

SUBR DIV() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1/VAL2;
   PUSH(TYP,VAL); 0);

SUBR REM() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1//VAL2;
   PUSH(TYP,VAL); 0);

SUBR AND() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1 AND VAL2;
   PUSH(TYP,VAL); 0);

SUBR IOR() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1 OR VAL2;
   PUSH(TYP,VAL); 0);

SUBR XOR() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1 XOR VAL2;
   PUSH(TYP,VAL); 0);

SUBR EQV() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1 EQV VAL2;
   PUSH(TYP,VAL); 0);

SUBR SHF() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1 LS VAL2;
   PUSH(TYP,VAL); 0);
```

```
SUBR ASH() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1 ALS VAL2;
   PUSH(TYP,VAL); 0);

SUBR ROT() IS
  (POP(TYP2,VAL2); POP(TYP1,VAL1);
   TYP_TYP1 AND TYP2; VAL_VAL1 LROT VAL2;
   PUSH(TYP,VAL); 0);


#*************************************************************
*                                                            *
*                        Stack Module                        *
*                                                            *
*************************************************************#

# STACK.I10   18-JAN-74   10-Sep-75 #  # CALL ME STACK; #

LET SL=20, SN=2;   STK IS SL*SN LONG;

SUBR ISTK() IS SP_-1;

SUBR PUSH(A,B) IS
  (SP GE (SL*SN)-1 => RETURN NOT 0;
   SP_SP+1; STK[SP]_B; SP_SP+1; STK[SP]_A; 0);

SUBR POP(A,B) IS
  (SP<0 => RETURN NOT 0;
   A_STK[SP]; SP_SP-1; B_STK[SP]; SP_SP-1; 0) %%
```

```
                        # AJP21A #
                     CALL ME SYMBOL;


# SYMBOL.I10[22,54]=AJP21A.ISP[22,54]    16-Sep-75   00:10 #

# IMP: absolute value function #
 <ATOM> ::= ABS <ATOM,A>
        ::= DEWOP(214B,AREG1(1,15B),A);

# directory access functions # LET BLN=6;
 <VBL> ::= <VBL,A>.LNK ::= "[A]";
 <VBL> ::= <VBL,A>.LEN ::= "[BLN-1][A]";
 <VBL> ::= <VBL,A>.SYM ::= "[BLN][A]";

# hash table definition #
 LET HLNG=71; HTAB IS HLNG LONG;

SUBR SEARCH(N) IS
  (LNG,SUM,TMP ARE REGISTER;
   LNG_0; SUM_0;
   WHILE TMP_N[LNG] DO
     (TMP_TMP LROT LNG;
      SUM_SUM XOR TMP;
      LNG_LNG+1);
   HSH_LOC(HTAB)+ABS(SUM)//HLNG;
   TMP,SUM ARE RELEASED;
   PTR IS REGISTER; PTR_[HSH];
   WHILE PTR DO
     (LNG=PTR.LEN =>
        (OLD,NEW,I ARE REGISTER; I_LNG;
         OLD_LOC(PTR.SYM); NEW_LOC(N);
         WHILE (I>0) AND ([OLD]=[NEW]) DO
            (OLD_OLD+1; NEW_NEW+1; I_I-1);
         I=0 => RETURN PTR;
         OLD,NEW,I ARE RELEASED);
      PTR_PTR.LNK);
   PTR IS SCRATCH;
   PTR_ALLOC(LNG+1+BLN);
   PTR IS PROTECTED;
   OLD,NEW,I ARE REGISTER;
   OLD_LOC(PTR.SYM); NEW_LOC(N);
     ([OLD]_[NEW]; OLD_OLD+1; NEW_NEW+1)
   FOR I FROM LNG;
   OLD,NEW,I ARE RELEASED;
   PTR.LEN_LNG; PTR.LNK_[HSH]; [HSH]_PTR);
   PTR,LNG ARE RELEASED %%
```

```
                        # AJP21A #
                     CALL ME LISTER;


# LISTER.I10[22,54]=AJP21A.ISP[22,54]    16-Sep-75   00:10 #

# number bases #
 LET BASE=10, RADIX=4, POWER=2;

# buffer and vector sizes #
 LET LINESIZE=120, PAGESIZE=60;

# special character codes #
 LET CR=15B, LF=12B, FF=14B, EOL=12B, EOF=177B;

# field definitions for listing #
LET ERRLOC=0,   ERRLEN=5,   ERRLIM=5,
    ADRLOC=6,   ADRLEN=5,   ADRLIM=11,
    DATLOC=13,  DATLEN=8,   DATLIM=21,
    CNTLOC=22,  CNTLEN=4,   CNTLIM=26,
    CHRLOC=27,  CHRLEN=93,  CHRLIM=LINESIZE,
    PAGLOC=11,  PAGLEN=3,   PAGLIM=14,
    TILLOC=16,  TILLEN=64,  TILLIM=LINESIZE;

PASS,ERRFLG ARE COMMON;

SUBR ILST() IS
  (PASS=0 => PAGCNT_0; LINCNT_0;
   TILPTR_PAGLIM; ERRPTR_ERRLOC;
   ADRPTR_ADRLOC; DATPTR_DATLOC; CHRPTR_CHRLOC;
   STMNUM_1; ANY_0; FLAG_0);

SUBR PUTITL(S) IS
  (LEN_STGLEN(S)>TILLEN => LEN_TILLEN;
   TITLE[TILLOC+I]_S[I] FOR I FROM LEN-1;
   TILPTR_TILLOC+LEN; 0);

SUBR PUTERR(C) IS
  (ERRFLG_ERRFLG+1; FLAG_1;
   ERRPTR GE ERRLIM => RETURN NOT 0;
   LINE[ERRPTR] C; ERRPTR_ERRPTR+1;
   ERRPTR>ANY => ANY_ERRPTR; 0);

SUBR PUTADR(A) IS
  (ADRPTR>ADRLOC => RETURN NOT 0;
   DATPTR>DATLOC => RETURN NOT 0;
   BINSTG(A,LINE[ADRPTR],ADRLEN);
   ADRPTR_ADRPTR+ADRLEN;
   ADRPTR>ANY => ANY_ADRPTR; 0);

SUBR PUTDAT(B,D) IS
  (LEN_B=0 => LEN_BITLEN(D);
   LEN_1+(LEN-1)/POWER;
   DATPTR+LEN>DATLIM => RETURN NOT Ø;
   BINSTG(D,LINE[DATPTR],LEN);
   DATPTR_DATPTR+LEN; DATPTR_DATPTR+1;
   DATPTR>ANY => ANY_DATPTR; Ø);

SUBR PUTCHR(C) IS
  (CHRPTR GE CHRLIM => RETURN NOT Ø;
   C GE R´ ´ =>
     (LINE[CHRPTR]_C; CHRPTR_CHRPTR+1;
      CHRPTR>ANY => ANY_CHRPTR);
   Ø);

SUBR PUTLIN() IS
  (PASS OR FLAG =>
     (LINCNT=Ø =>
        (PAGCNT>Ø => LIST(FF); PAGCNT_PAGCNT+1;
         DECSTG(PAGCNT,TITLE[PAGLOC],PAGLEN);
         LIST(TITLE[I]) FOR I TO TILPTR-1;
         LIST(CR); LIST(LF); LIST(LF); LINCNT_2);
      CHRPTR>CHRLOC =>
         DECSTG(STMNUM,LINE[CNTLOC],CNTLEN);
      ANY>Ø => (LIST(LINE[I]) FOR I TO ANY-1);
      LIST(CR); LIST(LF);
      LINCNT_LINCNT+1 GE PAGESIZE => LINCNT_Ø);
   CHRPTR>CHRLOC => STMNUM_STMNUM+1;
   ANY>Ø => (LINE[I]_R´ ´ FOR I FROM ANY-1);
   ERRPTR_ERRLOC; ADRPTR_ADRLOC; DATPTR_DATLOC;
   CHRPTR_CHRLOC; ANY_Ø; FLAG_Ø);

SUBR PUTPAG() IS (LINCNT_PAGESIZE; Ø);

TITLE:
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´P´,R´a´);
   DATA(R´g´,R´e´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);

LINE:
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
```

```
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´);
   DATA(R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´,R´ ´) %%
```

```
#*********************************************************
*                                                        *
*                  Character Output Module               *
*                                                        *
**********************************************************

OPENL(DEV,FIL,EXT,PRJ,PRG)
   Effect: Initialize a file for output.
   Parameters:
      DEV - Device name terminated by a null byte.
      FIL - File name terminated by a null byte.
      EXT - File extension terminated by a null byte.
      PRJ - Project number (octal); if zero, this UFD.
      PRG - Programmer number (octal); if zero, this UFD.
   Value: An error code (usually no-room), zero if ok.
   Errors: None.

LIST(CHR)
   Effect: Advances the current file by one character.
   Parameters:
      STG - The character to be added to the file.
   Value: An error code (usually I/O-error), zero if ok.
   Errors: None.

CLOSEL()
   Effect: Terminates output processing.
   Parameters: None.
   Value: None.
   Errors: None.  #

# LIST.I10   16-Nov-72   10-Sep-75 #   CALL ME LIST;

LET MODE=0, BYTSIZ=0, LEN=128, NBUF=2;
OBUFS IS NBUF*LEN+3 LONG; !.JBFF! IS COMMON;

SUBR OPENL(DEV,FIL,EXT,PRJ,PRG) IS
  (CH_GETCHA(0)<0 => RETURN CH;
   E_INIT(CH,MODE,DEV,OBUF,'0') => RETURN E;
   E_ENTER(CH,FIL,EXT,PRJ,PRG) => (RELEA(CH); RETURN E);
   SAVJBF_!.JBFF!<R>; !.JBFF!<R>_LOC(OBUFS);
   OUTBUF(CH,NBUF); !.JBFF!<R>_SAVJBF;
   BYTSIZ => OBUF[1]<6,24>_BYTSIZ;
   0);   OBUF:DATA(0,0,0);

SUBR LIST(CHR) IS
  (OBUF[2]_OBUF[2]-1 LE 0 =>
     (E_OUTPUT(CH) => RETURN E);
   <+OBUF[1]>_CHR;
   0);

SUBR CLOSEL() IS RELEA(CH) %%
```

```
# ISP 3.6                                                  16-Sep-75 #

                           # AJP21A #
                         CALL ME LOADER;

# LOADER.I10[22,54]=AJP21A.ISP[22,54]    16-Sep-75   00:10 #

LET
   BYTESIZE=16, # bits per byte        #
   BYTES=1,     # bytes per address    #
   COD=1,       # id of code block     #
   BS=32,       # maximum block size   #
   HS=3,        # header size          #
   B=BUF[HS];   # beginning of data    #

BUF IS HS+BS LONG;

# Load file format:                    #
#                                      #
#   BUF[0]                 block-id    #
#   BUF[1:BYTES]           byte count  #
#   BUF[BYTES+1:HS-1]      block origin #
#   BUF[HS:BS-1]           data bytes  #
#   BUF[BS]                checksum    #
#                                      #
# where checksum = -sum(BUF[0:BS-1]),  #
# truncated on the left to BYTESIZE.   #

SUBR BEGLOD() IS (BUF[0]_COD; GO_BC_0);

SUBR BYTLOD(ADR,VAL) IS
  (BC=0 => BO_ADR;
   (BC GE BS) OR (BO+BC NE ADR) =>
     (DMPLOD(); BO_ADR);
   B[BC]_VAL; BC_BC+1; 0);

SUBR DMPLOD() IS
  (BC_BC+HS; CK_0;
   BUF[1]_BC; BUF[2]_BO;
     (LOAD(BUF[I]); CK_CK+BUF[I])
   FOR I TO BC-1;
   LOAD(-CK AND (1 LS BYTESIZE)-1);
   BC_0);

SUBR GOALOD(ADR) IS (GO_ADR; 0);

SUBR ENDLOD() IS
  (BC>0 => DMPLOD();
   BO_GO; DMPLOD(); 0) %%
```

```
#***********************************************************
*                                                         *
*                 Binary Output Module                    *
*                                                         *
***********************************************************

OPENO(DEV,FIL,EXT,PRJ,PRG)
   Effect: Initialize a file for output.
   Parameters:
      DEV - Device name terminated by a null byte.
      FIL - File name terminated by a null byte.
      EXT - File extension terminated by a null byte.
      PRJ - Project number (octal); if zero, this UFD.
      PRG - Programmer number (octal); if zero, this UFD.
   Value: An error code (usually no-room), zero if ok.
   Errors: None.

LOAD(NXT)
   Effect: Advances the current file to the next word.
   Parameters:
      NXT - The word to be added to the file.
   Value: An error code (usually I/O-error), zero if ok.
   Errors: None.

CLOSEO()
   Effect: Terminates output processing.
   Parameters: None.
   Value: None.
   Errors: None.  #

# LOAD.I10   16-Nov-72   10-Sep-75 #   CALL ME LOAD;

LET MODE=10B, BYTSIZ=0, LEN=128, NBUF=2;
OBUFS IS NBUF*LEN+3 LONG; !.JBFF! IS COMMON;

SUBR OPENO(DEV,FIL,EXT,PRJ,PRG) IS
  (CH_GETCHA(0)<0 => RETURN CH;
   E_INIT(CH,MODE,DEV,OBUF,'0') => RETURN E;
   E_ENTER(CH,FIL,EXT,PRJ,PRG) => (RELEA(CH); RETURN E);
   SAVJBF_!.JBFF!<R>; !.JBFF!<R>_LOC(OBUFS);
   OUTBUF(CH,NBUF); !.JBFF!<R>_SAVJBF;
   BYTSIZ => OBUF[1]<6,24>_BYTSIZ;
   0);   OBUF:DATA(0,0,0);

SUBR LOAD(NXT) IS
  (OBUF[2]_OBUF[2]-1 LE 0 =>
     (E_OUTPUT(CH) => RETURN E);
   <+OBUF[1]>_NXT; 0);

SUBR CLOSEO() IS RELEA(CH) %%
```

```
#***********************************************************
*                                                         *
*               Storage Allocation Module                 *
*                                                         *
***********************************************************

ALLOC(N)
   Effect: Allocates N words (expanding core if necessary).
   Parameters: N - Number of words to allocate.
   Value: Address of N word block available to the caller.
   Errors: Free storage exhausted.

DEALOC(A)
   Effect: Deallocates addresses greater than or equal to
      A (contracting the low segment if necessary).
   Parameters: A - Low-order address for deallocation.
   Value: None.
   Errors: Free storage exhausted.

GLOBALS REFERENCED:
   .JBREL - hicore of low segment (set by monitor).
   .JBFF  - first free location in low segment.  #

# STORAG.I10   02-Mar-72   10-Sep-75 #   CALL ME STORAG;

 <ATOM> ::= EXIT ::= DEWOP(047B,0,ADDR(STACKUP(12B)));
 <ATOM> ::= OUTSTR <EXP,A>  ::= DEWOP(051B,AREG(3),A);
 <ATOM> ::= CORE(<EXP,A>,<VBL,B>) ::= HOOK(A,
            ADDOP(047B,ADDR(STACKUP(11B)),
               DEWOP(200B,AREG(0),A)),
            FREEZE(DEWOP(254B,AREG(0),B)));

SUBR ALLOC(N) IS
  (!.JBREL!,!.JBFF! ARE COMMON;
   PTR,TMP ARE REGISTER;
   PTR_!.JBFF!<R>; TMP_PTR+N;
   TMP_TMP-1>!.JBREL!<R> => CORE(TMP,E);
   TMP_TMP+1; !.JBFF!<R>_TMP;
   WHILE TMP_TMP-1 GE PTR DO [TMP]_0;
   PTR); PTR,TMP ARE RELEASED;

SUBR DEALOC(A) IS
  (PTR IS REGISTER; PTR_A;
   !.JBFF!<R>_PTR; PTR_PTR-1;
   !.JBREL!<R>-PTR GE 1024 => CORE(PTR,E);
   0); PTR IS RELEASED;

E:OUTSTR('?Out of core'); EXIT %%
```

```
? Bootstrap AJP21A.ISP   16-Sep-75   00:10
? This machine uses base 4. for constants.

'..'        4.  0. 1. 16.
'.'         3.  0. 1.  0.
'PAGE'      2.  5. 0.  0.
'TITLE'     2.  6. 0.  0.
'BEGIN'     2.  7. 0.  0.
'ALIGN'     2.  8. 0.  0.
'ORG'       2.  9. 0.  0.
'OPDEF'     2. 10. 0.  0.
'FIELD'     2. 11. 0.  0.
'PAD'       2. 12. 0.  0.
'DATA'      2. 14. 0.  0.
'.fmt20'    1. 20. 1.  0.   ? .fmt20: ib xb a
'.fmt21'    1. 21. 1.  0.   ? .fmt21: (null)
PAGE
TITLE     'AJP21A Op-code Definitions'

OPDEF     'GTR'(.fmt20)=36864.     ? .fmt20: ib xb a
OPDEF     'ADD'(.fmt20)=16384.     ? .fmt20: ib xb a
OPDEF     'SIT'(.fmt20)=8192.      ? .fmt20: ib xb a
OPDEF     'HLT'(.fmt21)=12288.     ? .fmt21: (null)
OPDEF     'STO'(.fmt20)=53248.     ? .fmt20: ib xb a
OPDEF     'DIV'(.fmt20)=28672.     ? .fmt20: ib xb a
OPDEF     'NOT'(.fmt21)=45056.     ? .fmt21: (null)
OPDEF     'LOD'(.fmt20)=49152.     ? .fmt20: ib xb a
OPDEF     'GET'(.fmt20)=57344.     ? .fmt20: ib xb a
OPDEF     'EQL'(.fmt20)=32768.     ? .fmt20: ib xb a
OPDEF     'MPY'(.fmt20)=24576.     ? .fmt20: ib xb a
OPDEF     'SUB'(.fmt20)=20480.     ? .fmt20: ib xb a
OPDEF     'JMP'(.fmt20)=0.         ? .fmt20: ib xb a
OPDEF     'AND'(.fmt20)=40960.     ? .fmt20: ib xb a
OPDEF     'JIF'(.fmt20)=4096.      ? .fmt20: ib xb a
OPDEF     'PUT'(.fmt20)=61440.     ? .fmt20: ib xb a
```

```
Page    1  Algorithm 1.2.10-M  Find the maximum [Knuth68]

                     1 TITLE 'Algorithm 1.2.10-M  Find the maximum [Knuth68]'

                     2 ? Given n elements A[1], A[2], ... ,A[n], find m and j
                     3 ? such that m = A[j] = max 0<k<n+1 A[k], and for which
                     4 ? j is as large as possible.

                     5 ?   1. [Initialize.]  Assign n -> j, n-1 -> k, A[n] -> m.
                     6 ?
                     7 ?   2. [All tested?]  If k=0, the algorithm terminates.
                     8 ?
                     9 ?   3. [Compare.]  Go to 5 unless A[k] > m.
                    10 ?
                    11 ?   4. [Change m.]  Assign k -> j, A[k] -> m.
                    12 ?
                    13 ?   5. [Decrease k.]  Decrease k by one, return to 2.

                    14 ? Assumption: n and A[1:n] are already in memory.
                    15 ? Comment: m := AC and k := X {index register}.
                    16 ? Comment: A[1:n] is stored in Mp[A+1:A+n].

         0          17 K=0      ? k is index register X := Mp[0]

                    18          ORG  100
         100        19          BEGIN here
00100    30200201   20 here:    LOD  @N
00101    31000133   21          STO  J        ? 1. n -> j
00102    31000000   22          STO  K        ?    n -> k
00103    30300201   23          LOD  @*A      ?    A[n] -> m
00110    02000131   24 loop:    SIT  done     ? 2. k-1 -> k = 0 => stop
00111    20300201   25          EQL  @*A
00112    01000110   26          JIF  loop     ? 3. m = A[k] => go to 2
00113    21300201   27          GTR  @*A
00120    01000110   28          JIF  loop     ?    m > A[k] => go to 2
00121    30200000   29          LOD  @K
```

```
Page    2  Algorithm 1.2.10-M  Find the maximum [Knuth68]

00122  31000133   30          STO  J      ? 4. k -> j
00123  30300201   31          LOD  @*A    ?    A[k] -> m
00130  00000110   32          JMP  loop   ? 5. go to 2
00131  31000200   33 done:    STO  M
00132  03000000   34          HLT

00133  00000000   35 J:       DATA  0     ? local, output
00200  00000000   36 M:       DATA  0     ? local, output
00201             37 N: A:                ? global, input
```

```
DUMP OF FILE BOOT.BIN TO DSK:BOOT.DMP        09-16-75  03:58


   000000000001   000000000003   000000000000   000000177774


DUMP OF FILE TST21A.BIN TO DSK:TST21A.DMP    09-16-75  03:59

   000000000001   000000000024   000000000020   000000144041
   000000150037   000000150000   000000146041   000000020035
   000000106041   000000010024   000000116041   000000010024
   000000144000   000000150037   000000146041   000000000024
   000000150040   000000030000   000000000000   000000000000
   000000177177
                  000000000001   000000000003   000000000020
   000000177754
```

APPENDIX D

CASE STUDY: THE PDP-8

The PDP-8 [DEC67] is an interesting machine because of its extremely small word length (12 bits); in spite of this choice, it was designed as a word-organized machine, in contrast to the byte-addressing schemes now used in most micro-processors and mini-computers with narrow data paths. To achieve the necessary flexibility in such a short word, the machine uses a partially micro-coded instruction set (section 5.1.1), and a page-relative operand addressing mechanism (section 4.3.2). It is sad but true that the methods described in those sections for dealing with these problems have not been implemented.

In particular, this version of the PDP-8 assembler does not allow combinations of micro-programmed op-codes. For the PDP-8, this is not a severe restriction. Specific micro-combinations can always be defined as separate op-codes using the OPDEF pseudo-op (see the sample programs at the end of this appendix).

A more annoying "feature" of this assembler is that it does not "understand" the page-relative addressing scheme, wherein an operand address must lie in the current page

(with respect to the PC) or in page zero. This deficiency is due entirely to the fact that the operand address function "a<Ø:11>" is not recognized as a form (see the ISP description following). Instead, the page-bit (pb) and the (relative) page-address (pa) are treated as separate operands, rather than as a single assembly-time calculation. The instruction form device described in chapter 4 would remedy this problem, but forms have not been implemented in the current version of the assembler generator.

The complete PDP-8 assembler is not included in this appendix, since the bulk of the code is similar to the examples given for the AJP-21a machine. Most of the modules differ only in the setting of compile-time parameters such as bytesize, input radix, and listing tabulations, or they differ in very short, localized code sequences. The notable exceptions, which are included following the ISP description, are the SYNTAX, CODE, and BOOT files. Finally, the sample program in appendix A has been re-coded for the PDP-8, and a listing and octal dump appear at the end of this appendix. In addition, examples of most of the pseudo-ops and instruction and data formats have been appended to the program, to illustrate those features not used in the simple maximum-finding routine.

```
                        {PDP8 LISTING}


{PDP8.LST[22,54]=PDP8.ISP[22,54]          16-Sep-75   05:21}

Mp_State

   M[0:7777b8]<0:11>
      Page_0[0:177b8]<0:11> := M[0:177b8]<0:11>
      Auto_index[0:7]<0:11> := M[10b8:17b8]<0:11>


Pc_State

   PC<0:11>
   AC<0:11>
   skip
   Run
   L

External_Pc_State

   Data_switches<0:11>


Instruction_Format

   I/Instruction[0:1]<0:11>
      op<0:2>                := I[0]<0:2>
      ib/indirect_bit        := I[0]<3>
      pb/page_bit            := I[0]<4>
      pa/page_address<0:6>   := I[0]<5:11>
      dev/IO_select<0:5>     := I[0]<3:8>
      fcn/IO_function<0:2>   := I[0]<9:11>
      opr                    := I[0]<3>
      cla                    := I[0]<4>
      cll                    := I[0]<5>
      cma                    := I[0]<6>
      cml                    := I[0]<7>
      mic<0:2>               := I[0]<8:10>
      iac                    := I[0]<11>
      eae                    := I[0]<11>
      sma                    := I[0]<5>
      sza                    := I[0]<6>
      snl                    := I[0]<7>
      sense                  := I[0]<8>
      osr                    := I[0]<9>
      hlt                    := I[0]<10>
      mqa                    := I[0]<5>
      sca                    := I[0]<6>
      mql                    := I[0]<7>
```

```
Instruction_Interpretation_Process

  'Interpret' :=
     Run =>
       (M[PC] -> I[0]; 1+PC -> PC next
        'Execute' next 'Interpret')


Effective_Address_Calculation

   PC'<0:11> := PC<0:11>-1

   page<0:4> := (~pb => 0; pb => PC'<0:4>)
   a<0:11> := page.pa

   z<0:11> := (~ib => a; ib => M[a])

   z'<0:11> :=
     (ib =>
        (a<0:7> eq 0 & a<8> =>
            1+M[a] -> M[a]
         next M[a]);
      ~ib => a)


Instruction_Execution_Process

  'Execute' :=
    ('AND' (:= op eq 000b) => M[z']&AC -> AC;
     'TAD' (:= op eq 001b) => M[z']+L.AC -> L.AC;
     'ISZ' (:= op eq 010b) =>
        (1+M[z'] -> M[z] next
         M[z] eq 0 => 1+PC -> PC);
     'DCA' (:= op eq 011b) => (AC -> M[z']; 0 -> AC);
     'JMS' (:= op eq 100b) => (PC -> M[z'] next z+1 -> PC);
     'JMP' (:= op eq 101b) => z' -> PC;
     'IOT' (:= op eq 110b) => undefined(fcn,dev);
     op eq 111b => 'Operate')


Instruction_Execution_Process {Micro_Groups}

  'Operate' :=
     (~opr =>
       ('CLA' (:= cla) => 0 -> AC;
        'CLL' (:= cll) => 0 -> L next
        'CMA' (:= cma) => ~AC -> AC;
        'CML' (:= cml) => ~L -> L next
        'IAC' (:= iac) => 1+L.AC -> L.AC next
        'RAL' (:= mic eq 010b) => L.AC*2 -> L.AC{rotate};
        'RTL' (:= mic eq 011b) => L.AC*4 -> L.AC{rotate};
     'RAR' (:= mic eq 100b) => L.AC/2 -> L.AC{rotate};
     'RTR' (:= mic eq 101b) => L.AC/4 -> L.AC{rotate});
   opr & ~eae =>
     (~sense =>
       (0 -> skip next   {"or" group}
       'SMA' (:= sma) => (AC lt 0)|skip -> skip next
       'SZA' (:= sza) => (AC eq 0)|skip -> skip next
       'SNL' (:= snl) => L|skip -> skip);
      sense =>
       (1 -> skip next   {"and" group}
       'SPA' (:= sma) => ~(AC lt 0)&skip -> skip next
       'SNA' (:= sza) => ~(AC eq 0)&skip -> skip next
       'SZL' (:= snl) => ~L&skip -> skip)
      next skip => 1+PC -> PC;
     'CLR' (:= cla) => 0 -> AC next
     'OSR' (:= osr) => Data_switches|AC -> AC;
     'HLT' (:= hlt) => 0 -> ~Run);
   opr & eae => 'EAE')


Pc_State {EAE}

  MQ<0:11>
  SC<0:4>

Instruction_Format {EAE}

  mds<0:11> := I[1]
  s<0:4>    := I[1]<7:11>


Instruction_Fetch_Process {EAE}

 'EAE_Fetch' := (M[PC] -> mds; 1+PC -> PC)


Instruction_Execution_Process {EAE}

 'EAE' :=
   ('CLE' (:= cla) => 0 -> AC next
    'MQA' (:= mqa) => AC|MQ -> AC;
    'SCA' (:= sca) => AC|SC -> AC;
    'MQL' (:= mql) => (AC -> MQ; 0 -> AC) next
    'SCL' (:= mic eq 001b) =>
      ('EAE_Fetch' next s -> SC);
    'MUY' (:= mic eq 010b) =>
      ('EAE_Fetch' next 0 -> SC;
       MQ*mds -> AC.MQ; 0 -> L);
    'DVI' (:= mic eq 011b) =>
      ('EAE_Fetch' next 0 -> SC;
       overflow(AC.MQ/mds) -> L;
       AC MQ/mds -> MQ; AC.MQ\mds -> AC);
    'NMI' (:= mic eq 100b) =>
```

```
     (normalize(AC.MQ) -> AC.MQ{arithmetic};
      norm_count(AC.MQ) -> SC{arithmetic});
   'SHL' (:= mic eq 101b) =>
     ('EAE_Fetch' next 0 -> SC;
      L.AC.MQ*(2^(s+1)) -> L.AC.MQ{logical});
   'ASR' (:= mic eq 110b) =>
     ('EAE_Fetch' next AC<0> -> L; 0 -> SC;
      AC.MQ/(2^(s+1)) -> AC.MQ{arithmetic});
   'LSR' (:= mic eq 111b) =>
     ('EAE_Fetch' next 0 -> L; 0 -> SC;
      AC.MQ/(2^(s+1)) -> AC.MQ{logical}))
```

{PDP8 SYNTAX}

```
{SYNTAX.SYN[22,54]=PDP8.ISP[22,54]        16-Sep-75   Ø5:21}

.opr=1;         .nam=2;         .num=3;         .stg=4;
.pag=5;         .til=6;         .beg=7;         .aln=8;
.org=9;         .def=1Ø;        .vfd=11;        .pad=12;
.fup=13;        .dat=14;        .fmt2Ø=2Ø;      .fmt21=21;
.fmt22=22;      .fmt23=23;      .fmt24=24;      .fmt25=25;

        opr=.opr;        num=.opr;        stg=.opr;

<prg> ::= <stl> 177b [END];

<stl> ::= 12b             [LINE] |
          <stm> 12b       [LINE] |
          <stl> 12b       [LINE] |
          <stl> <stm> 12b [LINE] ;

<stm> ::=
   [HASH] .nam [MNEMON] ':' [LABEL] |
   [HASH] .nam [MNEMON] ':' [LABEL] <stm> |
   [HASH] .nam [MNEMON] '=' <exp> [EQUATE];

<vfd> ::= '<' <exp> '>' [SIZE] <exp> [FIELD] |
    <vfd> '<' <exp> '>' [SIZE] <exp> [FIELD] |
    <vfd> ',' <exp> [FIELD];

<exp> ::= <atm> |
   <exp> '+' <atm> [ADD] | <exp> '-' <atm> [SUB] |
   <exp> '*' <atm> [MUL] | <exp> '/' <atm> [DIV] |
   <exp> '\' <atm> [REM] | <exp> '&' <atm> [AND] |
   <exp> '|' <atm> [IOR] | <exp> '=' <atm> [EQV] |
   <exp> '#' <atm> [XOR] | <exp> '~' <atm> [SHF] ;

<atm> ::=
   .nam [NAME] | .num [NUMBER] | .stg [STRING] |
   '~' <atm> [NOT] | '|' <atm> [ABV] |
   '-' <atm> [NEG] | '+' <atm> |
   '(' <exp> ')' ;

<stm> ::= [HASH] .fmt2Ø [PROLOG]
          <exp> [SETs] [EPILOG];

<stm> ::= [HASH] .fmt21 [PROLOG]
          <exp> [SETmds] [EPILOG];

<stm> ::= [HASH] .fmt22 [PROLOG] [EPILOG];

<stm> ::= [HASH] .fmt23 [PROLOG]
          '%' <exp> [SETpa] [EPILOG];
```

```
<stm> ::= [HASH] .fmt23 [PROLOG]
          '' [SET1pb] <exp> [SETpa] [EPILOG];
<stm> ::= [HASH] .fmt23 [PROLOG]
          '@' [SETlib] '%' <exp> [SETpa] [EPILOG];
<stm> ::= [HASH] .fmt23 [PROLOG]
          '@' [SETlib] '' [SET1pb] <exp> [SETpa] [EPILOG];

<stm> ::= [HASH] .fmt24 [PROLOG] [EPILOG];

<stm> ::= [HASH] .fmt25 [PROLOG]
          <exp> [SETdev] ,
          <exp> [SETfcn] [EPILOG];

<stm> ::=
   [HASH] .org ''     [ORIGIN] |
   [HASH] .org <exp> [ORIGIN] |
   [HASH] .dat <exp> [DATA]   |
   [HASH] .vfd <vfd> |
   [HASH] .aln <exp> [ALIGN]  |
   [HASH] .pad <exp> [PAD]    |
   [HASH] .pag       [PAGE]   |
   [HASH] .til .stg  [TITLE]  |
   [HASH] .beg <exp> [BEGIN]  |
   [HASH] .def .stg  [MNEMON]
     '('.nam [FORMAT]')' '=' <exp> [OPDEF] |
   .stg [MNEMON] <exp> <exp> <exp> <exp> [BOOT];
```

```
                        # PDP8 #
                      CALL ME CODE;


# CODE.I1Ø[22,54]=PDP8.ISP[22,54]        16-Sep-75   Ø5:21 #

# IMP: absolute value function #
 <ATOM> ::= ABS <ATOM,A>
        ::= DEWOP(214B,AREG1(1,15B),A);

# LEX: lexeme format # LET
 LTYP=TOK,       # lexical type #
 LADR=TOK[1],    # symbol table pointer #
 LLEN=TOK[2],    # character length #
 LVAL=TOK[3];    # value (char-array) #
 TOK IS COMMON;  # owned by LEX #

# INTRPT: directory modes #
 LET UDF=Ø, ABL=1, REL=2, EXT=3;

# SYMBOL: directory access functions #
 <VBL> ::= <VBL,A>.TYP ::= "[1][A]"; # type   #
 <VBL> ::= <VBL,A>.FMT ::= "[2][A]"; # format #
 <VBL> ::= <VBL,A>.MOD ::= "[3][A]"; # mode   #
 <VBL> ::= <VBL,A>.VAL ::= "[4][A]"; # value  #
 <VBL> ::= <VBL,A>.LEN ::= "[5][A]"; # length #
 <VBL> ::= <VBL,A>.SYM ::= "[6][A]"; # symbol #

# SEM: instruction field sizes # LET
 NOP=ØB, BYTES=1, BYTESIZE=12, INSTSIZE=12, WORDSIZE=12;

PASS,DOT,DIT ARE COMMON;

SUBR PROLOG() IS
   (DEL_LOP[LTYP-2Ø];
    BLINE(BYTESIZE);
    PASS =>
      (PUTADR(DOT.VAL);
       I[Ø]_LADR.VAL<12,Ø>);
    Ø);

DEL:DATA(Ø); I:DATA(Ø,Ø);
LOP:DATA(2,2,1,1,1,1);

SUBR SETfcn() IS
   (POP(M,V); PASS =>
      (M=UDF => PUTERR(R´U´);
       T_V ARS 3 => NOT T => PUTERR(R´T´);
       I[Ø]<3,Ø>_V);
    Ø);

SUBR SETdev() IS
```

```
   (POP(M,V); PASS =>
      (M=UDF => PUTERR(R´U´);
       T_V ARS 6 => NOT T => PUTERR(R´T´);
       I[Ø]<6,3>_V);
    Ø);

SUBR SETpa() IS
   (POP(M,V); PASS =>
      (M=UDF => PUTERR(R´U´);
       T_V ARS 7 => NOT T => PUTERR(R´T´);
       I[Ø]<7,Ø>_V);
    Ø);

SUBR SETlpb() IS
   (PASS => (I[Ø]<1,7>_1); Ø);

SUBR SETlib() IS
   (PASS => (I[Ø]<1,8>_1); Ø);

SUBR SETmds() IS
   (POP(M,V); PASS =>
      (M=UDF => PUTERR(R´U´);
       T_V ARS 12 => NOT T => PUTERR(R´T´);
       I[1]<12,Ø>_V);
    Ø);

SUBR SETs() IS
   (POP(M,V); PASS =>
      (M=UDF => PUTERR(R´U´);
       T_V ARS 5 => NOT T => PUTERR(R´T´);
       I[1]<5,Ø>_V);
    Ø);

SUBR EPILOG() IS
   (  (PASS =>
         (PUTDAT(INSTSIZE,I[J]);
          BYTLOD(DOT.VAL,I[J]);
          I[J]_Ø);
       DOT.VAL_DOT.VAL+1)
    FOR J TO DEL-1; DEL_Ø);

SUBR DATA() IS
   (BLINE(BYTESIZE);
    POP(M,V); PASS =>
      (PUTADR(DOT.VAL);
       M=UDF => PUTERR(R´U´);
       T_V ARS WORDSIZE =>
          NOT T => PUTERR(R´T´);
       V_V AND (1 LS WORDSIZE)-1;
       PUTDAT(WORDSIZE,V);
       BYTLOD(DOT.VAL,V));
    DOT.VAL_DOT.VAL+1; Ø);

SUBR ALINE(N) IS
```

```
  (FIL_DOT.VAL//N =>
      DOT.VAL_DOT.VAL+FIL_N-FIL;
   FIL);

SUBR BLINE(N) IS
  (FIL_DIT.VAL//N => FILTER(FIL,Ø); FIL);

SUBR FILTER(S,V) IS
  (POS_DIT.VAL; SIZ_S; VAL_V;
   WHILE SIZ>Ø DO
      (BTS_SIZ>POS => BTS_POS;
       PASS =>
          (MSK_(1 LS SIZ)-1; VAL_VAL AND MSK;
           II_II OR VAL LS POS-SIZ);
       POS_POS-BTS=Ø =>
          (PASS => (BYTLOD(DOT.VAL,II); II_Ø);
           DOT.VAL_DOT.VAL+1; POS_BYTESIZE);
       SIZ_SIZ-BTS);
   DIT.VAL_POS; Ø);

II:DATA(Ø) %%
```

```
? Bootstrap PDP8.ISP   23-Sep-75   18:Ø8
? This machine uses base 8. for constants.

´..´        4.  Ø. 1. 12.
´.´         3.  Ø. 1.  Ø.
´PAGE´      2.  5. Ø.  Ø.
´TITLE´     2.  6. Ø.  Ø.
´BEGIN´     2.  7. Ø.  Ø.
´ALIGN´     2.  8. Ø.  Ø.
´ORG´       2.  9. Ø.  Ø.
´OPDEF´     2. 10. Ø.  Ø.
´FIELD´     2. 11. Ø.  Ø.
´PAD´       2. 12. Ø.  Ø.
´DATA´      2. 14. Ø.  Ø.
´.fmt2Ø´    1. 2Ø. 1.  Ø.   ? .fmt2Ø: s
´.fmt21´    1. 21. 1.  Ø.   ? .fmt21: mds
´.fmt22´    1. 22. 1.  Ø.   ? .fmt22: (null)
´.fmt23´    1. 23. 1.  Ø.   ? .fmt23: ib pb pa
´.fmt24´    1. 24. 1.  Ø.   ? .fmt24: sense
´.fmt25´    1. 25. 1.  Ø.   ? .fmt25: dev fcn
PAGE
TITLE    ´PDP8 Op-code Definitions´

OPDEF    ´LSR´(.fmt2Ø)=7417  ? .fmt2Ø: s
OPDEF    ´MUY´(.fmt21)=74Ø5  ? .fmt21: mds
OPDEF    ´SCL´(.fmt2Ø)=74Ø3  ? .fmt2Ø: s
OPDEF    ´SZL´(.fmt22)=743Ø  ? .fmt22: (null)
OPDEF    ´CML´(.fmt22)=7Ø2Ø  ? .fmt22: (null)
OPDEF    ´RAR´(.fmt22)=7Ø1Ø  ? .fmt22: (null)
OPDEF    ´NMI´(.fmt22)=7411  ? .fmt22: (null)
OPDEF    ´TAD´(.fmt23)=1ØØØ  ? .fmt23: ib pb pa
OPDEF    ´HLT´(.fmt24)=74Ø2  ? .fmt24: sense
OPDEF    ´RTR´(.fmt22)=7Ø12  ? .fmt22: (null)
OPDEF    ´DCA´(.fmt23)=3ØØØ  ? .fmt23: ib pb pa
OPDEF    ´ISZ´(.fmt23)=2ØØØ  ? .fmt23: ib pb pa
OPDEF    ´CLA´(.fmt22)=72ØØ  ? .fmt22: (null)
OPDEF    ´SNL´(.fmt22)=742Ø  ? .fmt22: (null)
OPDEF    ´SHL´(.fmt2Ø)=7413  ? .fmt2Ø: s
OPDEF    ´SMA´(.fmt22)=75ØØ  ? .fmt22: (null)
OPDEF    ´DVI´(.fmt21)=74Ø7  ? .fmt21: mds
OPDEF    ´IAC´(.fmt22)=7ØØ1  ? .fmt22: (null)
OPDEF    ´CLL´(.fmt22)=71ØØ  ? .fmt22: (null)
OPDEF    ´MQA´(.fmt22)=75Ø1  ? .fmt22: (null)
OPDEF    ´SCA´(.fmt22)=7441  ? .fmt22: (null)
OPDEF    ´SZA´(.fmt22)=744Ø  ? .fmt22: (null)
OPDEF    ´ASR´(.fmt2Ø)=7415  ? .fmt2Ø: s
OPDEF    ´JMP´(.fmt23)=5ØØØ  ? .fmt23: ib pb pa
OPDEF    ´CMA´(.fmt22)=7Ø4Ø  ? .fmt22: (null)
OPDEF    ´RAL´(.fmt22)=7ØØ4  ? .fmt22: (null)
OPDEF    ´AND´(.fmt23)=ØØØØ  ? .fmt23: ib pb pa
OPDEF    ´CLE´(.fmt22)=76Ø1  ? .fmt22: (null)
OPDEF    ´OSR´(.fmt24)=74Ø4  ? .fmt24: sense
OPDEF    ´RTL´(.fmt22)=7ØØ6  ? .fmt22: (null)
OPDEF    ´SPA´(.fmt22)=751Ø  ? .fmt22: (null)
OPDEF  'JMS'(.fmt23)=4000  ? .fmt23: ib pb pa
OPDEF  'MQL'(.fmt22)=7421  ? .fmt22: (null)
OPDEF  'SNA'(.fmt22)=7450  ? .fmt22: (null)
OPDEF  'CLR'(.fmt24)=7600  ? .fmt24: sense
OPDEF  'IOT'(.fmt25)=6000  ? .fmt25: dev fcn
```

```
Page   1  Algorithm 1.2.10-M  Find the maximum [Knuth68]

                  1 TITLE 'Algorithm 1.2.10-M  Find the maximum [Knuth68]'

                  2 ? Given n elements A[1], A[2], ... ,A[n], find m and j
                  3 ? such that m = A[j] = max 0<k<n+1 A[k], and for which
                  4 ? j is as large as possible.

                  5 ?   1. [Initialize.]  Assign A[n] -> m, n -> j, n-1 -> k.
                  6 ?
                  7 ?   2. [All tested?]  If k=0, the algorithm terminates.
                  8 ?
                  9 ?   3. [Compare.]  Go to 5 unless A[k] > m.
                 10 ?
                 11 ?   4. [Change m.]  Assign A[k] -> m, k -> j.
                 12 ?
                 13 ?   5. [Decrease k.]  Decrease k by one, return to 2.

                 14 ? Assumption: n and A[1:n] are already in memory.
                 15 ? Comment: A is processed in 1:n order using auto-inc feature,
                 16 ? Comment: but k' and j' count upward toward zero from -n.
                 17 ? Comment: A[1:n] is stored in Mp[A+1:A+n].

      10         18 X=10     ? auto-inc register
      7041       19 OPDEF    'NEG'(CMA)=CMA|IAC

                 20          ORG     100
      100        21          BEGIN   init
0100  7200       22 init:    CLA
0101  1331       23          TAD     N
0102  7041       24          NEG
0103  3326       25          DCA     K        ? 1. -n -> k'

                 26          ORG     X
0010  0131       27          DATA    A        ?    X := loc A[0]
                 28          ORG
```

```
Page    2  Algorithm 1.2.1Ø-M  Find the maximum [Knuth68]

Ø1Ø4  141Ø        29           TAD     @%X      ?    A[1] -> AC
Ø1Ø5  5314        3Ø           JMP     change

Ø1Ø6  1327        31 loop:     TAD     M        ? 3. Compare
Ø1Ø7  7Ø41        32           NEG
Ø11Ø  141Ø        33           TAD     @%X
Ø111  751Ø        34           SPA              ?    A[k]-m < Ø
Ø112  5317        35           JMP     test     ?    => go to 5
Ø113  1327        36           TAD     M
Ø114  3327        37 change:   DCA     M        ? 4. A[k] -> m
Ø115  1326        38           TAD     K
Ø116  333Ø        39           DCA     J        ?    k´ -> j´
Ø117  2326        4Ø test:     ISZ     K        ? 5. k´ + 1 -> k´
Ø12Ø  53Ø6        41           JMP     loop     ? 2. k´ ne Ø => go to 3

Ø121  1331        42           TAD     N
Ø122  133Ø        43           TAD     J
Ø123  7ØØ1        44           IAC
Ø124  333Ø        45           DCA     J        ? n + j´ + 1 -> j
Ø125  74Ø2        46           HLT

Ø126  ØØØØ        47 K:        DATA    Ø        ? local
Ø127  ØØØØ        48 M:        DATA    Ø        ? local, output
Ø13Ø  ØØØØ        49 J:        DATA    Ø        ? local, output
Ø131              5Ø N: A:     PAGE             ? global, input
```

```
Page    3  PDP-8 Test Program - exercise the pseudo-ops

                  51 TITLE ´PDP-8 Test Program - exercise the pseudo-ops´

                  52 ? All numbers are octal (for this machine) except for
                  53 ? decimal numbers, which are followed by decimal points.

      47          54          DSK=39.            ? The usual parameter
      4ØØ         55          PAGE2=4ØØ          ? definition facility;
      177         56          MASK = 1~7-1       ? any exp is allowed.

Ø131  Ø137        57          DATA WORD+5        ? Basic data pseudo-op, here 12 bits.
Ø132  Ø132        58 WORD:    DATA .             ? This is the location counter (words).
Ø133  ØØ14        59          DATA ..            ? This is the position counter (bits).

                  6Ø ? This is the variable field data defininition pseudo.
                  61 ? The bit size is given in brackets, followed by any
                  62 ? number of expressions separated by commas. E.g.:

Ø134  43 5Ø 1Ø1   63          FIELD <6> ´C´-4Ø, ´H´-4Ø <7> ´A´, ´R´, ´S´
      Ø136 ØØØ5   64          FIELD <12> ., ..
      Ø4ØØ 124    65          FIELD <12.> PAGE2 <7> ´T´, ´o´, ´E´

                  66 ? The PAD pseudo-op aligns the position counter (..)
                  67 ? to the nearest (greater) multiple of its argument.
                  68 ? The ALIGN works in a similar way on the LC (.).

      12          69          dit=..     ? Where are we?
      4           7Ø          PAD  4
      1Ø          71          dit=..     ? Where are we?
      3           72          PAD  3
      6           73          dit=..     ? Where are we?
      6 5         74          FIELD <3> 6, 5
      144         75          dot=.      ? Where are we?
```

```
Page    4   PDP-8 Test Program - exercise the pseudo-ops

                      76 ? The OPDEF pseudo-op can be used to define some
                      77 ? useful combinations of the micro instructions.

      724Ø            78 OPDEF 'LAO'(.fmt22)=CLA|CMA  ? Load Ac with Ones.
      712Ø            79 OPDEF 'LLO'(CLA)=CLL|CML   ? Load Link with Ones.

                      8Ø ? The following is a nonsense program that illustrates
                      81 ? most of the instruction and data formats, as well as
                      82 ? the rest of the pseudo-ops.

Ø144  1132            83          TAD  %WORD       ? percent means page zero
Ø145  3746            84          DCA  @THERE      ? an off-page reference
Ø146  Ø411            85 THERE:   DATA HERE+1      ? a pointer to page two

                      86          ORG  PAGE2       ? sets the location counter
Ø4ØØ  74Ø5 Ø2ØØ       87          MUY  128.        ? this is decimal, not octal
Ø4Ø2  Ø147            88          AND  %WHERE      ? an off-page reference
      1Ø              89          ALIGN  8.        ? align . to multiple of 8
Ø41Ø  7417 ØØ25       9Ø HERE :   LSR  PAGE2+5&37|2Ø   ? double-word format
Ø412  712Ø            91          LLO              ? see if the OPDEF worked
Ø413  6474            92          IOT  DSK, dit+3~-1   ? input/output format

                      93          ORG              ? this restores last ORG
Ø147  47              94 WHERE:   FIELD <6> DSK
Ø15Ø  73ØØ            95          DATA CLA|CLL     ? another way to get micros
      4ØØ             96          BEGIN PAGE2      ? the last start address wins

                      97 ? There is no "END" statement (eof works fine).
```

```
DUMP OF FILE   TST8.BIN TO  DSK:TST8.DMP     Ø9-23-75  18:37

  ØØØØØØØØØØØ1   ØØØØØØØØØØØ7   ØØØØØØØØØ1ØØ   ØØØØØØØØ72ØØ
  ØØØØØØØØ1331   ØØØØØØØØ7Ø41   ØØØØØØØØ3326   ØØØØØØØØ455Ø

  ØØØØØØØØØØØ1   ØØØØØØØØØØØ4   ØØØØØØØØØØ1Ø   ØØØØØØØØØ131
  ØØØØØØØØ7632
                 ØØØØØØØØØØØ1   ØØØØØØØØØØ43   ØØØØØØØØØ1Ø4
  ØØØØØØØØ141Ø   ØØØØØØØØ5314   ØØØØØØØØ1327   ØØØØØØØØ7Ø41
  ØØØØØØØØ141Ø   ØØØØØØØØ751Ø   ØØØØØØØØ5317   ØØØØØØØØ1327
  ØØØØØØØØ3327   ØØØØØØØØ1326   ØØØØØØØØ333Ø   ØØØØØØØØ2326
  ØØØØØØØØ53Ø6   ØØØØØØØØ1331   ØØØØØØØØ133Ø   ØØØØØØØØ7ØØ1
  ØØØØØØØØ333Ø   ØØØØØØØØ74Ø2   ØØØØØØØØØØØØ   ØØØØØØØØØØØØ
  ØØØØØØØØØØØØ   ØØØØØØØØØ137   ØØØØØØØØØ132   ØØØØØØØØØØ14
  ØØØØØØØØ435Ø   ØØØØØØØØ4Ø64   ØØØØØØØØ523Ø   ØØØØØØØØ57ØØ
  ØØØØØØØØ121Ø   ØØØØØØØØØ124   ØØØØØØØØ6761   ØØØØØØØØ2Ø65
  ØØØØØØØØ4756
                 ØØØØØØØØØØØ1   ØØØØØØØØØØØ6   ØØØØØØØØØ144
  ØØØØØØØØ1132   ØØØØØØØØ3746   ØØØØØØØØØ411   ØØØØØØØØ2114

  ØØØØØØØØØØØ1   ØØØØØØØØØØØ6   ØØØØØØØØØ4ØØ   ØØØØØØØØ74Ø5
  ØØØØØØØØØ2ØØ   ØØØØØØØØØ147   ØØØØØØØØ7415
                                               ØØØØØØØØØØØ1
  ØØØØØØØØØØØ7   ØØØØØØØØØ41Ø   ØØØØØØØØ7417   ØØØØØØØØØØ25
  ØØØØØØØØ712Ø   ØØØØØØØØ6474   ØØØØØØØØ21ØØ
                                               ØØØØØØØØØØØ1
  ØØØØØØØØØØØ5   ØØØØØØØØØ147   ØØØØØØØØ47ØØ   ØØØØØØØØ73ØØ
  ØØØØØØØØ3423
                 ØØØØØØØØØØØ1   ØØØØØØØØØØØ3   ØØØØØØØØØ4ØØ
  ØØØØØØØØ7374
```

APPENDIX E

CASE STUDY: THE IBM-1800

The IBM-1800 [IBM66, IBM68] is the most complex example presented in the thesis. Although it is usually described as a mini-computer, it is a difficult machine from the assembler standpoint (see section 4.3.1 and 5.1.2.2). This is because of the relatively large number of special cases associated with most of the instructions, and the fact that instruction formats are in general not uniquely determined by op-code values. The "modify index" instruction, for example, has been described using four distinct mnemonics: BR (branch, or modify PC), MDM (modify memory), MDX (modify index short), and MDXF (modify index long). Those familiar with IBM's version of the assembler for this machine will find that several "new" instructions have been created in this way. Also, since instruction forms have not been implemented, this version of the assembler makes no distinction between the absolute (disp) and relative (rel) addressing schemes used in most short-format instructions.

As before, the SYNTAX, CODE, and BOOT files follow the ISP description below. The sample program in appendix A has again been re-coded, and the listing and octal dump appear at the end of the appendix.

```
                        {IBM1800 LISTING}

{IBM1800.LST[22,54]=IBM1800.ISP[22,54]    16-Sep-75   08:59}

Mp_State

   M[0:0FFFFb16]<0:15>


Pc_State

   PC<0:15>
   X[1:3]<0:15>
   A<0:15>; Q<0:15>
   C; Ov
   Run


External_Pc_State {IO}

   IOCC[0:1]<0:15>
      IO_address<0:15> := IOCC[0]
      IO_area<0:4>     := IOCC[1]<0:4>
      IO_function<0:2> := IOCC[1]<5:7>
      IO_modifier<0:7> := IOCC[1]<8:15>


Instruction_Format

   I/Instruction[0:1]<0:15>
      op<0:4>    := I[0]<0:4>
      f          := I[0]<5>
      t<0:1>     := I[0]<6:7>
      shop<0:1>  := I[0]<8:9>
      sc<0:5>    := I[0]<10:15>
      d<0:7>     := I[0]<8:15>
      ia         := I[0]<8>
      bo         := I[0]<9>
      cond<0:5>  := I[0]<10:15>
      a<0:15>    := I[1]<0:15>


Instruction_Interpretation_Process

  'Interpret' :=
      Run =>
        (M[PC] -> I[0]; 1+PC -> PC next
         f => (M[PC] -> I[1]; 1+PC -> PC)
         next 'Execute' next 'Interpret')
```

```
Effective_Address_Calculation

   disp<Ø:15> := sign_extend(d)

   rel<Ø:15> := disp+PC

   x<Ø:15> := (~ia => a; ia => M[a])

   y<Ø:15> :=
      (t eq Ø => rel;
       t ne Ø => disp+X[t])

   yd<Ø:15> := (y<15> => y; ~y<15> => y+1)

   z<Ø:15> :=
      (t eq Ø => x;
       t ne Ø => x+X[t])

   zd<Ø:15> := (z<15> => z; ~z<15> => z+1)

   Msum<Ø:15> := disp+M[a]

   Ysum<Ø:15> := disp+X[t]

   Xsum<Ø:15> := x+X[t]

   s<Ø:5> :=
      (t eq Ø => sc;
       t ne Ø => X[t]<1Ø:15>)

   skip :=
      (cond<5> => Ø -> Ov;
       (~Ov & cond<5>) | (~C & cond<4>) |
       (A<15> & cond<3>) | (A gt Ø & cond<2>) |
       (A lt Ø & cond<1>) | (A eq Ø & cond<Ø>))


Instruction_Execution_Process

   ´Execute´ :=
     (´XIO´(:= op eq ØØØØ1b & ~f) =>
         (M[y].M[yd] -> IOCC[Ø:1] next ´Execute_IO´);
      ´XIOF´(:= op eq ØØØØ1b & f) =>
         (M[z].M[zd] -> IOCC[Ø:1] next ´Execute_IO´);

      op eq ØØØ1Øb & ~f =>
       (´SLA´(:= shop eq ØØb) =>
           C.A*(2^s) -> C.A{logical};
        ´SLAC´(:= shop eq Ø1b) =>
          (t eq Ø => C.A*(2^s) -> C.A{logical};
           t ne Ø =>
              (norm_count(A) ge s =>
                 (A*(2^s) -> A{logical};
                  Ø -> X[t]<8:15>; Ø -> C);
```

```
              norm_count(A) lt s =>
                 (normalize(A) -> A{logical}; 1 -> C;
                  s-norm_count(A) -> X[t]<8:15>))));
        ´SLT´(:= shop eq 1Øb) =>
           C.A.Q*(2^s) -> C.A.Q{logical};
        ´SLC´(:= shop eq 11b) =>
          (t eq Ø => C.A.Q*(2^s) -> C.A.Q{logical};
           t ne Ø =>
              (norm_count(A.Q) ge s =>
                 (A.Q*(2^s) -> A.Q{logical};
                  Ø -> X[t]<8:15>; Ø -> C);
               norm_count(A.Q) lt s =>
                 (normalize(A.Q) -> A.Q{logical}; 1 -> C;
                  s-norm_count(A.Q) -> X[t]<8:15>)))));
      op eq ØØØ11b & ~f =>
       (´SRA´(:= shop eq ØØb) => A/(2^s) -> A{logical};
        ´SRT´(:= shop eq 1Øb) => A.Q/(2^s) -> A.Q{arith};
        ´RTE´(:= shop eq 11b) => A.Q/(2^s) -> A.Q{rotate});

      ´LDS´(:= op eq ØØ1ØØb & ~f) =>
         (cond<4> -> C; cond<5> -> Ov);
      ´STS´(:= op eq ØØ1Ø1b & ~f) =>
         (Ø -> M[y]<8:13>; Ø -> C; Ø -> Ov;
          C -> M[y]<14>; Ov -> M[y]<15>);
      ´STSF´(:= op eq ØØ1Ø1b & f & ~bo) =>
         (Ø -> M[z]<8:13>; Ø -> C; Ø -> Ov;
          C -> M[z]<14>; Ov -> M[z]<15>);
      ´STR´(:= op eq ØØ1Ø1b & f & bo & ~cond<5>) =>
          undefined(M[z]); {enable read access}
      ´STW´(:= op eq ØØ1Ø1b & f & bo & cond<5>) =>
          undefined(M[z]); {enable write access}
      ´WAIT´(:= op eq ØØ11Øb & ~f) => undefined;

      ´SKI´(:= op eq Ø1ØØØb & ~f) =>
          (y+1 -> PC; PC -> M[y]);
      ´BRI´(:= op eq Ø1ØØØb & f) =>
          ~skip => (z+1 -> PC; PC -> M[z]);
      ´SKC´(:= op eq Ø1ØØ1b & ~f) =>
          skip => 1+PC -> PC;
      ´BRC´(:= op eq Ø1ØØ1b & f & ~bo) =>
          ~skip => z -> PC;
      ´BOC´(:= op eq Ø1ØØ1b & f & bo) =>
         (~skip => z -> PC; undefined(bo));

      ´JMP´(:= op eq Ø11ØØb & ~f & t eq Ø) => disp -> PC;
      ´JMPF´(:= op eq Ø11ØØb & f & t eq Ø) => x -> PC;
      ´LDX´(:= op eq Ø11ØØb & ~f & t ne Ø) => disp -> X[t];
      ´LDXF´(:= op eq Ø11ØØb & f & t ne Ø) => x -> X[t];
      ´STX´(:= op eq Ø11Ø1b & ~f) =>
         (t eq Ø => PC -> M[rel];
          t ne Ø => X[t] -> M[rel]);
      ´STXF´(:= op eq Ø11Ø1b & f) =>
         (t eq Ø => PC -> M[x];
          t ne Ø => X[t] -> M[x]);
```

```
'BR'(:= op eq 01110b & ~f & t eq 0) => rel -> PC;
'MDM'(:= op eq 01110b & f & t eq 0) =>
   (Msum -> M[a];
    Msum eq 0 | (M[a]<0>#Msum<0>) => 1+PC -> PC);
'MDX'(:= op eq 01110b & ~f & t ne 0) =>
   (Ysum -> X[t];
    Ysum eq 0 | (X[t]<0>#Ysum<0>) => 1+PC -> PC);
'MDXF'(:= op eq 01110b & f & t ne 0) =>
   (Xsum -> X[t];
    Xsum eq 0 | (X[t]<0>#Xsum<0>) => 1+PC -> PC);
'AD'(:= op eq 10000b & ~f) =>
   (A+M[y] -> C.A;
    overflow(A+M[y]) -> Ov);
'ADF'(:= op eq 10000b & f) =>
   (A+M[z] -> C.A;
    overflow(A+M[z]) -> Ov);
'ADD'(:= op eq 10001b & ~f) =>
   (A Q+M[y].M[yd] -> C.A.Q;
    overflow(A.Q+M[y].M[yd]) -> Ov);
'ADDF'(:= op eq 10001b & f) =>
   (A Q+M[z].M[zd] -> C.A.Q;
    overflow(A.Q+M[z].M[zd]) -> Ov);

'SB'(:= op eq 10010b & ~f) =>
   (A-M[y] -> C.A;
    overflow(A-M[y]) -> Ov);
'SBF'(:= op eq 10010b & f) =>
   (A-M[z] -> C.A;
    overflow(A-M[z]) -> Ov);
'SBD'(:= op eq 10011b & ~f) =>
   (A.Q-M[y].M[yd] -> C.A.Q;
    overflow(A.Q-M[y].M[yd]) -> Ov);
'SBDF'(:= op eq 10011b & f) =>
   (A.Q-M[z].M[zd] -> C.A.Q;
    overflow(A.Q-M[z].M[zd]) -> Ov);

'ML'(:= op eq 10100b & ~f) => A*M[y] -> A.Q;
'MLF'(:= op eq 10100b & f) => A*M[z] -> A.Q;
'DV'(:= op eq 10101b & ~f) =>
   (M[y] eq 0 => 1 -> Ov;
    M[y] ne 0 =>
       (A.Q/M[y] -> A; A.Q\M[y] -> Q;
        overflow(A.Q/M[y]) -> Ov));
'DVF'(:= op eq 10101b & f) =>
   (M[z] eq 0 => 1 -> Ov;
    M[z] ne 0 =>
       (A.Q/M[z] -> A; A.Q\M[z] -> Q;
        overflow(A.Q/M[z]) -> Ov));

'CMP'(:= op eq 10110b & ~f) =>
   (A lt M[y] => 1+PC -> PC;
    A eq M[y] => 2+PC -> PC);
```

```
'CMPF'(:= op eq 10110b & f) =>
   (A lt M[z] => 1+PC -> PC;
    A eq M[z] => 2+PC -> PC);
'DCM'(:= op eq 10111b & ~f) =>
   (A.Q lt M[y].M[yd] => 1+PC -> PC;
    A.Q eq M[y].M[yd] => 2+PC -> PC);
'DCMF'(:= op eq 10111b & f) =>
   (A.Q lt M[z].M[zd] => 1+PC -> PC;
    A.Q eq M[z].M[zd] => 2+PC -> PC);

'LD'(:= op eq 11000b & ~f) => M[y] -> A;
'LDF'(:= op eq 11000b & f) => M[z] -> A;
'LDD'(:= op eq 11001b & ~f) => M[y].M[yd] -> A.Q;
'LDDF'(:= op eq 11001b & f) => M[z].M[zd] -> A.Q;
'STO'(:= op eq 11010b & ~f) => A -> M[y];
'STOF'(:= op eq 11010b & f) => A -> M[z];
'STD'(:= op eq 11011b & ~f) =>
   (A -> M[y]; ~y<15> => Q -> M[y+1]);
'STDF'(:= op eq 11011b & f) =>
   (A -> M[z]; ~z<15> => Q -> M[z+1]);

'AND'(:= op eq 11100b & ~f) => A&M[y] -> A;
'ANDF'(:= op eq 11100b & f) => A&M[z] -> A;
'OR'(:= op eq 11101b & ~f) => A|M[y] -> A;
'ORF'(:= op eq 11101b & f) => A|M[z] -> A;
'XOR'(:= op eq 11110b & ~f) => A#M[y] -> A;
'XORF'(:= op eq 11110b & f) => A#M[z] -> A)


{IO_Instruction_Set}

  'Execute_IO' :=
     (IO_function eq 000b => undefined{off_line};
      IO_function eq 001b => undefined{write};
      IO_function eq 010b => undefined{read};
      IO_function eq 011b => undefined{sense_level};
      IO_function eq 100b => undefined{control};
      IO_function eq 101b => undefined{init_write};
      IO_function eq 110b => undefined{init_read};
      IO_function eq 111b => undefined{sense})
```

```
                       {IBM18ØØ SYNTAX}

{SYNTAX.SYN[22,54]=IBM18ØØ.ISP[22,54]      16-Sep-75   Ø7:11}

.opr=1;         .nam=2;         .num=3;         .stg=4;
.pag=5;         .til=6;         .beg=7;         .aln=8;
.org=9;         .def=1Ø;        .vfd=11;        .pad=12;
.fup=13;        .dat=14;        .fmt2Ø=2Ø;      .fmt21=21;
.fmt22=22;      .fmt23=23;      .fmt24=24;      .fmt25=25;
.fmt26=26;      .fmt27=27;      .fmt28=28;

      opr=.opr;       num=.opr;       stg=.opr;

<prg> ::= <stl> 177b [END];

<stl> ::= 12b             [LINE] |
          <stm> 12b       [LINE] |
          <stl> 12b       [LINE] |
          <stl> <stm> 12b [LINE] ;

<stm> ::=
   [HASH] .nam [MNEMON] ´:´ [LABEL] |
   [HASH] .nam [MNEMON] ´:´ [LABEL] <stm> |
   [HASH] .nam [MNEMON] ´=´ <exp> [EQUATE];

<vfd> ::= ´<´ <exp> ´>´ [SIZE] <exp> [FIELD] |
    <vfd> ´<´ <exp> ´>´ [SIZE] <exp> [FIELD] |
    <vfd> ´,´ <exp> [FIELD];

<exp> ::= <atm> |
   <exp> ´+´ <atm> [ADD] | <exp> ´-´ <atm> [SUB] |
   <exp> ´*´ <atm> [MUL] | <exp> ´/´ <atm> [DIV] |
   <exp> ´\´ <atm> [REM] | <exp> ´&´ <atm> [AND] |
   <exp> ´|´ <atm> [IOR] | <exp> ´=´ <atm> [EQV] |
   <exp> ´#´ <atm> [XOR] | <exp> ´~´ <atm> [SHF] ;

<atm> ::=
   .nam [NAME] | .num [NUMBER] | .stg [STRING] |
   ´~´ <atm> [NOT] | ´|´ <atm> [ABV] |
   ´-´ <atm> [NEG] | ´+´ <atm> |
   ´(´ <exp> ´)´ ;

<stm> ::= [HASH] .fmt2Ø [PROLOG]
          <exp> [SETt] ,
          <exp> [SETd] [EPILOG];

<stm> ::= [HASH] .fmt21 [PROLOG]
          <exp> [SETt] ,
          <exp> [SETcond] ,
          <exp> [SETa] [EPILOG];
<stm> ::= [HASH] .fmt21 [PROLOG]
```

```
          <exp> [SETt] ,
          ´@´ [SETlia] <exp> [SETcond] ,
          <exp> [SETa] [EPILOG];

<stm> ::= [HASH] .fmt22 [PROLOG]
          <exp> [SETt] ,
          <exp> [SETsc] [EPILOG];

<stm> ::= [HASH] .fmt23 [PROLOG]
          <exp> [SETt] ,
          <exp> [SETa] [EPILOG];
<stm> ::= [HASH] .fmt23 [PROLOG]
          <exp> [SETt] ,
          ´@´ [SETlia] <exp> [SETa] [EPILOG];

<stm> ::= [HASH] .fmt24 [PROLOG]
          <exp> [SETd] ,
          <exp> [SETa] [EPILOG];

<stm> ::= [HASH] .fmt25 [PROLOG]
          <exp> [SETcond] [EPILOG];

<stm> ::= [HASH] .fmt26 [PROLOG]
          <exp> [SETd] [EPILOG];

<stm> ::= [HASH] .fmt27 [PROLOG] [EPILOG];

<stm> ::= [HASH] .fmt28 [PROLOG]
          <exp> [SETa] [EPILOG];
<stm> ::= [HASH] .fmt28 [PROLOG]
          ´@´ [SETlia] <exp> [SETa] [EPILOG];

<stm> ::=
   [HASH] .org ´´      [ORIGIN]  |
   [HASH] .org <exp>  [ORIGIN]  |
   [HASH] .dat <exp>  [DATA]    |
   [HASH] .vfd <vfd>  |
   [HASH] .aln <exp>  [ALIGN]   |
   [HASH] .pad <exp>  [PAD]     |
   [HASH] .pag        [PAGE]    |
   [HASH] .til .stg   [TITLE]   |
   [HASH] .beg <exp>  [BEGIN]   |
   [HASH] .def .stg   [MNEMON]
      ´(´.nam [FORMAT]´)´ ´=´ <exp> [OPDEF] |
   .stg [MNEMON] <exp> <exp> <exp> <exp> [BOOT];
```

```
# ISP 3.6                                           16-Sep-75 #

                        # IBM18ØØ #
                        CALL ME CODE;


# CODE.I1Ø[22,54]=IBM18ØØ.ISP[22,54]     16-Sep-75   Ø7:11 #

# IMP: absolute value function #
 <ATOM> ::= ABS <ATOM,A>
        ::= DEWOP(214B,AREG1(1,15B),A);

# LEX: lexeme format # LET
 LTYP=TOK,       # lexical type #
 LADR=TOK[1],    # symbol table pointer #
 LLEN=TOK[2],    # character length #
 LVAL=TOK[3];    # value (char-array) #
 TOK IS COMMON; # owned by LEX #

# INTRPT: directory modes #
 LET UDF=Ø, ABL=1, REL=2, EXT=3;

# SYMBOL: directory access functions #
 <VBL> ::= <VBL,A>.TYP ::= "[1][A]"; # type   #
 <VBL> ::= <VBL,A>.FMT ::= "[2][A]"; # format #
 <VBL> ::= <VBL,A>.MOD ::= "[3][A]"; # mode   #
 <VBL> ::= <VBL,A>.VAL ::= "[4][A]"; # value  #
 <VBL> ::= <VBL,A>.LEN ::= "[5][A]"; # length #
 <VBL> ::= <VBL,A>.SYM ::= "[6][A]"; # symbol #

# SEM: instruction field sizes # LET
 NOP=ØB, BYTES=1, BYTESIZE=16, INSTSIZE=16, WORDSIZE=16;

PASS,DOT,DIT ARE COMMON;

SUBR PROLOG() IS
  (DEL_LOP[LTYP-2Ø];
   BLINE(BYTESIZE);
   PASS =>
     (PUTADR(DOT.VAL);
      I[Ø]_LADR.VAL<16,Ø>);
   Ø);

DEL:DATA(Ø); I:DATA(Ø,Ø);
LOP:DATA(1,2,1,2,2,1,1,1,2);

SUBR SETsc() IS
  (POP(M V); PASS =>
     (M=UDF => PUTERR(R'U');
      T_V ARS 6 => NOT T => PUTERR(R'T');
      I[Ø]<6,Ø>_V);
   Ø);

SUBR SETa() IS
```

```
  (POP(M,V); PASS =>
     (M=UDF => PUTERR(R'U');
      T_V ARS 16 => NOT T => PUTERR(R'T');
      I[1]<16,Ø>_V);
   Ø);

SUBR SETcond() IS
  (POP(M,V); PASS =>
     (M=UDF => PUTERR(R'U');
      T_V ARS 6 => NOT T => PUTERR(R'T');
      I[Ø]<6,Ø>_V);
   Ø);

SUBR SETlia() IS
  (PASS => (I[Ø]<1,7>_1); Ø);

SUBR SETd() IS
  (POP(M,V); PASS =>
     (M=UDF => PUTERR(R'U');
      T_V ARS 8 => NOT T => PUTERR(R'T');
      I[Ø]<8,Ø>_V);
   Ø);

SUBR SETt() IS
  (POP(M,V); PASS =>
     (M=UDF => PUTERR(R'U');
      T_V ARS 2 => NOT T => PUTERR(R'T');
      I[Ø]<2,8>_V);
   Ø);

SUBR EPILOG() IS
  (  (PASS =>
        (PUTDAT(INSTSIZE,I[J]);
         BYTLOD(DOT.VAL,I[J]);
         I[J]_Ø);
      DOT.VAL_DOT.VAL+1)
   FOR J TO DEL-1; DEL_Ø);

SUBR DATA() IS
  (BLINE(BYTESIZE);
   POP(M V); PASS =>
     (PUTADR(DOT.VAL);
      M=UDF => PUTERR(R'U');
      T_V ARS WORDSIZE =>
         NOT T => PUTERR(R'T');
      V_V AND (1 LS WORDSIZE)-1;
      PUTDAT(WORDSIZE,V);
      BYTLOD(DOT.VAL,V));
   DOT.VAL_DOT.VAL+1; Ø);

SUBR ALINE(N) IS
  (FIL_DOT.VAL//N =>
      DOT.VAL_DOT.VAL+FIL_N-FIL;
   FIL);
```

```
SUBR BLINE(N) IS
  (FIL_DIT.VAL//N => FILTER(FIL,Ø); FIL);

SUBR FILTER(S,V) IS
  (POS_DIT.VAL; SIZ_S; VAL_V;
   WHILE SIZ>Ø DO
      (BTS_SIZ>POS => BTS_POS;
       PASS =>
         (MSK_(1 LS SIZ)-1; VAL_VAL AND MSK;
          II_II OR VAL LS POS-SIZ);
       POS_POS-BTS=Ø =>
         (PASS => (BYTLOD(DOT.VAL,II); II_Ø);
          DOT.VAL_DOT.VAL+1; POS_BYTESIZE);
       SIZ_SIZ-BTS);
   DIT.VAL_POS; Ø);

II:DATA(Ø) %%
```

```
? Bootstrap IBM18ØØ.ISP   16-Sep-75   Ø8:59
? This machine uses base 16. for constants.

'..'      4.  Ø. 1. 16.
'.'       3.  Ø. 1.  Ø.
'PAGE'    2.  5. Ø.  Ø.
'TITLE'   2.  6. Ø.  Ø.
'BEGIN'   2.  7. Ø.  Ø.
'ALIGN'   2.  8. Ø.  Ø.
'ORG'     2.  9. Ø.  Ø.
'OPDEF'   2. 1Ø. Ø.  Ø.
'FIELD'   2. 11. Ø.  Ø.
'PAD'     2. 12. Ø.  Ø.
'DATA'    2. 14. Ø.  Ø.
'.fmt2Ø'  1. 2Ø. 1.  Ø.  ? .fmt2Ø: t d
'.fmt21'  1. 21. 1.  Ø.  ? .fmt21: t ia cond a
'.fmt22'  1. 22. 1.  Ø.  ? .fmt22: t sc
'.fmt23'  1. 23. 1.  Ø.  ? .fmt23: t ia a
'.fmt24'  1. 24. 1.  Ø.  ? .fmt24: d a
'.fmt25'  1. 25. 1.  Ø.  ? .fmt25: cond
'.fmt26'  1. 26. 1.  Ø.  ? .fmt26: d
'.fmt27'  1. 27. 1.  Ø.  ? .fmt27: (null)
'.fmt28'  1. 28. 1.  Ø.  ? .fmt28: ia a
PAGE
TITLE    'IBM18ØØ Op-code Definitions'

OPDEF    'DCM'(.fmt2Ø)=471Ø4.       ? .fmt2Ø: t d
OPDEF    'BRI'(.fmt21)=174Ø8.       ? .fmt21: t ia cond a
OPDEF    'SLA'(.fmt22)=4Ø96.        ? .fmt22: t sc
OPDEF    'STD'(.fmt2Ø)=55296.       ? .fmt2Ø: t d
OPDEF    'ADD'(.fmt2Ø)=34816.       ? .fmt2Ø: t d
OPDEF    'STXF'(.fmt23)=27648.      ? .fmt23: t ia a
OPDEF    'STSF'(.fmt23)=11264.      ? .fmt23: t ia a
OPDEF    'OR'(.fmt2Ø)=59392.        ? .fmt2Ø: t d
OPDEF    'LDF'(.fmt23)=50176.       ? .fmt23: t ia a
OPDEF    'DCMF'(.fmt23)=48128.      ? .fmt23: t ia a
OPDEF    'RTE'(.fmt22)=6336.        ? .fmt22: t sc
OPDEF    'MDM'(.fmt24)=29696.       ? .fmt24: d a
OPDEF    'STDF'(.fmt23)=5632Ø.      ? .fmt23: t ia a
OPDEF    'ADDF'(.fmt23)=3584Ø.      ? .fmt23: t ia a
OPDEF    'LDX'(.fmt2Ø)=25344.       ? .fmt2Ø: t d
OPDEF    'SBF'(.fmt23)=37888.       ? .fmt23: t ia a
OPDEF    'MLF'(.fmt23)=41984.       ? .fmt23: t ia a
OPDEF    'BOC'(.fmt21)=1952Ø.       ? .fmt21: t ia cond a
OPDEF    'LDS'(.fmt25)=8192.        ? .fmt25: cond
OPDEF    'SLAC'(.fmt22)=416Ø.       ? .fmt22: t sc
OPDEF    'AD'(.fmt2Ø)=32768.        ? .fmt2Ø: t d
OPDEF    'BR'(.fmt26)=28672.        ? .fmt26: d
OPDEF    'LDD'(.fmt2Ø)=512ØØ.       ? .fmt2Ø: t d
OPDEF    'LDXF'(.fmt23)=26368.      ? .fmt23: t ia a
OPDEF    'SKC'(.fmt25)=18432.       ? .fmt25: cond
OPDEF    'DVF'(.fmt23)=44Ø32.       ? .fmt23: t ia a
OPDEF    'STO'(.fmt2Ø)=53248.       ? .fmt2Ø: t d
OPDEF    'SRT'(.fmt22)=6272.        ? .fmt22: t sc
OPDEF   ´SBD´(.fmt2Ø)=38912.       ? .fmt2Ø: t d
OPDEF   ´LDDF´(.fmt23)=52224.      ? .fmt23: t ia a
OPDEF   ´MDX´(.fmt2Ø)=2944Ø.       ? .fmt2Ø: t d
OPDEF   ´STOF´(.fmt23)=54272.      ? .fmt23: t ia a
OPDEF   ´LD´(.fmt2Ø)=49152.        ? .fmt2Ø: t d
OPDEF   ´SLT´(.fmt22)=4224.        ? .fmt22: t sc
OPDEF   ´STW´(.fmt23)=11329.       ? .fmt23: t ia a
OPDEF   ´SBDF´(.fmt23)=39936.      ? .fmt23: t ia a
OPDEF   ´STR´(.fmt23)=11328.       ? .fmt23: t ia a
OPDEF   ´XIO´(.fmt2Ø)=2Ø48.        ? .fmt2Ø: t d
OPDEF   ´SB´(.fmt2Ø)=36864.        ? .fmt2Ø: t d
OPDEF   ´MDXF´(.fmt23)=3Ø464.      ? .fmt23: t ia a
OPDEF   ´ML´(.fmt2Ø)=4Ø96Ø.        ? .fmt2Ø: t d
OPDEF   ´XOR´(.fmt2Ø)=6144Ø.       ? .fmt2Ø: t d
OPDEF   ´BRC´(.fmt21)=19456.       ? .fmt21: t ia cond a
OPDEF   ´ORF´(.fmt23)=6Ø416.       ? .fmt23: t ia a
OPDEF   ´CMP´(.fmt2Ø)=45Ø56.       ? .fmt2Ø: t d
OPDEF   ´SKI´(.fmt2Ø)=16384.       ? .fmt2Ø: t d
OPDEF   ´WAIT´(.fmt27)=12288.      ? .fmt27: (null)
OPDEF   ´XIOF´(.fmt23)=3Ø72.       ? .fmt23: t ia a
OPDEF   ´DV´(.fmt2Ø)=43ØØ8.        ? .fmt2Ø: t d
OPDEF   ´JMP´(.fmt26)=24576.       ? .fmt26: d
OPDEF   ´XORF´(.fmt23)=62464.      ? .fmt23: t ia a
OPDEF   ´CMPF´(.fmt23)=46Ø8Ø.      ? .fmt23: t ia a
OPDEF   ´SLC´(.fmt22)=4288.        ? .fmt22: t sc
OPDEF   ´AND´(.fmt2Ø)=57344.       ? .fmt2Ø: t d
OPDEF   ´ADF´(.fmt23)=33792.       ? .fmt23: t ia a
OPDEF   ´JMPF´(.fmt28)=256ØØ.      ? .fmt28: ia a
OPDEF   ´SRA´(.fmt22)=6144.        ? .fmt22: t sc
OPDEF   ´STX´(.fmt2Ø)=26624.       ? .fmt2Ø: t d
OPDEF   ´STS´(.fmt2Ø)=1Ø24Ø.       ? .fmt2Ø: t d
OPDEF   ´ANDF´(.fmt23)=58368.      ? .fmt23: t ia a
```

```
Page    1  Algorithm 1.2.1Ø-M  Find the maximum [Knuth68]

                    1 TITLE ´Algorithm 1.2.1Ø-M  Find the maximum [Knuth68]´

                    2 ? Given n elements A[1], A[2], ... ,A[n], find m and j
                    3 ? such that m = A[j] = max Ø<k<n+1 A[k], and for which
                    4 ? j is as large as possible.

                    5 ?   1. [Initialize.]  Assign n -> j, n-1 -> k, A[n] -> m.
                    6 ?
                    7 ?   2. [All tested?]  If k=Ø, the algorithm terminates.
                    8 ?
                    9 ?   3. [Compare.]  Go to 5 unless A[k] > m.
                   1Ø ?
                   11 ?   4. [Change m.]  Assign k -> j, A[k] -> m.
                   12 ?
                   13 ?   5. [Decrease k.]  Decrease k by one, return to 2.
                   14 ? Assumption: n and A[1:n] are already in memory.
                   15 ? Comment: m := AC and k := X[1] {index register}.
                   16 ? Comment: A[1:n] is stored in Mp[A+1:A+n].

      1            17 K=1      ? k is index register X[1]

                   18          ORG  128.
      8Ø           19          BEGIN here
ØØ8Ø  658Ø ØØ93    2Ø here:    LDXF  K,@N    ? 1. n -> k
ØØ82  6Ø88         21          JMP   load
ØØ83  B5ØØ ØØ93    22 loop:    CMPF  K,A     ? 3. Compare
ØØ85  6Ø8C         23          JMP   test    ?    m > A[k]
ØØ86  483F         24          SKC   -1      ?    m < A[k]
ØØ87  6Ø8C         25          JMP   test    ?    m = A[k]
ØØ88  6DØØ ØØ91    26 load:    STXF  K,J     ? 4. k -> j
ØØ8A  C5ØØ ØØ93    27          LDF   K,A     ?    A[k] -> m
ØØ8C  71FF         28 test:    MDX   K,-1    ? 5. k-1 -> k
ØØ8D  6Ø83         29          JMP   loop    ? 2. k ne Ø => go to 3
```

```
Page    2  Algorithm 1.2.10-M  Find the maximum [Knuth68]

008E  D400 0092    30         STOF  0,M
0090  3000         31         WAIT

0091  0000         32 J:      DATA    0    ? local, output
0092  0000         33 M:      DATA    0    ? local, output
0093               34 N: A:                ? global, input
```

```
DUMP OF FILE TST180.BIN TO DSK:TST180.DMP    09-16-75  09:05

000000000001   000000000026   000000000200   000000062600
000000000223   000000060210   000000132400   000000000223
000000060214   000000044077   000000060214   000000066400
000000000221   000000142400   000000000223   000000070777
000000060203   000000152000   000000000222   000000030000
000000000000   000000000000   000000067654

000000000003   000000000200   000000177574   000000000001
```

REFERENCES

[Altman74] Altman, V. E. A language implementation system. Project MAC TR-126, Massachusetts Institute of Technology, 1974.

[Barbacci72] Barbacci, M. R., Bell, C. G., and Newell, A. ISP: a language to describe instruction sets and other register transfer systems. Proc. IEEE COMPCON (September 1972), 219-22.

[Barbacci73] Barbacci, M. R. A comparison of register transfer languages for describing computers and digital systems. Dept. of Computer Science, Carnegie-Mellon University, 1973.

[Barbacci74] Barbacci, M. R. and Siewiorek, D. P. Some aspects of the symbolic manipulation of computer descriptions. Dept. of Computer Science, Carnegie-Mellon University, 1974.

[Barbacci75] Barbacci, M. R. A user's guide to the ISPL compiler. Dept. of Computer Science, Carnegie-Mellon University, 1975.

[Barron69] Barron, D. W. Assemblers and Loaders. American Elsevier, New York, 1969.

[Bell70] Bell, C. G. and Newell, A. The PMS and ISP descriptive systems for computer structures. Proc. AFIPS SJCC 36 (May 1970), 351-74.

[Bell71] Bell, C. G. and Newell, A. Computer Structures: Readings and Examples. McGraw-Hill, New York, 1971.

[Bell72] Bell, C. G., Grason, J., and Newell, A. Designing Computers and Digital Systems. Digital Press, Maynard, Mass., 1972.

[Bilofsky73] Bilofsky, W. PDP-10 IMP72 reference manual. Res. Rept., Dept. of Computer Science, Yale University, 1973.

[Brown74] Brown, P. J. Macro Processors and Techniques for Portable Software. Wiley, New York, 1974.

[CDC67a] Control Data 6400/6500/6600 Computer Systems: Reference Manual. Pub. 60100000, Control Data Corp., Minneapolis, 1967.

[CDC67b] Control Data 6400/6500/6600 Computer Systems: COMPASS Reference Manual. Pub. 60190900, Control Data Corp., Minneapolis, 1967.

[Chu72] Chu, Y. Computer Organization and Microprogramming. Prentice-Hall, New York, 1972.

[Chu74] Chu, Y. Why do we need computer hardware description languages? IEEE Computer 7,12 (December 1974), 18-22.

[Cohen70] Cohen, D. J. and Gotlieb, C. C. A list structure form of grammars for syntactic analysis. Computer Surveys 2,1 (March 1970), 65-82.

[Conway63] Conway, M. E. Design of a separable transition diagram compiler. Comm. ACM 6,7 (July 1963), 396-408.

[Darringer69] Darringer, J. A. The description, simulation, and automatic implementation of digital computer processors. Ph.D. thesis, Dept. of Elec. Engr., Carnegie-Mellon University, 1969.

[DEC67] Digital Small Computer Handbook. Digital Equipment Corp., Maynard, Mass., 1967.

[DEC70] PDP-11 absolute binary loader -- V006A. Digital Equipment Corp., Maynard, Mass., 1970.

[DEC71] PDP-11/45 Processor Handbook. Digital Equipment Corp., Maynard, Mass., 1971.

[DEC73] decsystem10 Assembly Language Handbook. 3rd ed. Digital Equipment Corp., Maynard, Mass., 1973.

[DGC74] Programmer's Reference Manual: ECLIPSE Computer. Pub. 015-000024-00.1, Data General Corp., Southboro, Mass., 1974.

[Donegan73] Donegan, M. K. An approach to the automatic generation of code generators. Ph.D. thesis, Rice University, 1973.

[Donovan72] Donovan, J. J. Systems Programming. McGraw-Hill, New York, 1972.

[Duley68] Duley, J. R. and Dietmeyer, D. L. A digital system design language (DDL). IEEE Transactions on Computers, C-17,9 (September 1968), 850-61.

[Feldman67] Feldman, J. A. and Gries, D. Translator writing systems. Tech. Rept. CS69, Computer Science Dept., Stanford University, 1967.

[Ferguson66] Ferguson, D. E. Evolution of the meta-assembly program. Comm. ACM 9,3 (March 1966), 19Ø-96.

[Flon74] Flon, L. A survey of some issues concerning abstract data types. Dept. of Computer Science, Carnegie-Mellon University, 1974.

[GE69] GE-625/635 Programming Reference Manual. Pub. CPB-1ØØ4F, General Electric Corp., 1969.

[Goldman74] Goldman, S. ISP compiler: project report and user manual. Dept. of Computer Science, Carnegie-Mellon University, 1974.

[Gries71] Gries, D. Compiler Construction for Digital Computers. Wiley, New York, 1971.

[Honeywell62] Honeywell 8ØØ Programmer's Reference Manual. Pub. DSI-31A, Minneapolis-Honeywell Regulator Co., Wellesley Hills, Mass., 1962.

[IBM57] SOAP II Programmer's Reference Manual. International Business Machines Corp., White Plains, N.Y., 1957.

[IBM6Ø] IBM-14Ø1 Data Processing System: Reference Manual. Pub. A24-14Ø3-1, International Business Machines Corp., White Plains, N.Y., 196Ø.

[IBM62] IBM 7Ø94 Principles of Operation. Pub. A22-67Ø3-4, International Business Machines Corp., White Plains, N.Y., 1962.

[IBM66] IBM 18ØØ Functional Characteristics. Pub. A26-5918-5, International Business Machines Corp., White Plains, N.Y., 1966.

[IBM67] A Programmer's Introduction to the IBM System/36Ø Architecture, Instructions, and Assembly Language. Pub. C2Ø-1646-4, International Business Machines Corp., White Plains, N.Y., 1967.

[IBM68] IBM 18ØØ Assembler Language. Pub. C26-5882-4, International Business Machines Corp., White Plains, N.Y., 1968.

[IBM7Ø] IBM System/36Ø Principles of Operation. Pub. GA22-6821-8, International Business Machines Corp., White Plains, N.Y., 197Ø.

[Irons7Ø] Irons, E. T. Experience with an extensible language. Comm. ACM 13,1 (January 197Ø), 31-4Ø.

[Irons71] Irons, E. T. Syntax graphs and fast context-free parsing. Res. Rept. 71-1, Dept. of Computer Science, Yale University, 1971.

[Iverson62] Iverson, K. E. A Programming Language. Wiley, New York, 1962.

[Johnson68] Johnson, W. L., Porter, J. H., Ackley, S. I., and Ross, D. T. Automatic generation of efficient lexical processors using finite state techniques. Comm. ACM 11,12 (December 1968), 8Ø5-13.

[King75] King, J. C. A new approach to program testing. ACM SIGPLAN Notices 1Ø,6 (June 1975), 228-33.

[Knuth68] Knuth, D. E. The Art of Computer Programming. Vol. 1: Fundamental Algorithms. Addison-Wesley, Reading, Mass., 1968.

[Knuth74] Knuth, D. E. Structured programming with GO TO statements. Computing Surveys 6,4 (December 1974), 261-3Ø1.

[Leavenworth66] Leavenworth, B. M. Syntax macros and extended translation. Comm. ACM 9,11 (November 1966), 79Ø-93.

[Liskov74] Liskov, B. and Zilles, S. Programming with abstract data types. ACM SIGPLAN Notices 9,4 (April 1974), 5Ø-59.

[Mealy63] Mealy, G. H. A generalized assembly system. Mem. RM-3646-PR, The Rand Corp., Santa Monica, Calif., 1963. Excerpts in Rosen, S., ed. Programming Systems and Languages. McGraw-Hill, New York, 1967, 535-59.

[Miller71] Miller, P. L. Automatic creation of a code generator from a machine description. Project MAC TR-85, Massachusetts Institute of Technology, 1971.

[McCarthy6Ø] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. Comm. ACM 3,4 (April 196Ø), 184-95.

[McKeeman7Ø] McKeeman, W. M., Horning, J. J., and Wortman, D. B. A Compiler Generator. Prentice-Hall, Englewood Cliffs, N.J., 197Ø.

[NATO69] Naur, P. and Randell, B., eds. Software Engineering. NATO Science Committee, Brussels, Belgium, 1969.

[NATO7Ø] Buxton, J. N. and Randell, B., eds. Software Engineering Techniques. NATO Science Committee, Brussels, Belgium, 197Ø.

[Newcomer] Newcomer, J. Machine independent generation of optimal local code. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University, in preparation.

[Olson] Olson, A. K. SLEUTH II 11Ø7 Case version. Case Institute of Technology. (Mimeograph.)

[Parnas72a] Parnas, D. L. A technique for software module specification with examples. Comm. ACM 15,5 (May 1972), 33Ø-36.

[Parnas72b] Parnas, D. L. On the criteria used in decomposing systems into modules. Comm. ACM 15,12 (December 1972), 1Ø53-58.

[Perlis72] Perlis, A. J. Introduction to Computer Science. Prelim. ed. Harper & Row, New York, 1972.

[Richards71] Richards, M. The portability of the BCPL compiler. Software -- Practice and Experience 1 (1971), 135-46.

[Richards74] Richards, M. Bootstrapping the BCPL compiler using INTCODE. In van der Poel, W. L. and Maarssen, L. A., eds. Machine Oriented Higher Level Languages. North-Holland, London, 1974, 265-7Ø.

[Ritchie74] Ritchie, D. M. and Thompson, K. The UNIX time-sharing system. Comm. ACM 17,7 (July 1974), 365-75.

[Ruth74] Ruth, G. R. Analysis of algorithm implementations. Project MAC TR-13Ø, Massachusetts Institute of Technology, 1974.

[SDS67] Scientific Data Systems Reference Manual: SDS 93Ø Computer. Scientific Data Systems Inc., Santa Monica, Calif., 1967.

[SIGPLAN74] Proceedings of a symposium on very high level languages. ACM SIGPLAN Notices 9,4 (April 1974).

[Sklansky68] Sklansky, J., Finkelstein, M., and Russell, E. C. A formalism for program translation. J. ACM 15,2 (April 1968), 165-75.

[Siewiorek74] Siewiorek, D. Introducing ISP. IEEE Computer 7,12 (December 1974), 39-41.

[Univac] Univac 11Ø7 SLEUTH II Programmer's Reference Guide. Pub. UP-367Ø Rev. 1, Univac Division of Sperry Rand Corp.

[Walter] Walter, K. Report on the Computer Sciences Corporation 11Ø7 assembler SLEUTH II. Case Institute of Technology. (Mimeograph.)

[Wegbreit74] Wegbreit, B. The treatment of data types in EL1. Comm. ACM 17,5 (May 1974), 251-64.

[Weingart73] Weingart, S. W. An efficient and systematic method of compiler code generation. Ph.D. thesis, Dept. of Computer Science, Yale University, 1973.

[Wick75a] Wick, J. D. A lexical analyzer generator. Res. Rept. 44, Dept. of Computer Science, Yale University, 1975.

[Wick75b] Wick, J. D. A parser generator. Res. Rept. 45, Dept. of Computer Science, Yale University, 1975.

[Wulf74] Wulf, W. A. ALPHARD: towards a language to support structured programs. Dept. of Computer Science, Carnegie-Mellon University, 1974.