

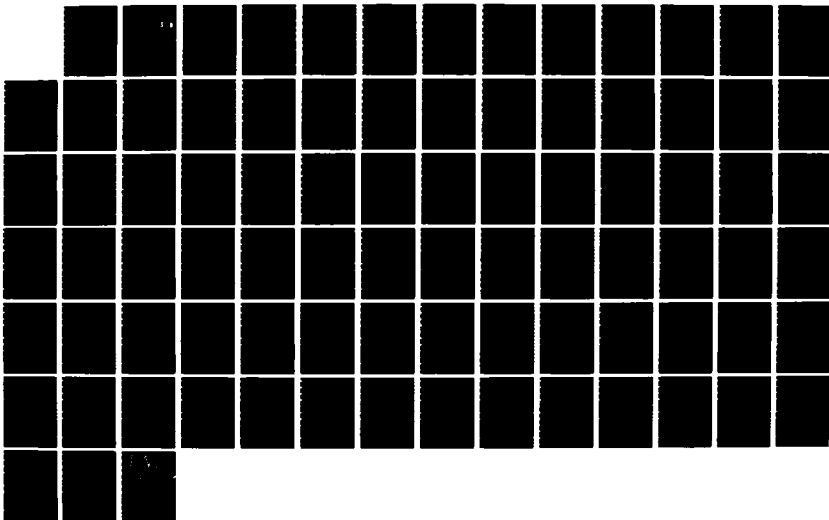
NO-A181 093

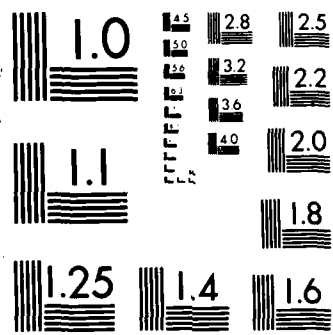
A MICROCODE COMPILER THAT RUNS ON THE IBM AT(U) PENGUIN 1/1
SOFTWARE INC LONG BEACH CA T H WEIGHT 22 APR 87
DRA010-86-C-0008

UNCLASSIFIED

F/G 12/5

NL





DTIC FILE COPY

1

AD-A181 093

FINAL REPORT

A MICROCODE COMPILER THAT RUNS ON THE IBM AT

by

Thomas H. Weight, Ph.D.

DTIC
ELECTE
MAY 26 1987
S R D

Period Covered: 24 sept 86 to 23 Mar 87

Contract DAAD10-86-C-0008

for

White Sands Missile Range, New Mexico 88002

22 Apr 87

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

PENGUIN SOFTWARE, Inc.
7005 E. Spring St.
Long Beach, Calif. 90808

Thomas H. Weight

Thomas H. Weight
Principal Investigator
PENGUIN SOFTWARE, Inc.

FINAL REPORT

A MICROCODE COMPILER THAT RUNS ON THE IBM AT

by

Thomas H. Weight, Ph.D.

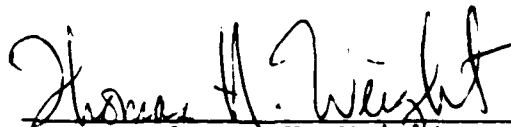
Period Covered: 24 sept 86 to 23 Mar 87

Contract DAAD10-86-C-0008
for

White Sands Missile Range, New Mexico 88002

22 Apr 87

PENGUIN SOFTWARE, Inc.
7005 E. Spring St.
Long Beach, Calif. 90808



Thomas H. Weight
Principal Investigator
PENGUIN SOFTWARE, Inc.



The views, opinions, and findings contained in this report are those of the author and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

By	
Date	
Reviewed by	
Approved by	
Special	
A-1	

REPORT DOCUMENTATION PAGE

A181093

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS -----										
2a. SECURITY CLASSIFICATION AUTHORITY -----			3. DISTRIBUTION / AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.										
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE -----													
4. PERFORMING ORGANIZATION REPORT NUMBER(S) -----			5. MONITORING ORGANIZATION REPORT NUMBER(S) -----										
6a. NAME OF PERFORMING ORGANIZATION PENGUIN SOFTWARE, INC.		6b. OFFICE SYMBOL (if applicable) -----	7a. NAME OF MONITORING ORGANIZATION U.S. ARMY WHITE SANDS MISSILE RANGE										
6c. ADDRESS (City, State, and ZIP Code) 7005 E. Spring St. Long Beach, Ca 90808			7b. ADDRESS (City, State, and ZIP Code) COMMANDING OFFICER, STEWS-ID-T U.S. ARMY WHITE SANDS MISSILE RANGE NEW MEXICO 88002-5143										
8a. NAME OF FUNDING / SPONSORING ORGANIZATION -----		8b. OFFICE SYMBOL (if applicable) -----	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAD10-86-C-0008										
8c. ADDRESS (City, State, and ZIP Code) -----			10. SOURCE OF FUNDING NUMBERS <table border="1"><tr><td>PROGRAM ELEMENT NO 665502</td><td>PROJECT NO 1P65502 M40</td><td>TASK NO -----</td><td>WORK UNIT ACCESSION NO -----</td></tr></table>		PROGRAM ELEMENT NO 665502	PROJECT NO 1P65502 M40	TASK NO -----	WORK UNIT ACCESSION NO -----					
PROGRAM ELEMENT NO 665502	PROJECT NO 1P65502 M40	TASK NO -----	WORK UNIT ACCESSION NO -----										
11. TITLE (Include Security Classification) A MICROCODE COMPILER THAT RUNS ON THE IBM AT													
12. PERSONAL AUTHOR(S) Thomas H. Weight													
13a. TYPE OF REPORT Final Technical		13b. TIME COVERED 24SEP86 TO 23MAR87		14. DATE OF REPORT (Year, Month, Day) 1987, APRIL, 22									
15. PAGE COUNT 82													
16. SUPPLEMENTARY NOTATION -----													
17. COSAT CODES <table border="1"><tr><th>FIELD</th><th>GROUP</th><th>SUB-GROUP</th></tr><tr><td>-----</td><td>-----</td><td>-----</td></tr><tr><td>-----</td><td>-----</td><td>-----</td></tr></table>			FIELD	GROUP	SUB-GROUP	-----	-----	-----	-----	-----	-----	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) High-level language, microprogramming, automated microcode generation, microcompiler	
FIELD	GROUP	SUB-GROUP											
-----	-----	-----											
-----	-----	-----											
19. ABSTRACT (Continue on reverse if necessary and identify by block number) PENGUIN SOFTWARE, Inc has demonstrated the feasibility of supporting microcode development with a retargetable microcode compiler. This approach avoids many of the problems associated with microcode compilers by allowing the designer to specify an efficient language tailored to the requirements of a particular computer design and programming algorithm. In particular, we avoid the issues of code compaction, and resource allocation. Prototype compilers were generated which have the capability of supporting a broad class of different languages, and generating microcode for virtually any digital hardware architecture. Based on the performance of these compilers, several recommendations have been made for future enhancements. One especially important modification is necessary to support cascaded, systolic, and parallel arrays of microprocessors. Another change concerns using HILEVEL Technologies HALE meta-assembly language as an intermediate language.													
20. DISTRIBUTION AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED										
22a. NAME OF RESPONSIBLE INDIVIDUAL FOO LAM			22b. TELEPHONE (Include Area Code) 705-678-3010	22c. OFFICE SYMBOL STWS-ID-T									

intentionally left blank

SUMMARY

Critical needs currently exist for a fully retargetable microcode compiler and for an effective development tool to support programming for the new non-von Neuman architecture microprocessors. As more parallel processors, systolic arrays, and cascadable processors become available, these needs can only become more acute. Most of the chips resulting from the VHSIC program and all near-term gallium arsenide devices will require microcoding. For the benefits of these advances to be fully realized, microcode development will have to be automated in the relatively near future.

This report describes the microcode compiler feasibility study and the compiler development undertaken at PENGUIN SOFTWARE, Inc. under the first phase of contract DAAD10-86-C-0008 sponsored by White Sands Missile Range.

The microcode compiler being developed at PENGUIN Software, Inc. is based on the concept of a retargetable compiler. This approach does not have a fixed machine-independent language, but allows the user to develop a language specific to each particular target machine. This provides a means for the user to incorporate knowledge of target machine design into the language definition, and avoids the necessity for resource allocation or code compaction in the application program.

This microcode compiler is a new type of program. According to one source, this approach has never been tried before; at least no other retargetable microcode compiler exists. The advantages of this approach appear to be:

1. Low Risk : there are no high risk algorithms remaining to be developed.
2. Robustness : This compiler can support any processor architecture supported by the current state-of-the-art meta-assemblers.
3. Speed : This compiler is competitive with meta-assemblers in terms of speed. This is in sharp contrast to other microcompiler designs.
4. Timeliness : this approach can be developed into a working tool quickly.

Currently, PENGUIN Software, Inc. is involved in a continuing development effort to produce a commercially marketable microcompiler which is capable of supporting the development of microcode for cascadable, parallel, and systolic arrays of microprocessors.

PREFACE

Previous microcode compilers have evolved from traditional compiler concepts and design methodologies. They have adopted an approach involving machine-independent languages and have attempted to capitalize on the associated advantages. In order for this to be successful, it is necessary to develop algorithms capable of producing compact microcode. This approach invariably gets bogged down in the problems associated with microcode compaction and resource allocation. Until these problems are solved, this approach has to be considered very high risk.

In contrast, the microcompiler being designed at PENGUIN SOFTWARE, Inc. has evolved from the classic microprogramming tool i.e. the meta-assembler. Our microcompiler starts out with an underlying meta-assembler and builds up a higher level language capability around it. This approach results in a microcode development tool which is a very low risk, and is capable of supporting virtually any digital hardware architecture.

While there are very few research journal articles related to our work, we have not been forced to work in a vacuum. We would like to thank the marketing staff at HILEVEL Technology in Irvine, and we would especially like to thank Mr. Warren Long, Product Marketing Manager at HILEVEL, for his many suggestions in the areas of requirements and engineering design. We would also like to thank the engineers and managers of Rockwell International, Hughes, Northrop, and TRW that have contributed their time and allowed themselves to be interviewed.

TABLE OF CONTENTS

	page
1.0 INTRODUCTION.....	1
2.0 MICROCODE COMPILER STUDY.....	4
2.1 Define Machine Dependent Languages.....	4
2.1.1 Statement Notation.....	4
2.1.2 Entity Notation.....	5
2.1.3 MACRO's.....	5
2.1.4 Structured Control.....	5
2.1.5 Phase II Features.....	5
2.1.6 Beyond Phase II.....	7
2.2 Define Syntax Definition Language.....	8
2.2.1 Syntax Definition.....	8
2.2.2 Semantics Definition.....	11
2.3 Generate Testcases.....	15
2.4 Generate Design Requirements.....	16
2.4.1 Special Characters.....	16
2.4.2 Sizing Parameters.....	16
2.4.3 Input/Output.....	17
2.5 Design Syntax Compiler Mods.....	17
2.6 Add Diagnostic Function.....	17
2.7 Add Syntax File Inversion.....	18
2.8 Design Microcompiler Mods.....	19
2.9 Perform Timing Study.....	19
3.0 STATUS OF ACCOMPLISHMENTS.....	21
3.1 Finalize Language Structure.....	21
3.2 Insert MACRO Capability.....	21
3.3 Optimize Syntax File Data Structures.....	21
3.4 Generate Skeleton Compilers.....	21
3.5 Perform Timing Study.....	22
3.6 Future Enhancements.....	22
4.0 CONCLUSIONS AND RECOMMENDATIONS.....	22
4.1 Conclusions.....	22
4.1.1 Tasks Completed and Objectives Attained.....	22
4.1.2 Low Risk Approach.....	22
4.1.3 High Speed.....	23
4.1.4 Graceful Degradation.....	23
4.2 Recommendations.....	23
4.2.1 Semantic Action Engine.....	23
4.2.2 Intermediate Language.....	24
4.2.3 Speed Becomes a Resource.....	24

APPENDIX A SYNTAX CHARTS
APPENDIX B SAMPLE TEST CASE
APPENDIX C DESIGN REQUIREMENTS
APPENDIX D PDL LISTINGS

LIST OF TABLES

Figure No.		Page No
1	List Notation Format.....	6
2	Compiler Sentence Format.....	6
3	CLAUSE Definition Format.....	6
4	Example: Literal Strings Definitions.....	9
5	Example: When Not to Use Literal Strings.....	9
6	Simple LIST Definition Format.....	10
7	Example: Using LIST Definition.....	10
8	Example: When Not to Use Simple LIST's.....	12
9	Dimensioned LIST Definition Format.....	12
10	Example: Using Dimensioned LIST.....	12
11	LITERAL Definition Format.....	13
12	Example: Using LITERAL.....	13
13	Mapping Semantic Actions to Constructs.....	14

1.0 INTRODUCTION

The microcode compiler being developed at PENGUIN SOFTWARE, Inc. is a retargetable compiler. This compiler does not have a fixed machine-independent language, but allows the user to develop a language specific to each particular target machine.

The microcode compiler is a two stage translation scheme. The first stage provides a fixed high level design language to permit the easy definition of the machine-dependent high level language. The language definition compiler, called the Syntax Compiler, produces syntax files which define both the syntax and semantics of the target machine language.

The application program is compiled by the MICROCOMPILER to produce the object file microcode. The application programs are written in a language specifically designed for the particular target processor. This language is almost completely free from constraints with just enough underlying language structure to facilitate the application language compilation process.

One of the crucial issues decided by this PHASE I study was the speed of the microcompiler. Because of the complexity of the algorithms being used, a great deal of effort was focused on enhancing the speed of the microcompiler. This microcode compiler has been designed to run very fast by using our own in-house developed data structures and algorithms, and using "inverted" syntax files.

The approach to automated microcode generation being developed at PENGUIN SOFTWARE, Inc. is based on the concept of machine-dependent languages. This provides a means for the user to incorporate knowledge of target machine design into the language definition. There are some obvious advantages and disadvantages to this approach.

Some of the advantages of this approach are expressiveness, robustness, and low risk.

- a. This approach allows the programmer to design a machine-dependent language tailored to the needs of a particular application. This can result in a highly expressive and efficient language.

- b. Because of this microcompiler's relationship to meta-assemblers, this microcompiler is extremely unlikely to fail due to a particular target machine design.

c. Finally, this approach avoids the tough problems of microcode compaction and resource allocation.

Some of the disadvantages of this approach are lack of a stable application language, and the difficulties associated with trying to imbed low level hardware details in a high level language.

a. A stable application language is desirable because it makes application programs more transportable and facilitates the development of microcode simulators.

b. Because of the hardware design, it may be difficult to design a suitable high-level language for a particular application.

The basic requirement of our Phase I effort was to demonstrate the feasibility of PENGUIN SOFTWARE's approach microcode development. Section 3.0 describes, in some detail, the research and development efforts expended at PENGUIN SOFTWARE to meet this requirement. Under the current contract, the following tasks were completed at PENGUIN SOFTWARE, Inc.

a. Define the Machine-Dependent Languages.

Specify the general class of statements to be supported by our microcompiler. This specification is to be based on a literature search, interviews with managers and engineers from the microprogramming community, and the flexibility of the existing LR(n) compiler. (See Appendix A.)

b. Define Syntax Definition Language.

Specify the general class of statements to be supported by the syntax compiler. This specification is to be based on the design of the existing compilers, the requirements generated for the application languages, and the flexibility of the existing LR(1) compiler. (See Appendix A.)

c. Generate Test Cases.

Generate typical application program statements and test programs to be used to evaluate the expressiveness of the proposed languages. (See Appendix B.)

d. Generate Microcompiler and Syntax Compiler Design Requirements.

Generate design requirements including identifying reserved characters, parameter sizing, input/output and listing formats. (See Appendix C.)

e. Design Syntax Compiler modifications.

Generate PDL (Program Design Language) descriptions of changes to be made to the syntax compiler. (See Appendix D.)

f. Add Diagnostic Function.

Insert a diagnostic capability into the syntax compiler in order to help debugging the syntax source files.

g. Add Syntax File Inversion.

The files which define the application language syntax will be inverted to facilitate the process of parsing the application program statements.

h. Design Microcompiler Modifications.

Generate PDL (Program Design Language) descriptions of changes to be made to the microcompiler. (See Appendix D.)

i. Perform Timing Studies.

Perform a timing study of the syntax definition phase and the microcompiler phase.

Section 4.0 describes the status of the technical objectives. This includes the extent to which the language structure was finalized. The optimization of the data structures was completed with the result that the microcompiler is now much faster than predicted. The construction of the prototype compilers was concluded without surfacing any difficult problems. Finally, the completion of the prototypes allowed us to perform the necessary timing study.

Section 5.0 gives a summary of conclusions and recommendations. We conclude that all technical objectives were attained. The results of the timing study show that the microcompiler is quite fast enough. Although this general approach is new, it appears to be the low risk approach now that all of the difficult algorithms have been demonstrated in our prototype. Finally, we conclude that this approach has the added benefit of being very robust in the sense that it is very unlikely that any future design will cause the microcompiler to fail.

2.0 MICROCODE COMPILER STUDY

PENGUIN SOFTWARE's retargetable microcode generator system emulates a wide range of computer architectures. By allowing the user to define a machine-dependent high-level language, and then write an application program in this new language, the microcode generator is able to translate the application program into efficient horizontal microcode.

Our minimum goal was to produce a working microcode compiler which would be a useful tool in industry. In order to achieve this goal, we determined to start with the most powerful meta-assembler that we could define and then enhance its capabilities to achieve a higher-level language capability. In retrospect it makes sense that many of the project tasks are divided into two parts. The first part, the syntactic analysis, determines which statements the microcompiler will accept as valid, and determines the readability of the language. The second part, the semantic action, determines to a large extent the level of the language.

2.1 Define Machine-Dependent Languages.

Having proposed using machine-dependent languages, we now had to decide if we would use a machine-independent language (such as a subset of "C" or ISP) which could be augmented, or just provide a minimal language structure.

Based on interviews with various engineers and managers in the microprogramming community, we naturally found a conflicting set of requirements. Although there was a lot of support for using a subset of "C", there was also a lot of negative feeling associated with the implied technical requirements associated with that decision. In the end, we decided to opt for providing the minimal language structure, and in light of the greater understanding we now have concerning the project, we believe that this was definitely the right choice.

Our literature search provided a list of candidate meta-assembler capabilities for the microcode compiler. These meta-assembler-derived capabilities were reviewed to determine which were inappropriate, which would be postponed, and which would be implemented immediately. The results of the study are summarized below.

2.1.1 Statement Notation.

Two types of statement notation were considered. The first type, "action verb", is similar to the standard assembly language statement. Unfortunately, "action verb" does not

seem compatible with the requirements of horizontal microprogramming. The second type of statement notation, "list notation", is ideal for horizontal microcoding. This statement has the basic form shown in Fig. 1. We have adopted the convention that all labels are immediately followed by a colon, the entities are separated by semicolons, and the last entity is followed by a period. Everything to the right of the period is a comment.

2.1.2 Entity Notation.

The lowest level of entity notation identified is "positional notation". In positional notation each entity is related to one particular field. This notation does not appear to have any role in our high-level language except possibly to force specific values into a specific microword, and even then there seems to be better ways to accomplish that.

Several nonpositional entity notations were identified. These include "function reference notation", "value mnemonic notation", and "keyword notation". All of these notations are desirable in different applications. Rather than decide on any one of them, a different notation has been selected which includes all of the above: this is the "free notation". In the free notation there are almost no constraints on the notation used. The only constraints imposed on the entity notation are those derived from the design requirements and from the implementation of the program.

2.1.3 MACRO's

One of the advantages of using a high-level language is the ability to generate more than one line of object code for one line of source code. This capability has been included in the microcode compiler.

2.1.4 Structured Control Statements

Structured control statements allow the development of block structured code and an orderly and controlled approach to program control flow. Structured control is included as an important part of the microcode compiler.

2.1.5 Phase II Features

The main objective of our Phase I study was to prove the feasibility of our approach. Namely, to prove the feasibility of a retargetable microcode compiler that runs on the IBM-AT in a reasonable amount of time. Several features were identified which, although desirable for a production quality product, were judged to be non-essential for attaining our

`<label>            <entity> <entity> ... <entity>       <comment>`

Figure 1. List Notation Format.

`<label>: <clause>;<clause>; ... ;<clause>.<comment>`

Figure 2. Compiler Sentence Format.

`CLAUSE    {<syntax definition>}[<semantic actions>]`

Figure 3. CLAUSE Definition Format.

Phase I objective. These features were postponed until Phase II.

a. Compiler evaluation of arithmetic expressions.

The ability to evaluate arithmetic expressions is not relevant to the Phase I feasibility study.

b. Compile time MACRO definition.

The capability to generate MACRO's in the application program is usually associated with assemblers rather than compilers. Still, this capability might be valuable in the event that the syntax compiler were too slow. (Considering the results of the timing study, it now seems less likely that this MACRO capability will be included in Phase II.)

c. Conditional compilation.

Some form of conditional compilation is part of one possible approach to handling cascadable microprocessors in PHASE II.

d. Relocatable Object code.

e. Linking of object modules.

These two capabilities are essential and will be included in PHASE II as a consequence of our approach based on using the HILEVEL Technology meta-assembly language, HALE, as a intermediate language.

2.1.6 Beyond Phase II

The following features were identified as being too high risk for inclusion in either Phase I or Phase II. They did, however, cause a shift in design philosophy which will be reflected in the Phase II design.

a. Microcode compaction.

Although the need for microcode compaction has been almost entirely avoided by the use of machine-dependent languages, it may be desirable to have a local compaction algorithm as part of a compiler optimization phase.

b. Compiler-directed resource allocation.

Resource allocation is another capability which we might eventually want to include in this compiler. Resource allocation is a difficult task chiefly in a situation involving global microcode compaction. By avoiding global compaction, we will eventually be able to include some

resource allocation capability in this compiler.

2.2 Define Syntax Definition Language.

Because of the period and semicolons in the list notation, we have adopted the terms "sentence" for the entire line from the label to the end of the comment, and the term "clause" for the entity as shown in Fig. 2. Since the structure of the sentence is hard-wired into the microcode compiler, it remains to the syntax compiler to define the syntax and semantics of the "clauses". A syntax compiler source file is constructed using the language outlined below. This file will define all of the legal application language clauses and define what these clauses mean. The basic clause definition appears as shown in Fig. 3.

2.2.1 Syntax Definition

The syntax definition process for each clause is accomplished by using three related constructs: literal strings, lists, and literals.

Literal strings are the simplest and in some cases the most efficient means for defining the syntax of a clause. A literal string is string of alphabetic or special characters enclosed in double quotes. Numerical digits can also be included, but they must be preceded by an alphabetic character. Ignoring the semantic definitions for the moment, the examples as shown in Fig. 4 show clauses defined using the literal string.

Literal strings are useful for clauses, such as "NOP", which are, in a sense, one-of-a-kind. Literal strings are not very useful in situations like "REG1 = REG12" where there are several registers that may appear on either side of the equation such as shown in Fig. 5. This situation is handled by the two types of lists described below.

First, the LIST is identified by a list name, then the LIST is defined by a list of tokens and associated semantics actions. Fig. 6 is a simple example of a LIST structure. A LIST is used in a CLAUSE definition simply by using the LIST name. Using a LIST in a CLAUSE definition can considerably reduce the amount of effort required as can be seen by the example in Fig. 7. In this example, we have a very simple situation where the contents of any source register can be moved into any destination register. Experience has shown that quite often these LIST's will be used repeatedly to define other CLAUSE's. This results in a tremendous saving of programming effort.

```

CLAUSE { "NOP" }

CLAUSE { "REG1 = REG12" }

CLAUSE { "REPEAT(FOREVER)" }

```

Figure 4. Example: Literal String Definitions.

```

CLAUSE { "REG1 = REG1" }
CLAUSE { "REG2 = REG1" }
CLAUSE { "REG3 = REG1" }
.
.
.
CLAUSE { "REG1F = REG1" }
CLAUSE { "REG1 = REG2" }
CLAUSE { "REG2 = REG2" }
.
.
.
CLAUSE { "REG1F = REG2" }
CLAUSE { "REG1 = REG3" }
.
.
.
CLAUSE { "REG1F = REG1F" }

```

Figure 5. Example: When Not to Use Literal Strings.

```

LIST  <list name> = {
        <token-1>[semantic action];
        <token-2>[semantic action];
        .
        .
        <token-n>[semantic action];
    }

```

Figure 6. LIST Definition Format.

```

CLAUSE { REGD "=" REGS }

LIST REGD = {
    REG1[semantic action];
    REG2[semantic action];
    REG3[semantic action];
    .
    .
    REG1F[semantic action]
}
LIST REGS = {
    REG1[semantic action];
    REG2[semantic action];
    REG3[semantic action];
    .
    .
    REG1F[semantic action]
}

```

Figure 7. Example: Using LIST Definition.

Occasionally, we have a situation, unlike the preceding example, where the source and destination registers are not orthogonal. This means, in this case, that only certain source registers can be used with certain corresponding destination registers. The result is that simple LIST's would no longer work efficiently. Now we could fall back on the literal strings discussed before and define several CLAUSE's as shown in Fig. 8. We have adopted an approach using dimensioned LIST's. Dimensioned LIST's have the particular property that if more than one column of the LIST is used in a CLAUSE definition, then the tokens must all come from the same row. The definition of a dimensioned LIST is shown in Fig. 9. A dimensioned list is used to solve the CLAUSE definition problem discussed above as shown in Fig. 10.

The final construct is the LITERAL. LITERAL's are used to define the processing of numbers and statement labels. The LITERAL is defined as shown in Fig. 11 and is used in a clause as shown in Fig. 12. The LITERAL allows the application programmer to include values in the program at compile time.

2.2.2 Semantics Definition

The semantics of a particular statement is defined in terms of the semantic actions to be performed when that statement is encountered in an application program. For each application program statement, the microcompiler must reconstruct the statement definition to determine the particular CLAUSE, LIST row, and LITERAL as appropriate. These semantic actions are associated with the CLAUSE's, rows of the LIST's, and the LITERAL's.

Some semantic actions only make sense in association with certain constructs, while others have only been implemented for other constructs. This is shown in Fig. 13.

The semantic actions implemented during PHASE I are described below:

- a. field value: a value is inserted into a field in the object file.
- b. PPA: an address is popped off an internal compiler stack. A label is generated and associated with the value just popped off the stack.
- c. PHA: the current microaddress counter is pushed onto an internal compiler stack.
- d. PPL: a label is popped off an internal compiler stack

```

CLAUSE { "REG1 = REG5" }
CLAUSE { "REG1 = REG7" }
CLAUSE { "REG2 = REG1" }
CLAUSE { "REG3 = REG5" }
.
.
.
CLAUSE { "REF1F = REG12" }

```

Figure 8. Example: When Not to Use Simple LIST's.

```

LIST <list name>(dim-1,dim-2,...,dim-m) = {
  <token-11,token-12,...,token-1m>[semantic action];
  <token-21,token-22,...,token-2m>[semantic action];
  .
  .
  .
  <token-n1,token-n2,...,token-nm>[semantic action];
}

```

Figure 9. Dimensioned LIST Definition Format.

```

CLAUSE { REGS(1) "=" REGS(2) }

LIST REGS(1,2) = {
  REG1,REG5[semantic action];
  REG1,REG7[semantic action];
  REG2,REG1[semantic action];
  REG3,REG5[semantic action];
  .
  .
  .
  REF1F,REG12[semantic action]
}

```

Figure 10. Example: Using Dimensioned LIST.

LITERAL(<literal number>) = [semantic action]

Figure 11. LITERAL Definition Format.

```
CLAUSE { REGS "=" LITERAL(1) }  
LITERAL(1) = [semantic action]  
LIST REGS = as defined above.
```

Figure 12. Example: Using LITERAL.

	CLAUSE	LIST	LITERAL
field value	yes	yes	yes
PPA	yes	no	no
PHA	yes	no	no
PPL	yes	no	no
PHL	yes	no	no
SWP	yes	no	no
LBL	yes	no	no
LIT	no	no	yes
ROR(n)	no	no	yes

Figure 13. Mapping Semantic Actions to Constructs.

and associated with the current value of the microaddress counter.

e. PHL: a label is created and pushed onto an internal compiler stack.

f. SWP: the top two entries on an internal compiler stack are swapped.

g. LBL: causes the compiler to generate a label and retrieve the value associated with this label. This value is processed as a literal (see LIT below).

h. LIT: a literal value is right justified, and zero filled as required. The result is then inserted into a field in the object file.

i. ROR(n): a literal number is conditioned as with LIT above then rotated n bits. The result is then inserted into a field in the object file.

2.3 Generate Test Cases.

Test cases were generated to satisfy three requirements: to debug the program; to perform the compiler timing studies; and to demonstrate the capabilities of the microcompiler.

The debugging effort was automated as much as possible to encourage and facilitate frequent testing. Both a test file(or files) and an answer file were created. DOS commands were then issued from a ".BAT" file to automatically run a test case and verify the results.

Test files were constructed for use in the timing studies. The Syntax compiler was timed with one large and one small test case, while the microcompiler was timed with a large, a medium, and a small test case for each of the syntax compiler output files. As it turned out both compilers were much faster than predicted, and both compilers could have been adequately characterized by a single large test case.

The third type of test case was designed to demonstrate the expressive power of the languages that could be supported by the microcompiler. No effort was made to create either a realistic language, or a complete one. But all of the different semantic actions supported by the microcompiler are exercised, and all of the supported statement types are included.

2.4 Generate Design Requirements.

In addition to implementing the syntax and semantics of the microcompiler, and the syntax compiler languages, several implementation-specific requirements had to be defined. These were associated with reserved characters, parameter sizing, and input/output.

2.4.1 Reserving Special Characters

After identifying certain characters to be used in the application language syntax, and others to be used in the syntax definition language, the remaining characters were left for inclusion in the application language itself. These remaining characters could be treated in at least two ways. First, each special character could be treated as a separate token. Thus, expressions like

REGA = REGB.

would be treated identically to

REGA=REGB.

where the spaces have been deleted. The second approach would be to treat special characters the same as alphabetic characters, and thus allow them to be embedded inside a token. This is commonly the cases with the underscore and is used as an embedded blank as in the following:

REG_A = REG_B.

We decided to treat all special characters as tokens for Phase I. Phase II will start treating various special characters in different ways.

2.4.2 Sizing Parameters

Designing software involves a certain amount of defining data structures. The microword is an example of a data structure that has to be sized. During the microcompiler processing a microword is built up by first setting it equal to a default value. Then various fields of the microword are modified as specified by the commanded semantic actions. One obvious question is "How wide should this microword be?".

There are at least three possible answers to this question. First, a dynamic resource allocation scheme could be used. This approach would involve requesting RAM allocation as required from the operating system during run time. With this approach there would be no need to know the size of the

problem in advance, and no limit on problem size would be imposed by the software. The hardware would limit problem size. In the event that the software required more RAM than was available, then the user could simply add more RAM space to the system.

The second approach simply sets a "hard-wired" limit into the software. This approach is especially appropriate when there is a simple work-around which allows the limits to be essentially exceeded. For example, the maximum size of a field might be set at sixteen bits. Thirty two(or more) bits are easily achieved by breaking a large field up into several smaller fields.

The third approach is sort of a combination of the preceeding two. A substantial block of RAM space is reserved in the software. This RAM is then allocated according to the values of certain parameters input as part of the application program. For example, the microword width could be specified in the application program, and sufficient RAM allocated as required.

During Phase I, we have used the last two approaches. These were essentially inherited from the previous design. The dynamic resource allocation scheme has some tremendous advantages, but represents a much more difficult programming task. The dynamic resource allocation scheme has been postponed and proposed for Phase II.

2.4.3 Input/Output

The input and output interfaces were designed to be compatible with the existing software, and as simple as possible. No attempt was made to provide a realistic operator interface, because the operator interface was considered to be irrelevant to the Phase I technical objectives.

2.5 Design Syntax Compiler Mods.

An existing LR(1) compiler was modified to provide the syntax compiler needed for our study.

2.6 Add Diagnostic Function.

A diagnostic capability was added in order to help debug the syntax compiler test cases. Since the syntax compiler was LR(1), one token look ahead, there was very little difficulty anticipating the next token and flagging an error if the token was not supplied. In the event of an error, an attempt was made to identify the missing token and to indicate on the listing where it was expected. Occasionally, a neighboring

location was mistakenly identified, but it was sufficiently accurate and reliable in order to enable rapid debugging of the larger test cases.

2.7 Add Syntax File Inversion.

The syntax compiler produced two types of files: syntax and semantic. The syntax files identify the various acceptable clauses, while the semantic files describe what semantic actions to take when each clause is encountered. Thus the microcompiler engages in a two step process. First a clause is read and identified, and next the appropriate semantic actions are taken. When the semantic actions are defined, they are built up into files which are in the most efficient form for use by the microcompiler. This is not the same for the syntax files.

When the syntax files are defined in the syntax compiler, they are defined in some sense from the top down. The general forms of the clauses are defined, and then the particulars of the tokens are supplied later. The microcompiler, on the other hand, encounters the individual tokens first, and then must work its way back to identify the basic clause. In this sense, the syntax files need to be inverted in order for the microcompiler to run most efficiently.

The inversion process used consisted of first expanding the packed data structures into arrays. These arrays were arranged with what was originally the independent variables in the first columns and the dependent variables in the last columns. These arrays were then inverted using a "quick sort", and the old dependent variables became the new independent variables, etc. The resulting "inverted" arrays were then packed into the original data structures to be used by the microcompiler.

As a final step, the syntax and semantics files were all packed together, and saved in a disk file. This avoids the necessity of running the syntax compiler everytime we run the microcompiler. It did, however, increase the complexity of the data being passed to the microcompiler (and resulted, temporarily, in considerable confusion).

Because of the complexity of these inverted data structures, it was difficult to verify their accuracy. This was accomplished by two methods. First, the data structures were printed out before they were packed into the disk file by the syntax compiler, and printed out after they were unpacked by the microcompiler. Second, small cases were inverted and packed by hand, and verified against the results of a syntax compiler run.

2.8 Design Microcompiler Mods.

An existing LR(n) compiler was modified to study the microcompiler requirements. We are using the term LR(n) to mean a compiler that can perform up to an "n" token look-ahead before parsing a particular sentence. This parameter "n" is determined when the "C" source program for the Syntax Compiler and Microcompiler are themselves compiled.

2.9 Perform Timing Study.

Our intention for the timing study was to perform a parametric analysis which could possibly lead to some speed enhancing design changes. Our concerns were that the algorithm's complexity combined with a long test case would result in unacceptable run times. We were also expecting a non-linear effect in the microcompiler run times to cause problems for larger test cases and to limit future applications. Both of these concerns were unfounded.

The Syntax Compiler requires only nine seconds for the largest test case that we ran. This test case was based on a large real world language which includes supporting the AMD 29116 chip. Because of the AMD 29116's extreme encoding, it represents a "worse case" chip for the purposes of this compiler study. Since the Syntax Compiler is so fast, we are now free to consider various operational concepts for the microcode development system. For example, we could predefine the syntax in a separate run from the application program, or we could recompile the syntax everytime we compile the application program. Also, defining MACRO's in the application program is not important when we can define them in the syntax at so little cost in run time.

The results of the microcode compiler timing study were even more surprising. For purposes of analysis, the run times were displayed at various times during a run. The first time was associated with the initial processing of the syntax files. The other two times were associated with the two passes of the microcompiler. The syntax file "set-up" time was found to be constant and small. This time was found to be a function of the maximum parameters of the program, and was insensitive to syntax program size.

The function of the first phase of the compiler is to define statement labels and to perform any required semantic actions. These first-phase actions could be to generate internal addresses or internal labels, and to perform processing associated with the block program structures. The

function of the second phase of the microcompiler is to produce the object code.

We were expecting some non-linearities in run time resulting from our use of hash tables, and other complex data structures. We found no significant non-linear effects. The times were almost strictly linear functions of the test case size.

We were also expecting the microcompiler to run slower than the state-of-the-art meta-assemblers. This expectation was based on the complexity of the algorithms and on previous computer run times for a similiar compiler. Our results show that the microcompiler is about as fast as the state-of-the-art meta-assemblers.

Comparing microcompilers in this manner is far from scientific. For example, STEP Engineering, Inc. claims "2000 fields per minute" for their meta-assembler " on small machines" and " over 10,000 fields per minute on larger machines(i.e. VAX 11/750)." Without knowing their target processor, or even what they consider a "small machine" it is difficult to make a precise comparision. In any case, we claim to be compiling over 2300 fields per minute on our IBM/AT.

3.0 Status of Accomplishments

During this research period, a full scale prototype was specified, designed, and constructed. The purpose of this prototype was to demonstrate the feasibility of producing a retargetable microcode compiler. At the completion of the feasibility study, a complete set of validated algorithms and data structures were arrived at for all components of the microcode generator.

3.1 Finalize Language Structure

A two-stage approach was adopted which used two compilers: the first compiler is used to define the application language syntax, and the second compiler is used to compile the application language and produce the microcode object file. The syntax compiler language was finalized to the extent necessary for the feasibility study, while the application languages were defined sufficiently to last into Phase III. The results of this study indicate some possible areas of improvement in the syntax definition language.

3.2 Insert MACRO Capability

Because of time constraints and in the interest of economy, two existing compilers were modified to provide the prototypes necessary for this study. One of the major requirements, not satisfied by the existing compilers, was the requirement to define MACRO's as part of the language definition, and to expand these MACRO's during the application language compilation. This capability was successfully included in the feasibility study microcode compilers.

3.3 Optimize Syntax File Data Structures

The syntax compiler data structures were optimized in order to facilitate the operation of the microcompiler. This optimization consisted of the "inversion" of the data structures produced by the syntax compiler. This inversion was accomplished with the result that the microcode compiler runs much faster than it would otherwise. Without this "inversion" the microcode compiler would be considered computationally infeasible.

3.4 Generate Skeleton Compilers

All of the software for the feasibility study was coded and tested. Specifically, the two existing compilers were modified to process the languages described above, to process MACRO's defined during the syntax definition, and to use the

optimized syntax file data structures for enhanced speed.

3.5 Perform Timing Study

In addition to the usual testing to eliminate programming errors, the software was thoroughly characterized by a timing study. The focus of this study was to determine those test case parameters which could be increased to cause the program to fail through computational infeasibility.

3.6 Future Enhancements

Considerable progress has been made toward defining the capabilities necessary for the next phase of this project. Requirements have been generated for the modification of the syntax definition language which will allow the use of a wider range of semantic actions. The microcompiler will be redesigned to reflect our new understanding of the microcompiler as an engine for executing semantic actions. The speed of the microcompiler is sufficient that it can be "traded off" for complexity of the internal data structures, which could result in the program being able to handle much larger and more complicated problems.

4.0 Conclusions and Recommendations

4.1 Conclusions

A new and powerful approach to microcode development has been investigated at PENGUIN Software, Inc. under the contract DAAD10-86-C-0008 sponsored by White Sands Missile Range. This approach has required the development of two prototype compilers. The first compiler is used to design a machine-dependent language for a particular target machine. The definition of this language is contained in a syntax and semantic definition file. This syntax file is inverted and input to the microcompiler. The microcode compiler is used to convert application program source code into object file microcode.

4.1.1 Tasks Completed and Objectives Attained.

All proposed tasks have been completed and all technical objectives have been attained resulting a demonstration of the feasibility of our proposed approach.

4.1.2 Low Risk Approach

After running several test cases during the software debug, software validation, and timing study, we have gained

considerable confidence not only in the correctness of the software but also in the validity of our approach. This being a new approach, it had several conjectured disadvantages. We have found that there are indeed no undeveloped algorithms waiting to trip us up, and there appears to be a straightforward way to proceed from here to a full up working production-quality microcode development system.

4.1.3 High Speed

In the real world, program speed is certainly an important issue. We were concerned that the microcode compiler would be non-competitive with state-of-the-art microcode development systems because of the added algorithm complexity. Our approach involved using proprietary algorithms and data structures developed here at PENGUIN SOFTWARE, Inc. over the past five years. The result is that the microcompiler runs in times more than competitive with existing meta-assemblers.

4.1.4 Graceful Degradation

All commercially successful meta-assemblers have at least one capability in common: This is the ability to support any machine that may be designed. Using the traditional approach, this is not the case with microcode compilers. It is easy to envision target machine designs that would render a traditional microcode compiler useless. The complexity and sensitivity of the resource allocation schemes and the microcode compaction algorithms leave these compilers extremely vulnerable to total failure. For example, no resource allocation schemes have been even proposed which can handle the extreme encoding of the AMD-29116.

With our approach, a situation involving total failure due to target machine complexity is avoided. In the worst cases, the high-level language supported will fail to meet user expectations with respect to the level of the language. In this case the greatest concern has to do with how successfully low level hardware design features can be incorporated into a high level language. This is a small issue compared to a case where the microcode compiler simply no longer works.

4.2 Recommendations

The following recommendations have been essentially presented in the Phase II proposal. Below we are attempting to give perhaps a little more insight into why these actions were proposed.

4.2.1 Semantic Action Engine

The original concept for the microcode compiler involved thinking of the compiler as simply outputting field values to be inserted into the microprogram object file. This works well for an application language at the level of an assembly language, but fails to attain what we usually think of as a high-level language capability. As more high-level capabilities are included it becomes necessary for the syntax compiler to command different types of semantic actions and equally necessary for the microcompiler to perform them. We now think of the microcode compiler as an engine for performing semantic actions rather than just in terms of generating field values.

This shift in thinking results in some obvious changes in the semantics data file being produced by the syntax compiler, but the changes in the microcompiler are perhaps more profound. With an assembly level language, the first pass of our compiler just needed to determine statement labels and addresses. In order to attain a high-level language capability, we needed to completely compile each statement during both the first and second passes. Since this capability was not exercised in the original design, our current prototype has a substantial amount of kludged software. This software must be redesigned to reflect our new understanding of the problem.

One of the advantages of this software redesign is that the compiler will become extremely flexible with regard to which semantic actions are possible. As new language structures are envisioned, they can be easily incorporated into the new design without a significant loss of reliability and without sacrificing sound software engineering principles.

4.2.2 Intermediate Language

Once we have conceptually established the microcompiler as an engine for performing semantic actions, then it becomes a simple matter for the microcompiler to output strings of text in response to an application language statement. In this case, the text strings will be HALE meta-assembly language source code which will go into an intermediate language output file. This simple approach allows full access to the HILEVEL Technology microcode development system resources at an extremely low cost.

4.2.3 Speed Becomes a Resource

The results of the timing study were somewhat surprising; the microcode compiler is much faster than expected. Speed now becomes a resource which we can use in future design

"trade-offs". One "trade-off", in particular, involves using more complicated algorithms (which will reduce speed) in order to increase the size of the problems that can be handled by the microcompiler. This may allow us to use a more natural approach to cascadeable microprocessors than might otherwise be possible. Another "trade-off" might be made on the operational level where the operator may choose to always run the syntax compiler with the microcompiler, perhaps as a preprocessor. A third possibility is that we may eventually incorporate a local code compaction algorithm in the microcompiler. This would be used to improve the efficiency of the microcode and would allow the use of less skilled application programmers.

APPENDIX A

SYNTAX CHARTS

A1.0 SYNTAX CHARTS

Syntax charts are an easy way to communicate the structure of a computer language. Aside from being mostly self explanatory, they are a valuable programming tool. The software design will consist to a large extent of attaching semantic actions to the various paths in the syntax charts. The coding phase consists of generating structured code to reflect the required semantic actions.

An application program is defined by starting with a root syntax chart and repeatedly applying the other syntax charts as required until all nonterminals are eliminated.

A2.0 SYNTAX CHART SYMBOL DEFINITIONS

1. Circles and ellipses contain terminals. These terminals are sequences of symbols which appear as tokens in the programs. Each program is written entirely in terms of these terminal tokens.
2. Rectangles contain nonterminals. Each nonterminal is identified by a name (in capital letters). This name identifies the syntax chart which will define the nonterminal.
3. Lines and arrows indicate the legitimate paths through the charts. By following all possible paths along the lines in the directions of the arrows, all possible legal statements can be generated.

A3.0 SYNTAX CHART EXPRESSION DEFINITIONS

1. string: A string is a sequence of letters, and digits. A string can also be a single special character. A string used as a terminal usually refers to a token which will be defined by the application programmer.
2. SINGLE QUOTES: Single quotes are used to enclose a particular string which must appear in the program at the specified location.
3. A-string: An A-string is a string composed of letters and digits which starts with a letter.
4. comment: An arbitrary string of characters which are treated as white space.
5. white space: White space consists of blanks, tabs, and comments. These have no semantic meaning other than to mark

the end of a token or string. Although New Lines sometimes have semantic meaning, they can also be used to generate white space.

6. single letter: A special string consisting of one letter.
7. ***_name: A name assigned to some data structure.
8. integer: A decimal integer unless specified otherwise.
9. HEX integer: A hexadecimal integer.
10. NL: A New Line is equivalent to a line feed, carriage return inserted at the end of a line of text.
11. EOF: An End of File mark at the end of a computer file.

A4.0 APPLICATION PROGRAM SYNTAX CHARTS

1. PROGRAM(Fig. A.1): This is the application program root syntax chart. It shows the basic structure of the application program, and all of the pseudo-operations(pseudo-ops). Every program will start with a 'WIDTH' pseudo-op to define the microprogram word width, and every program will end with the 'END' pseudo-op. The other pseudo-op's and program sentences are used as required by the particular application program.

2. SENTENCE(Fig. A-2): The sentence is the application program statement which defines the microword. Each sentence requires a period: this is the only required part of the sentence. From the period to the end of the line defines the comment. A sentence consisting of just a comment, and no clauses will not advance the program counter. In other words, it does not generate a microword of output just a comment in the output listing.

In order to generate a microword, the sentence must include at least one clause. Each sentence can contain TBD clauses separated by semicolons and with the last clause followed by the period. A sentence containing at least one clause can be labeled. This label is assigned the value of the program counter associated with the sentence.

3. LABEL(Fig. A-3): The label is identified by an A-string followed by a colon.

4. CLAUSE(Fig. A-4): The clause is an ordered sequence of strings as defined in the syntax compiler. While the strings within a clause are ordered, the clauses themselves can be in any order.

5. WIDTH(Fig. A-5): The 'WIDTH' of the microword is the number of bits in the microword. This pseudo-op is required and must be the first to appear in any application program.
6. END(Fig. A-6): The 'END' identifies the end of the application program and is required.
7. ORIGIN(Fig. A-7): The 'ORIGIN' pseudo-op is used to set the program counter to some desired location. This allows the programmer to control where the microcode will be located.
8. TITLE(Fig. A-8): The 'TITLE' pseudo-op is used to determine the title to appear at the top of the output listings. The title can be changed at any time and will appear at the top of the succeeding listing pages.
9. SUBTITLE(Fig. A-9): The 'SUBTITLE' pseudo-op is similar to the 'TITLE' in that it generates a subtitle which appears on the second line of each page.
10. RADIX(Fig. A-10): The 'RADIX' defines the operative radix. Any number not having a specified radix will be interpreted with regard to the operative radix. The possible radices are binary('BIN'), octal('OCT'), decimal('DEC'), and hexadecimal('HEX').
11. DEFAULT(Fig. A-11): Bit locations in a microword which are not specified by a clause will be assigned a default value as defined by the 'DEFAULT' pseudo-op. This pseudo-op requires a hexadecimal integer operand.
12. PAGE(Fig. A-12): The 'PAGE' pseudo-op is used to force a page eject.

A5.0 SYNTAX DEFINITION PROGRAM SYNTAX CHARTS

1. SYNTAX-DEF(Fig. A-13): This is the root of the syntax definition program. First, the microword is divided up into fields which usually correspond to various functions and resources to be controlled. The CLAUSE's, LIST's, and LITERAL's are used to define the application language. Finally, the program is terminated when an End-of-File mark is encountered.

During the phase I effort, the syntax compiler merely exists to generate test cases for the microcompiler. As a consequence, the syntax compiler was constructed with the minimum of pseudo-op's and features.

2. FIELDS(Fig. A-14): The bits in the microword are numbered from right to left and from one to n (where n is the length

of the microword). A microword is typically divided up into contiguous bit fields corresponding to the various resources to be controlled. 'FIELDS' starts the field definition and is followed by left and right curly brackets which enclose the field definitions.

3. FIELD-DEF(Fig. A-15): A field is defined by giving it a name and specifying the bits in the field. In the case of a one bit field, it is only necessary to specify the one bit. For fields with more than one bit, then the number of the left bit and the number of the right bit are specified.

4. FIELD-NAME(Fig. A-16): The field name consists of a single letter followed by an integer.

5. LIT-STRING(Fig. A-17): The literal string (or series of strings) is a string enclosed in double quotes. This is used to specify a particular string (or strings) that must always appear in a given position of the clause.

6. CLAUSE(Fig. A-18): CLAUSE is followed by a definition of the tokens in the clause and then possibly by some semantic actions to be performed. There are four ways to define the tokens appearing in the clause. They are by specifying:

- a) a literal string(LIT-STRING).
- b) a literal(LIT-NAME).
- c) a list name(list-name).
- d) a dimensioned list column(DLIST_COL).

7. LIT-NAME(Fig. A-19): A literal name consists of the word LITERAL followed by an integer enclosed in parentheses.

8. DLIST-COL(Fig. A-20): A particular column in a dimensioned list is identified by the list name followed by parentheses containing the integer associated with the required column.

9. C-FIELD-VAL(Fig. A-21): C-FIELD-VAL is similar to L-FIELD-VAL with the addition of the LBL field command and five commands to the microcompiler. The LBL command tells the microcompiler to get the address associated with the internally generated label for this instruction. This address is then truncated or right justified and zero filled on the left as required for the specified field. The literal thus obtained is rotated if necessary as specified by the integer enclosed in the parentheses following the LBL command.

The microcompiler commands used to generate the internally

generated labels and addresses are as follows:

a) 'PPA' causes an address to be popped off the stack, and causes a label to be generated. This address is then associated with this new label.

b) 'PPL' causes a label to be popped off the stack, and associated with the current microaddress.

c) 'PHA' causes the current microaddress to be pushed onto the stack.

d) 'PHL' causes a label to be generated and pushed onto the stack.

e) 'SWP' causes the top two entries in the stack to be swapped.

10. LIST(Fig. A-22): The LIST pseudo-op is used to define simple lists and dimensioned lists. Both lists consist of the name of the list and a list definition. The dimensioned list names are of the form defined in DLIST.

11. DLIST(Fig. A-23): The dimensioned list name is followed by parentheses. These enclose integers associated with the multiple columns(possibly one) in the dimensioned list.

12. LIST-DEF(Fig. A-24): A simple list definition is enclosed in curly brackets. Each token in the list is followed by the associated field values. The token and field value parts are separated by semicolons.

13. L-FIELD-VAL(Fig. A-25): The field value definitions are enclosed in square brackets. A field name is separated from the HEX value to be inserted in that field by a back slash. If more than one field value is to be inserted, then the field name\value pairs are separated by semicolons.

14. DLIST-DEF(Fig. A-26): The dimensioned list consists of columns of tokens (separated by commas) followed by the associated field values. Multiple rows are separated by semicolons.

15. LITERAL(Fig. A-27): A literal definition consists of a literal name and a formula for determining the fields and values to be inserted.

16. LIT-VAL(Fig. A-28): The formula for determining the fields and values is enclosed in square brackets. A field name is followed by a back slash and then by one of three possible ways to define the value to be inserted into the

specified field. These three ways are:

a) As in L-FIELD-VAL, a hex integer to be inserted into the specified field.

b) A literal number (a constant or a value associated with a label) is right justified and zero filled or truncated on the left as required.

c) First a literal number is conditioned as above in b) and then it is rotated by n bits to the right where n is the integer enclosed in parentheses following ROR.

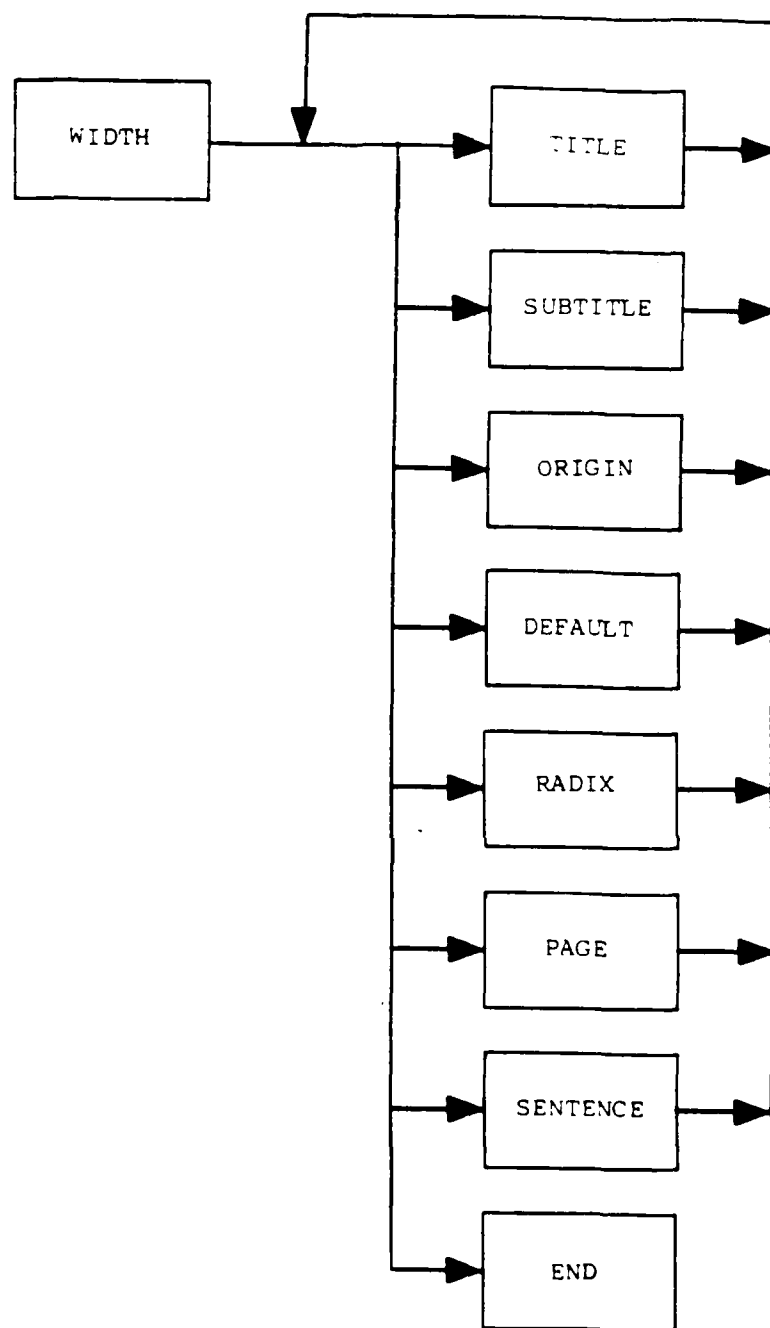


Figure A-1. PROGRAM.

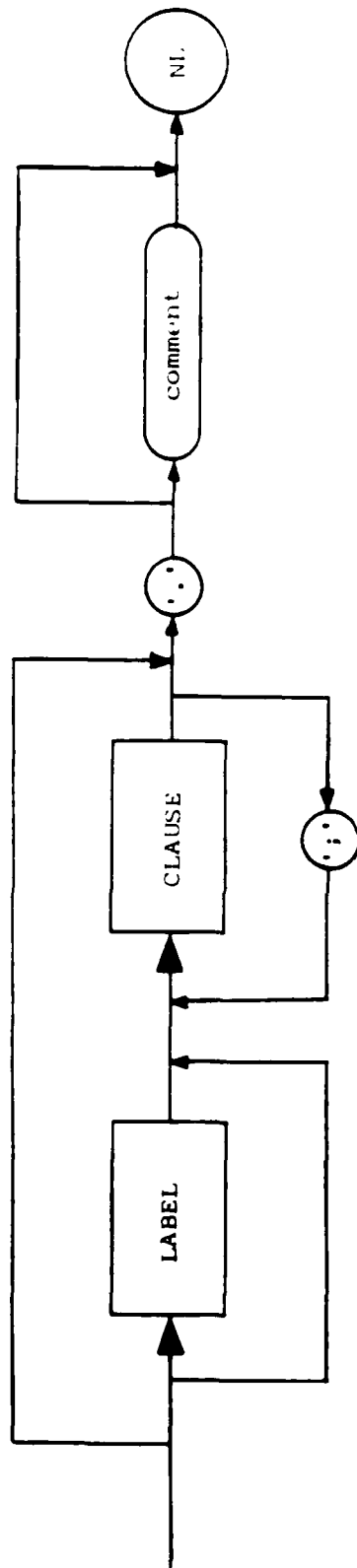


Figure A-2. SENTENCE.

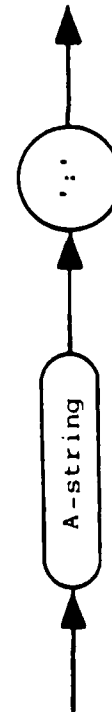


Figure A-3. LABEL.

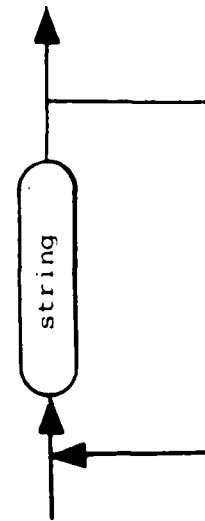


Figure A-4. CLAUSE.

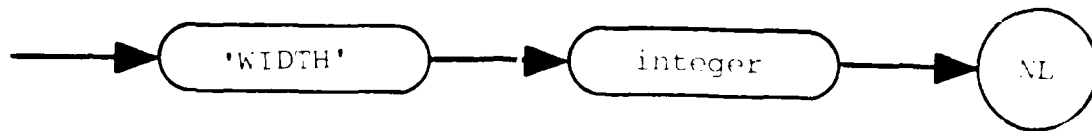


Figure A-5. WIDTH.

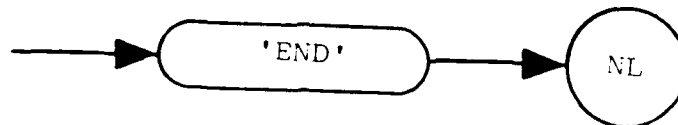


Figure A-6. END.

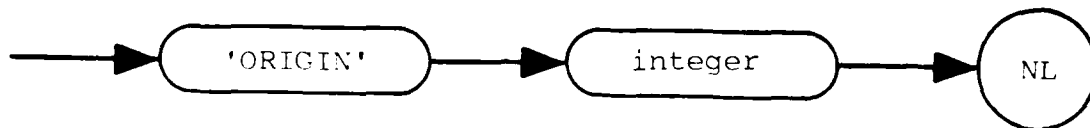


Figure A-7. ORIGIN.

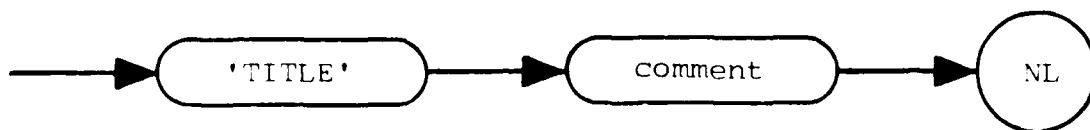


Figure A-8. TITLE.

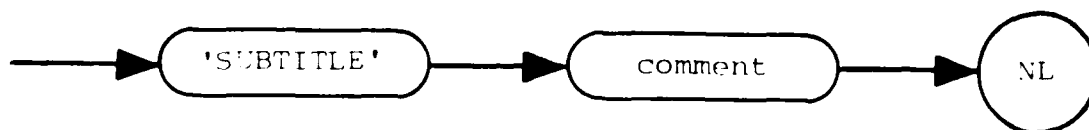


Figure A-9. SUBTITLE.

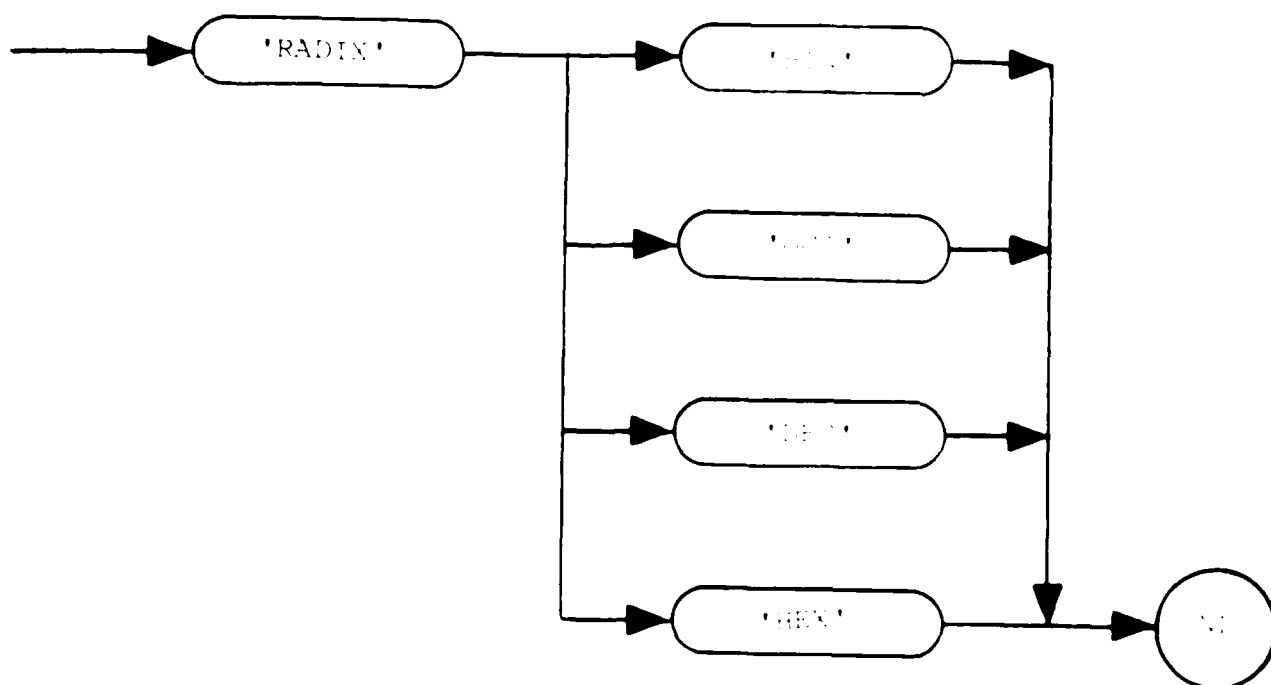


Figure A-10. RADIX.

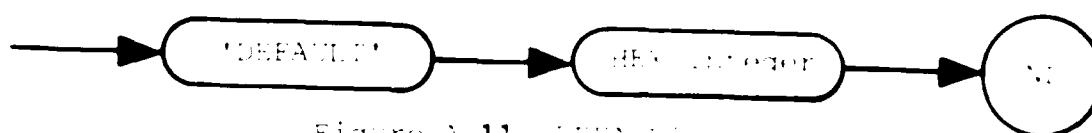


Figure A-11. DEFAULT.

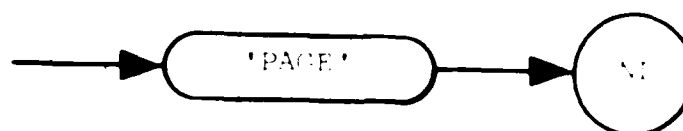


Figure A-12. PAGE.

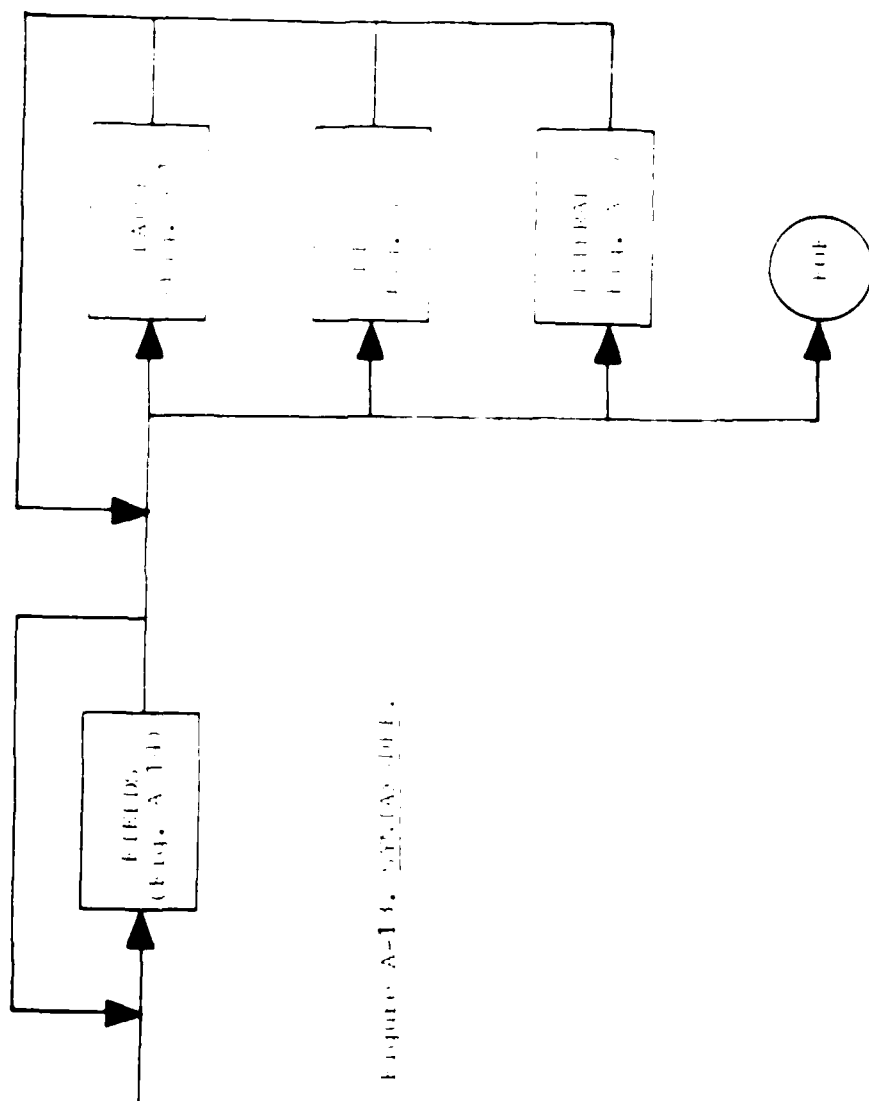
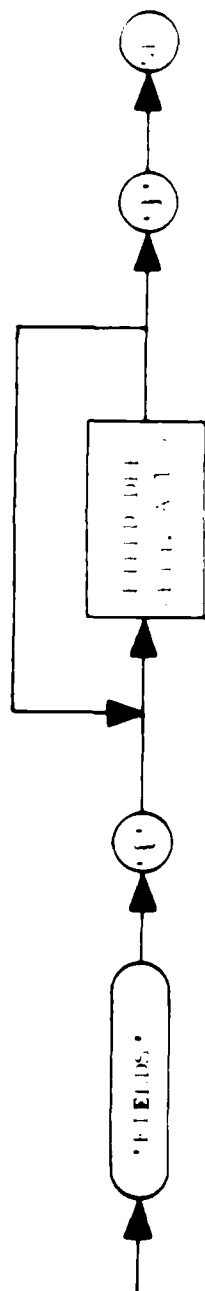
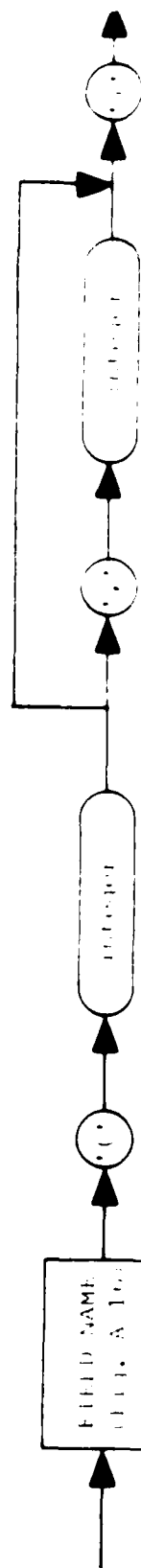


Figure A-14. $\frac{v(t)}{v(t)}$ (A-14)



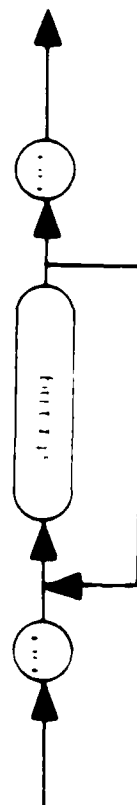
100



100



100



— — — — —

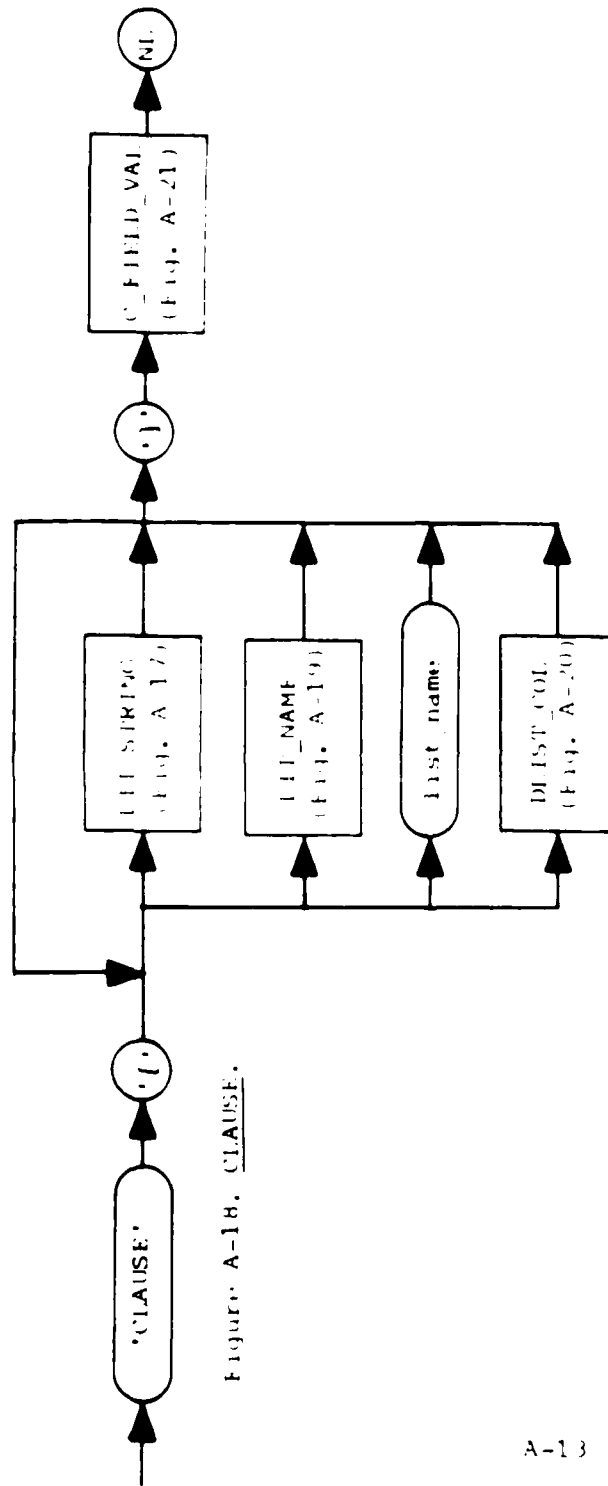


FIGURE A-18. CLAUSE.

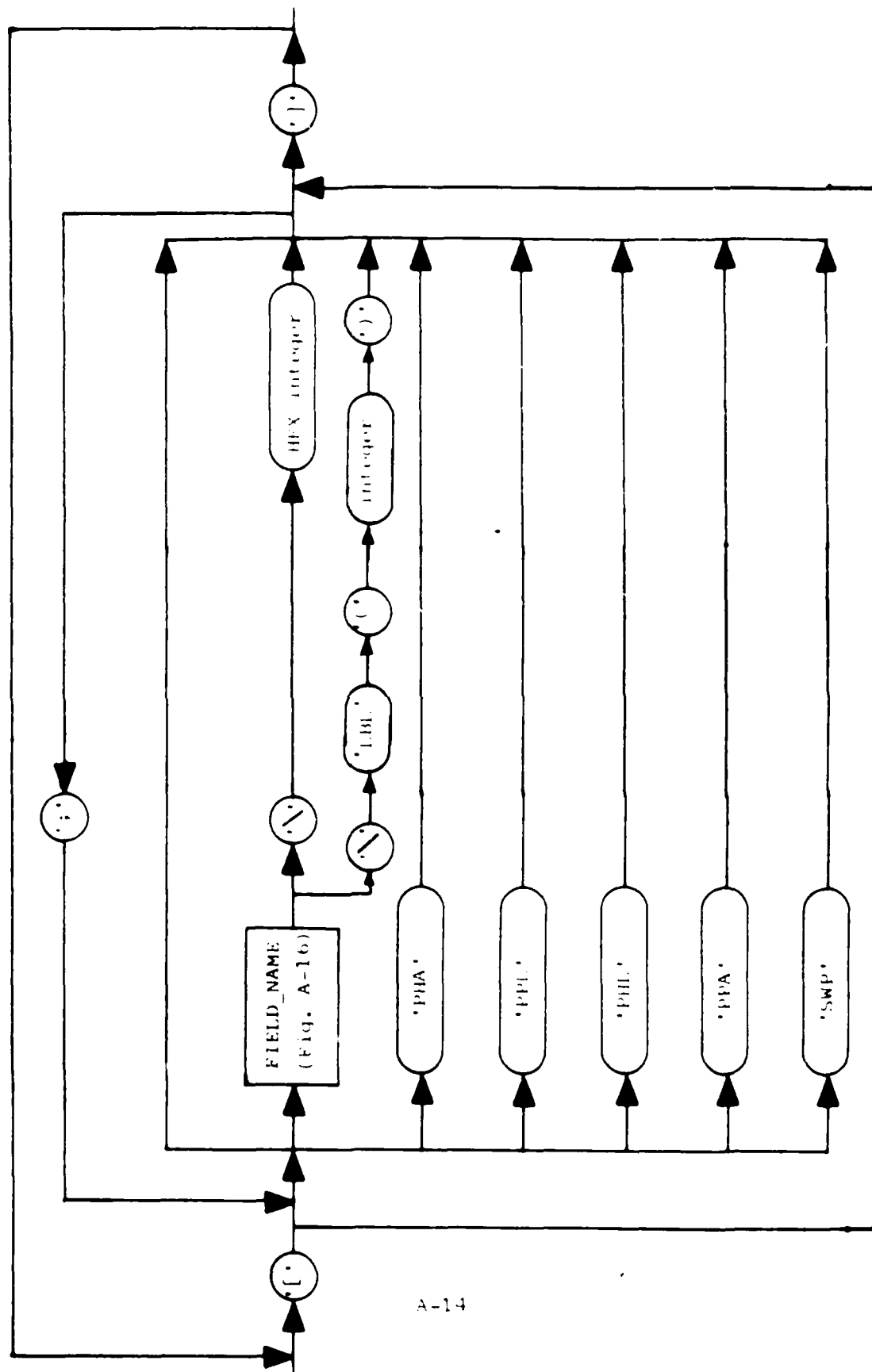


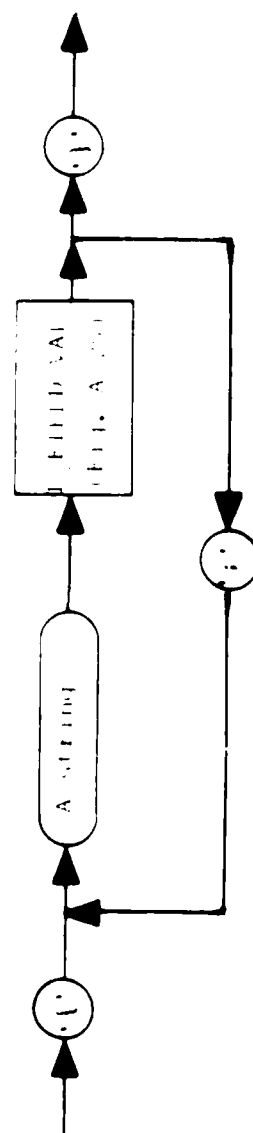
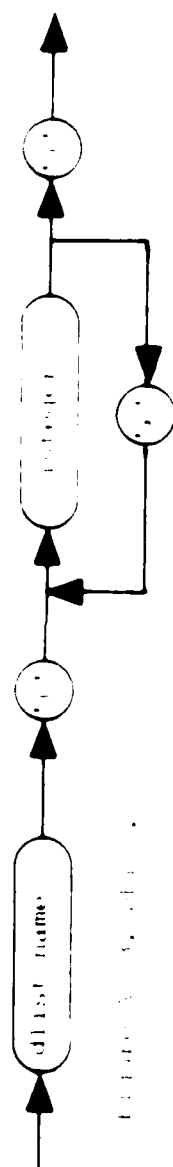
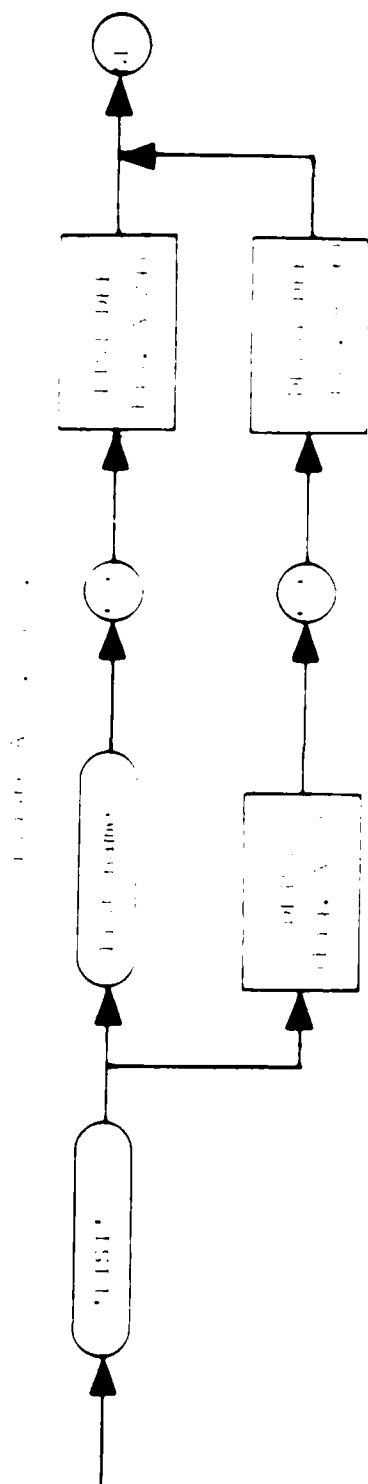
Figure A-19. LIT-NAME.



Figure A-20. DLIST-COL.

Figure A-21. C-FLHED-VAF.





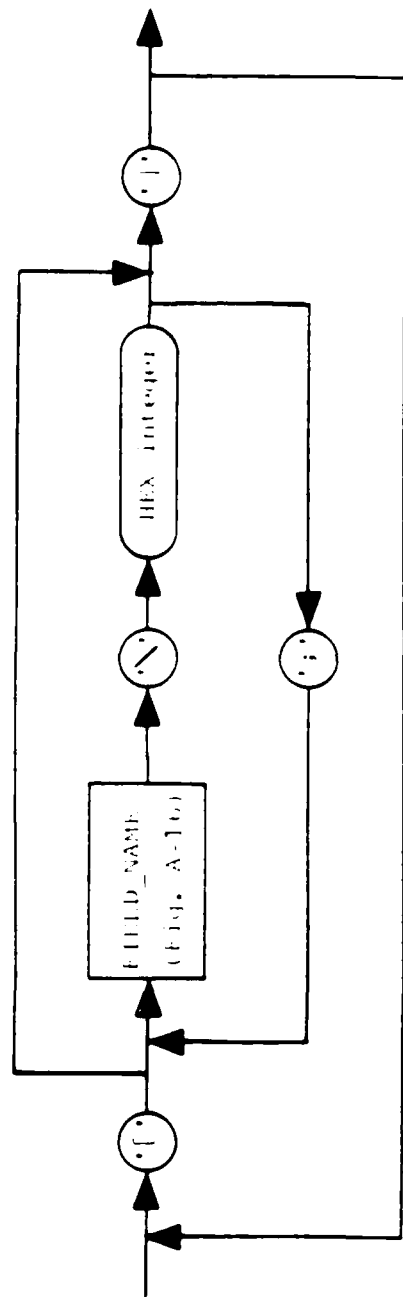


Figure A-25. L_FIELD_VAL.

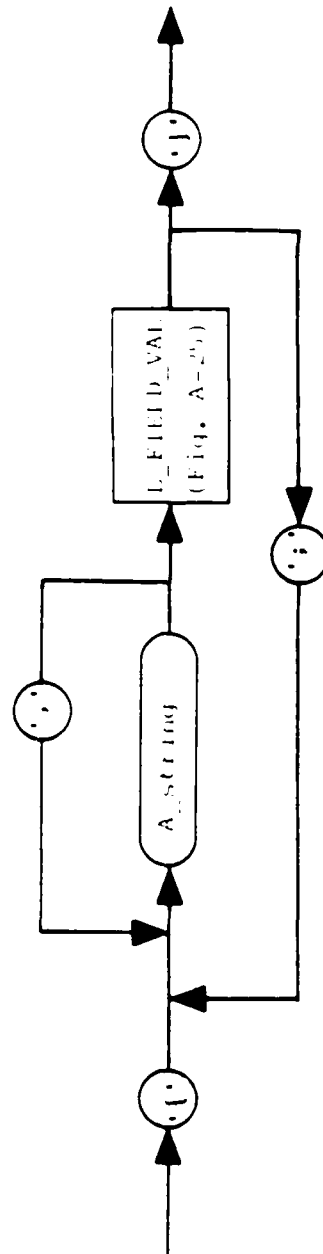


Figure A-26. DLIST-DEF.

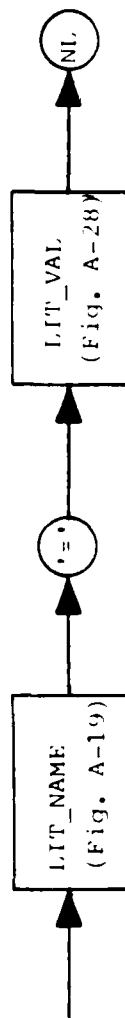


Figure A-27. LITERAL.

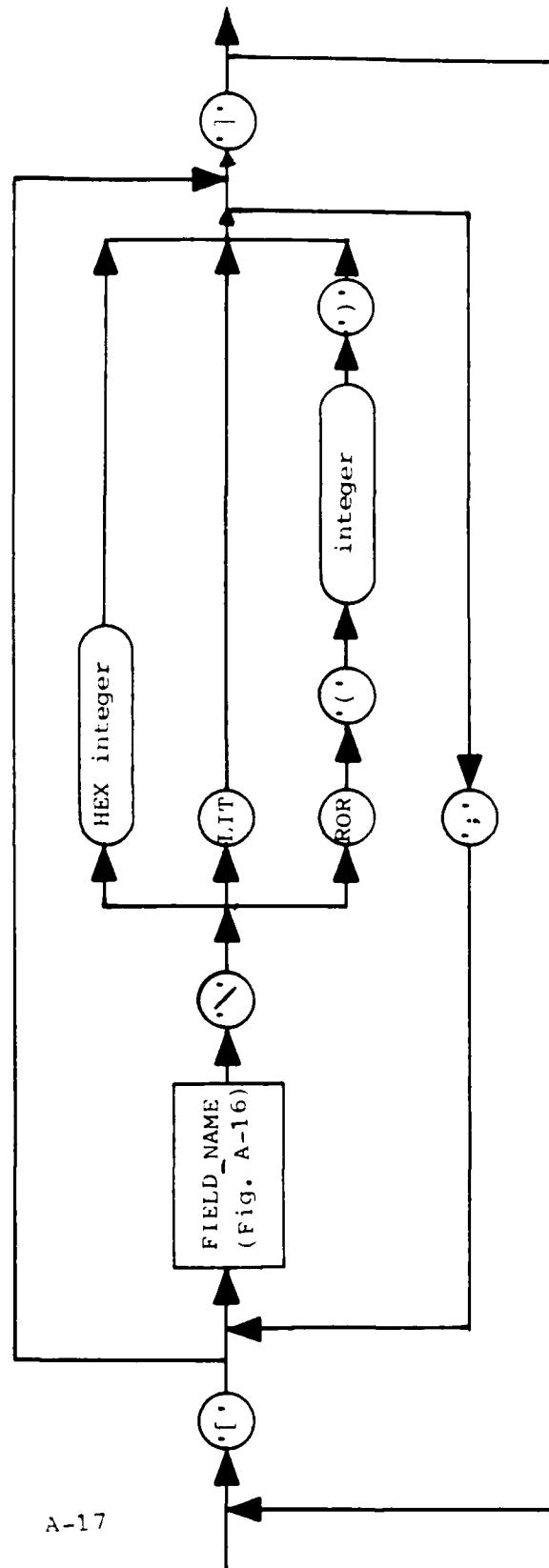


Figure A-28. 11T-VAL.

APPENDIX B

SAMPLE TEST CASE

```

.      FILENAME      AMDSRC.MIN
.
.  THIS IS A TEST PROGRAM FOR THE MICRO COMPILER
.  BASED ON THE AMD PROGRAM
.
.      WIDTH      32
.      TITLE      AMD ONE'S COUNTING PROGAM
.      STITLE      SEE PAGE 4-24 AM2900 FAMILY DATA BOOK
.      DEFAULT 02100000
.      ORIGIN      0
.
.  INITIALIZE DATA REGISTERS
.
.      R0 = 15.
.      R1 = 9.
.      R2 = 0.
.
.  INITIALIZE BIT COUNTER AND TOTAL
.
.      CNTR = 4.
.      TOTAL = 0.
.
.  COUNT ONES
.
.      REPEAT.
.          IF(R0(SRA) IS ODD) CALL UPTOTAL.
.          IF(R1(SRA) IS ODD) CALL UPTOTAL.
.          IF(R2(SRA) IS ODD) CALL UPTOTAL.
.      UNTIL(DEC(CNTR) = ZERO).
.
.  LOOP WHILE OUTPUTTING TOTAL
.
.      REPEAT.
.          OUTPUT TOTAL;
.      UNTIL(FOREVER).
.
.  ROUTINE INCREMENTS ONES COUNTER
.
UPTOTAL:  ROUTINE;
.          INC(TOTAL);
.      RETURN.
.      END

```

```

.      filename          AMDTEST.SRC
.      This is a test file for the syntacts generator

.      Set the word width to 32 bits
WIDTH  32
.      Define the fields of the microword

FIELDS {
        X1(32,29)          . Branch address
        X2(28,25)          . Next instruction
        X3(24)             . Mux 1
        X4(23,21)          . Destination Control
        X5(20)             . Mux 0
        X6(19,17)          . Source selection
        X7(16)             . Cn
        X8(15,13)          . ALU
        X9(12,9)           . A Source
        X10(8,5)           . B Source
        X11(4,1)           . D Source
    }
LIST IFCON(1,2) = {
        IFNOT,ZERO[X2\0];
        IFNOT,OVER[X2\E];
        IF,ZERO[X2\C];
        IF,F3[X2\D];
        IF,C4[X2\F]
    }
LIST MCS1 = {
        CONT[X2\2];
        RETURN[X2\6];
        PUSH[X2\9];
        POP[X2\A]
    }
LITERAL(1) = [X1\LIT]
LIST MCS2 = {
        ZERO[X2\8];
        C4[X2\B]
    }
LIST ABREGS(1) = {
        R0[X10\0;X9\0];
        R1[X10\1;X9\1];
        R2[X10\2;X9\2];
        TOTAL[X10\3;X9\3];
        CNTR[X10\4;X9\4];
        R5[X10\5;X9\5];
        R6[X10\6;X9\6];
        R7[X10\7;X9\7];
        R8[X10\8;X9\8];
        R9[X10\9;X9\9];
        R10[X10\A;X9\A];
        R11[X10\B;X9\B];
    }

```

```

R12[X10\C;X9\C];
R13[X10\D;X9\D];
R14[X10\E;X9\E];
R15[X10\F;X9\F]
}
LIST BREGS(1) = {
R0[X10\0];
R1[X10\1];
R2[X10\2];
TOTAL[X10\3];
CNTR[X10\4];
R5[X10\5];
R6[X10\6];
R7[X10\7];
R8[X10\8];
R9[X10\9];
R10[X10\A];
R11[X10\B];
R12[X10\C];
R13[X10\D];
R14[X10\E];
R15[X10\F]
}
LIST CREGS = {
R0[X10\0;X9\0][X10\0];
R1[X10\1;X9\1][X10\1];
R2[X10\2;X9\2][X10\2];
TOTAL[X10\3;X9\3][X10\3];
CNTR[X10\4;X9\4][X10\4];
R5[X10\5;X9\5][X10\5];
R6[X10\6;X9\6][X10\6];
R7[X10\7;X9\7][X10\7];
R8[X10\8;X9\8][X10\8];
R9[X10\9;X9\9][X10\9];
R10[X10\A;X9\A][X10\A];
R11[X10\B;X9\B][X10\B];
R12[X10\C;X9\C][X10\C];
R13[X10\D;X9\D][X10\D];
R14[X10\E;X9\E][X10\E];
R15[X10\F;X9\F][X10\F]
}
CLAUSE {BREGS(1) "=" LITERAL(2)}[X4\3;X6\7;X8\3]
LITERAL(2) = [X11\LIT]
LITERAL(3) = [][X1\LIT]
CLAUSE {ABREGS(1) "AND" LITERAL(2)}[X6\5;X8\4]
CLAUSE {BREGS(1) "OR" LITERAL(2)}[X6\3;X8\3]
CLAUSE {BREGS(1) "=" BREGS(1) "+" "ONE"}[X4\3;X6\3;X7\1;X8\0]
CLAUSE {"INC(" BREGS(1) ")"}[X4\3;X6\3;X7\1;X8\0]
CLAUSE {IFCON(1) IFCON(2) "GOTO" LITERAL(1)}
CLAUSE { MCS_ }
CLAUSE {"GOSUB" LITERAL(1)}[X2\5]

```

```

CLAUSE {"CALL" LITERAL(1)}[X2\5]
CLAUSE {"GOTO" LITERAL(1)}[X2\1]
CLAUSE {"GOTO" "SWITCH"}[X2\3]
CLAUSE {"GOTO" "FILE"}[X2\7]
CLAUSE {"IFNOT" "ZERO" "GOSUB" LITERAL(1)}[X2\4]
CLAUSE {"IF" MCS2 "END" "LOOP" "AND" "POP"}
CLAUSE {BREGS(1) "=" BREGS(1) "-" "ONE"}[X4\3;X6\3;X7\0;X8\1]
CLAUSE {"DEC(" BREGS(1) ")"}[X4\3;X6\3;X7\0;X8\1]
CLAUSE {BREGS(1) "=" "SRA" BREGS(1)}[X4\5;X6\3;X8\3]
CLAUSE {"SRA(" BREGS(1) ")"}[X4\5;X6\3;X8\3]
CLAUSE {"REPEAT"}[PHA]
CLAUSE {"UNTIL(FOREVER)"}[PPA;X2\1;X1\LBL]
CLAUSE {"ROUTINE"}
CLAUSE {"OUTPUT" BREGS(1)}[X6\3;X8\3;X11\0]
CLAUSE {"UNTIL(DEC("BREGS(1)")=ZERO)"}
      [X4\3;X6\3;X7\0;X8\1][PPA;X2\0;X1\LBL]
CLAUSE {"IF(" CREGS "(SRA) IS ODD)GOSUB" LITERAL(3)}
      [X6\5;X8\4;X11\1][X2\4;X4\5;X6\3;X8\3]
CLAUSE {"IF(" CREGS "(SRA) IS ODD)CALL" LITERAL(3)}
      [X6\5;X8\4;X11\1][X2\4;X4\5;X6\3;X8\3]

```

APPENDIX C

DESIGN REQUIREMENTS

C1.0 Design requirements

In addition to the syntax charts, certain other design requirements were identified. The following requirements were derived based on the initial interviews and based on the designs of the compilers that were to be modified.

The reserved characters naturally fell into three categories: reserved for the microcompiler; reserved for the syntax compiler; and not reserved. The microcompiler has the following three reserved characters: the period ".", the colon ":", and the semicolon ";". The syntax compiler has the following five reserved characters: "[", "]", "\", "{", and "}". The remaining characters are not reserved and are available to be included in the definition of the application language.

As it turned out, most parameters were used by both compilers. These parameters were defined to be greater than twice the size of the largest test case.

Parameter Name	Parameter Value	Parameter Definition
MAXCOL	4	max cols in list
MAXLEN	20	max length of input string
MAXCLS	200	max clause definitions
MAXLIT	8	max literal definitions
MAXLST	90	max list definitions
MAXFLD	50	max fields
MAXSTR	400	max strings
MAXLNG	9	max token length + 1
MXTPC	10	max tokens per clause
MXCPS	10	max clauses per sentence
MAXLAB	1000	max labels
MAXWID	132	max microword width
AFPL	3	average fields per list
ARPL	8	average rows per list
ACPL	2	average columns per list
AFPT	2	average fields per literal
ATPC	5	average tokens per clause
AFPC	2	average fields per clause
STSHSIZ	128	string hash table size
HASHSIZE	32	label hash table size

APPENDIX D

PDL LISTINGS

FILENAME: SYNMAN.PDL

This is the main driver for the syntacts generator.

```
main()
{
    initialize for phase 1
    synpar()
    initialize for phase 2
    synpar()
    invert syntax tables
    output data and listing
}
```

FILENAME: SYNPAR.PDL

This the parser for the syntacts generator.

```
synpar()
{
    TOKEN = NEXT_TOKEN
    DO {
        SWITCH(TOKEN){
            CASE FIELD:      sfield()
                             BREAK
            CASE CLAUSE:     sclaus()
                             BREAK
            CASE LIST:       slist()
                             BREAK
            CASE LITERAL:    slit()
                             BREAK
            CASE EOF:        return
            CASE PAGE:       eject page
                             BREAK
            default:         report error
                             BREAK
        }
    } WHILE(FOREVER)
}
```

FILENAME: SFIELD.PDL

This function handles all field definitions.

```
sfield()
{
    token = next_token
    if(token ne '('){
        report error
        return
    }
    token = next_token
    if(token is not a letter){
        report error
        return
    }
    field_letter = token
    do {
        if(phase = 1) {
            if(too many fields) {
                report error
                return
            }
        }
        token = next_token
        if(token is not a number) {
            report error
            return
        }
        if(phase = 1) {
            field_number = token
            if(field already defined) {
                report error
                return
            }
        }
        token = next_token
        if(token ne '(') {
            report error
            return
        }
        token = next_token
        if(token is not a number) {
            report error
            return
        }
        if(phase = 1) field_start=token
        token = next_token
        if(token ne ',') {
            if(token ne ')') {
                report error
            }
        }
    }
```

```

        return
    }
    field_stop = field_start
}
else {
    token = next_token
    if(token is not a number) {
        report error
        return
    }
    if(phase = 1) field_stop=token
    token = next_token
}
if(token ne '))') {
    report error
    return
}
if(phase = 1) {
    if(field too big) {
        report error
        return
    }
    if(field reversed) {
        report error
        return
    }
    field_let[fortot] = field_letter
    field_num[fortot] = field_number
    field_strt[fortot] = field_start
    field_stp[fortot++] = field_stop
}
token = next_token
if(token is a letter) field_letter = token
} while(token is a letter)
if(token ne '))') {
    report error
    return
}
token = next_token
return
}

```

FILENAME: SCLAUS.PDL

This function handles all clause definitions.

```
clause()
{
    errstr=FALSE
    IF(phase = 2) {
        IF(too many clauses) {
            report error
            return
        }
        initialize clspnt(temporary clause definition pointer)
        initialize cfpntr(clause fields definition pointer)
    }
    token = next token
    IF(token ne '[') {
        report error
        return
    }
    token = next token
    IF(token = ']') {
        report error
        return
    }
    DO {
        SWITCH(toktype) {
            CASE LITERAL: {
                token = next token
                IF(token ne '[') {
                    report error
                    return
                }
                token = next token
                IF(token ne number) {
                    report error
                    return
                }
            }
            IF(phase = 2) {
                IF(literal undefined) {
                    report error
                    return
                }
                clspnt = clspnt + 1
                cls_def[clspnt]=-(literal number)-1
                clspnt = clspnt + 1
                cls_def[clspnt++]=--1
            }
            token = next token
            IF(token ne '[') {
                report error
            }
        }
    }
```

```

        return
    }
    token = next_token
    SWITCH(token) {
    CASE LITERAL:          BREAK
    CASE STRING:           BREAK
    CASE QOUTES:           BREAK
    CASE '}'':            BREAK
    DEFAULT:               report error
                           return
    }
    BREAK
}
CASE STRING: {
    IF(phase = 2) {
        tmpstr = token
        decnum=0
    }
    token = next_token
    SWITCH(token) {
    CASE '(': {
        token = next_token
        IF(token ne number) {
            report error
            return
        }
        ELSE {
            IF(phase = 2) {
                decnum = token
                IF(decnum<=0) {
                    report error
                    return
                }
            }
            token = next_token
            IF(token ne '^') {
                report error
                return
            }
        }
        token = next_token
        BREAK
    }
    CASE LITERAL: BREAK
    CASE STRING:  BREAK
    CASE QOUTES:  BREAK
    CASE '}'':    BREAK
    DEFAULT:      report error
                  return
    }

    IF(phase = 2) {

```

```

        IF(list undefined) {
            report error
            return
        }
        clspnt = clspnt + 1
        cls_def[clspnt]=list number-MAXLST-MAXLIT-1
        clspnt = clspnt + 1
        cls_def[clspnt]=decnum
    }
    BREAK
}
CASE QOUTES:    {
    token = next_token
    IF(token==QOTTK) {
        report error
        return
    }
    while(toktype ne '"') {
        IF(phase = 2) {
            clspnt = clspnt + 1
            cls_def[clspnt]=string number
            clspnt = clspnt + 1
            cls_def[clspnt]=-1
        }
        token = next_token
    }
    token = next_token
    SWITCH(token) {
        CASE LITERAL: BREAK
        CASE STRING:  BREAK
        CASE QOUTES:  BREAK
        CASE '}'':    BREAK
        DEFAULT:      report error
                      return
    }
    BREAK
}
DEFAULT: {
    report error
    return
}
}
} WHILE(toktype ne '}')
token = next_token
SWITCH(token) {
CASE PAGE:      BREAK
CASE EOF:       BREAK
CASE FIELDS:    BREAK
CASE CLAUSE:    BREAK
CASE LIST:      BREAK
CASE LITERAL:   BREAK

```

```

CASE '[': {
    IF(phase = 2) cfield(TRUE)
    ELSE          cfield(FALSE)
    BREAK
}
DEFAULT: {
    report error
    return
}
}
IF(phase = 2) {
    update cdpntr = clspnt-1
    update clstot = clstot + 1
}
return
}

```

FILENAME: SLIST.PDL

This function handles all list definitions.

```
slist()
{
    IF(phaseno==1) {
        IF(too many lists) {
            report error
            return
        }
    }
    token = next_token
    IF(token is not a string) {
        report error
        return
    }
    IF(phase = 1) {
        IF(list already defined) {
            report error
            return
        }
        define list
        initialize lpnt(temporary pointer into lstaray)
        initialize lstdp(pointer into list_def)
        initialize lstfdp(pointer into lstfpp)
        initialize itemp(temporary list dimension counter)
    }
    token = next_token
    IF(token = '(') {
        do {
            token = next_token
            IF(token is not a number) {
                report error
                return
            }
            IF(phase = 1) {
                itemp = itemp + 1
                lstaray[lpnt] = lstaray[lpnt] + 1
                lstaray[lpnt+itemp] = token
            }
            token = next_token
            while(token = ',')
            IF(toktype ne ')') {
                report error
                return
            }
            token = next_token
        }
    }
    IF(phase = 1) {
        IF((listdim=lstaray[lpnt]) = 0) listdim = listdim + 1
    }
}
```

```

        lstpnt[lstnum]=lpnt+itemp
    }
    IF(toktype ne '=') {
        report error
        return
    }
    token = next_token
    IF(token ne '[') {
        report error
        return
    }
    do {
        colcnt=0
        do {
            token = next_token
            IF(token is not a string) {
                report error
                return
            }
            IF(phase = 1) {
                IF(colcnt<MAXCOL) {
                    lstdtmp[colcnt] = string number of token
                    colcnt = colcnt + 1
                }
                ELSE {
                    report error
                    return
                }
            }
            token = next_token
        } while(token = '[')
        IF(toktype ne '[') {
            report error
            return
        }
        IF(phase = 1) {
            IF(colcnt ne listdim) {
                report error
                return
            }
            lstdp[lstnum]=lstdp[lstnum]+listdim
            move data to list_def from lstdtmp
            initialize lstfpp(pointer into lswfdp)
            lfield(TRUE)
        }
        ELSE {
            lfield(FALSE)
        }
    } while(toktype = '[')
    IF(toktype ne '}') {
        report error
    }

```

```
        return  
    }  
    token = next_token  
    return  
}
```

FILENAME: SLIT.PDL

This function handles the definition of all literals.

```
slit()
{
    token = next_token
    IF(token ne '(') {
        report error
        return
    }
    token = next_token
    IF(token is not a number) {
        report error
        return
    }
    IF(phase = 1) numlit = token
    token = next_token
    IF(token ne ')') {
        report error
        return
    }
    IF(phase = 1) {
        IF(literal is already defined) {
            report error
            return
        }
        IF(too many literals) {
            report error
            return
        }
        define literal
    }
    token = next_token
    IF(token ne '=') {
        report error
        return
    }
    token = next_token
    IF(token ne '[') {
        report error
        return
    }
    initialize ldpntr[littot]
    initialize ltdftot = ltwpntr[ldpntr[littot]]
    do {
        do {
            token = next_token
            IF(token's not a letter) {
                IF(token ne ']') {
                    report error
                }
            }
        }
    }
```

```

        return
    }
}
ELSE {
    token = next_token
    IF(token is not a number) {
        report error
        return
    }
    IF(phase = 1) {
        IF(field is undefined) {
            report error
            return
        }
    }
    token = next_token
    IF(token ne '(') {
        report error
        return
    }
    token = next_token
    IF(token is a HEX number) {
        IF(phase = 1) {
            ltdftot++
            lit_def[ltdftot++] = field number
            lit_def[ltdftot] = hex number
        }
        token = next_token
    }
    ELSE {
        IF(token = LTTTK) {
            IF(phase = 1) {
                ltdftot++
                lit_def[ltdftot++] = -numfor-1
                lit_def[ltdftot] = 0
            }
            token = next_token
        }
        ELSE {
            IF(token = ROR) {
                token = next_token
                IF(token ne '(') {
                    report error
                    return
                }
                token = next_token
                IF(token ne number) {
                    report error
                    return
                }
                IF(phase = 1) litnum = atoi(token)
            }
        }
    }
}

```

```

        token = next_token;
        IF(token ne '[') {
            report error;
            return;
        }
        IF(phase = 1) {
            lttot++;
            lit_def[ltttot++] = -number - 1;
            lit_def[1+ltttot] = 1+ltttot;
        }
        token = next_token;
    }
    ELSE {
        report error;
        return;
    }
}

} while(token = ';')
IF(token ne ']') {
    report error;
    return;
}
IF(phase = 1) {
    update ldpntr[ltttot];
    update ltwpntr[ldpntr[ltttot]] = 1+ltttot;
}
token = next_token;
} while(token = '[')
IF(phase = 1) lttot = lttot + 1;
return;
}

```

FILENAME: CFIELD.FIL

This function handles all clause field definitions.

cfldf(storeff)

```
{
  IF(storeff) {
    initialize fct=cwptr[cfptr, 1st * 1]
  }
  DO {
    10 {
      token = next token
      SWITCH(token) {
        CASE PHA:  ifrm = -1-MAXFIL
                   inum = 0
                   token = next token
                   BREAK
        CASE PPI:  iform = -2-MAXFIL
                   inum = 0
                   token = next token
                   BREAK
        CASE PHI:  iform = -3-MAXFIL
                   inum = 0
                   token = next token
                   BREAK
        CASE PPA:  iform = -4-MAXFIL
                   inum = 0
                   token = next token
                   BREAK
        CASE SWF:  iform = -5-MAXFIL
                   inum = 0
                   token = next token
                   BREAK
        CASE LET:  {
                     flet = token
                     token = next token
                     IF(token is not a number) {
                       report error
                       return
                     }
                     fnum = token
                     IF(field is not defined) {
                       report error
                       return
                     }
                     token = next token
                     IF(token ne ' '){
                       report error
                       return
                     }
                     token = next token

```

```

        IF(token = a HEX number){
            inum = token
            token = next_token
        }
        ELSE {
            IF(token is not a label) {
                report error
                return
            }
            ELSE {
                iform = -iform-1
                inum = 0
                token = next_token
                IF(token = '[') {
                    token = next_token
                    IF(token ne number) {
                        report error
                        return
                    }
                    inum = token
                    token = next_token
                    IF(token ne ']') {
                        report error
                        return
                    }
                    token = next_token
                }
            }
        }
        BREAK
    }
    CASE '[': {
        storef = FALSE
        BREAK
    }
    DEFAULT: {
        report error
        return
    }
    IF(storef) {
        ftot = ftot + 1
        cls_fld_def[ftot] = iform
        ftot = ftot + 1
        cls_fld_def[ftot] = inum
    }
} WHILE(token = ';')
IF(token ne ']') {
    report error
    return
}
cfpntr[clstot] = cfpntr[clstot] + 1
cwpntr[cfpntr[clstot]] = ftot

```

```
        token = next_token  
    } WHILE(token = LSBTK)  
    return  
}
```

FILENAME: IFIELD.PPL

This function handles all list field definitions.

```
lfield(storef)
{
    IF(storef) {
        itemp = lstfpp[lstfdp[lsttot-1]]
        initialize lswfdp[itemp+1] = lswfdp[itemp]
        initialize ftot = lswfdp[itemp+1]
    }
    DO {
        DO {
            token = next_token
            IF(token is not a letter) {
                IF(token ne ']') {
                    report error
                    return
                }
            }
            ELSE {
                token = next_token
                IF(token is not a number) {
                    report error
                    return
                }
                IF(storef) {
                    IF(field is undefined) {
                        report error
                        return
                    }
                }
                token = next_token
                IF(token ne '\'){
                    report error
                    return
                }
                token = next_token
                IF(token is not a HEX number){
                    report error
                    return
                }
                IF(storef) {
                    ftot = ftot + 1
                    lst fld def[ftot] = 1
                    ftot = ftot + 1
                    lst fld def[ftot] = token
                }
                token = next_token
            }
        } WHILE(token = ' ')
    }
}
```

```

        IF(token ne ']') {
            report error
            return
        }
        IF(storef) {
            i = ++lstfpp[lstfdp[lsttot-1]]
            lswfdp[i] = ftot
        }
        token = next token
    } WHILE(token = '[')
    return
}

```

FILENAME: MAIN.PDL

This is the main driver for the microcode compiler.

```
main()
{
    lodtab()
    initialize for phase 1
    parse()
    initialize for phase 2
    parse()
    exit()
}
```

FILENAME: LODTAB.PDL

This function loads the syntax tables.

```
lodtab()
{
    cdpntr = clause definition pointer
    cdef = clause definition array
    lsttot = total number of lists
    lstpnt = lstaray pointer
    lstaray = (#dims,dim#,dim#,...,dim#)
    lstdp = ldefp pointer
    ldefp = array of strings in lists
    cfpntr = cwpntr pointer
    cwpntr = cfld pointer
    cfld = array of fields for clauses
    littot = total number of literal defs
    litdp = ltwpntr pointer
    ltwpntr = litdef pointer
    litdef = array of fields for literals
    lstfdp = lstffp pointer
    lstffp = lfld pointer
    lfld = array of fields by list row
    fortot = total number of formats
    format definitions
    string symbol tables
    l2cp = list to clause pointer
    l2c = list to clause array
    s2cp = string to clause pointer
    s2c = string to clause array
    s2lp = string to list pointer
    s2l = string to list array
    clstot = total number of clauses
    return
}
```

FILENAME: PARSE.PDL

This the parser for the microcode compiler.

```
parse()
{
    token = next_token
    DO {
        SWITCH(token) {
            CASE STRING:    psent()
                           BREAK
            CASE NUMBER:    psent()
                           BREAK
            CASE NEWLINE:   BREAK
            CASE EOF:       report error
                           return
            CASE LABEL:     insert token into label symbol table
                           BREAK
            CASE END:       return
            CASE TITLE:     format title
                           BREAK
            CASE SUBTITLE:  format subtitle
                           BREAK
            CASE RADIX:     store default radix
                           BREAK
            CASE ORIGIN:    set new maddr(micro address)
                           BREAK
            CASE DEFAULT:   store new dault microword
                           BREAK
            CASE WIDTH:     process word width
                           BREAK
            CASE PAGE:      eject page
                           BREAK
            CASE PERIOD:    skip rest of line
                           BREAK
            default:        report error
                           BREAK
        }
    } WHILE(FOREVER)
}
```

FILENAME: PSENT.PDL

This function processes the sentences

```
psent()  
{  
    set microwords to default  
    set all clause field(clfld) to empty  
    psorn()  
    DO {  
        token = next token  
        SWITCH(token) {  
            CASE STRING:    psorn()  
                           BREAK  
            CASE NUMBER:    psorn()  
                           BREAK  
            CASE CLAUSE:    IF(clause empty) report error  
                           fldctr = 0  
                           clsctr = clsctr + 1  
                           BREAK  
            CASE PERIOD:    IF(clause empty) report error  
                           fldctr = 0  
                           clsctr = clsctr + 1  
                           BREAK  
            default:        report error  
                           BREAK  
        }  
    }WHILE((token ne '.')&&(not too many clauses or fields))  
    IF(error in clause) report error  
    ELSE psnt()  
    IF(phasel) maddr = maddr + wrdtot  
    return  
}
```

FILENAME: PSORN.PDL

This function processes strings and numbers

```
psorn()
{
    clstok[clsctr, fldctr] = token
    IF(token is a number) {
        clfld[clsctr][fldctr] = -1
        clval[clsctr][fldctr] = token
    }
    ELSE {
        IF(string is defined) {
            clfld[clsctr][fldctr] = string number
        }
        ELSE {
            IF(phase = 1) {
                define label
                clfld[clsctr][fldctr] = -1
                clval[clsctr][fldctr] = label number
            }
            ELSE {
                IF(string is a label) {
                    clfld[clsctr][fldctr] = -1
                    clval[clsctr][fldctr] = label number
                }
                ELSE {
                    clfld[clsctr][fldctr] = -2
                    report error
                }
            }
        }
    }
    fldctr = fldctr + 1
    return
}
```

FILENAME: PSNT.PDL

This function generates the object code.

```
psnt()
{
    FOR(clsno = 0; clsno < clsctr; clsno++) {
        numcol = 0
        FOR(i = 0; i < clstot; i++) poscls[i] = TRUE
        FOR(f = 0; (f < MXFPC) && (clfld[clsno][f] != -3); f++) {
            pclsl(f)
            FOR(i = 0; i < clstot; i++)
                poscls[i] = poscls[i] && clspos[i]
            numcol++
        }
        FOR(i = 0; i < clstot; i++) {
            jstr = -1
            if(i > 0) jstr = cdpntr[i-1]
            if(f1 == ((cdpntr[i] - jstr) / 2)) poscls[i] = FALSE
        }
        totok = 0
        FOR(clsn = 0; clsn < clstot; clsn++) {
            if(poscls[clsn]) pclsl2(clsn, numcol)
        }
        FOR(h = 0; h < MXWPS; h++) {
            FOR(i = 0; i < numints; i++) {
                clsmask[h][clsno][i] = 0
                clsword[h][clsno][i] = 0
            }
        }
        switch(totok) {
            case 0:    report too few possible clauses
                      break
            case 1:    build object code for the clause
                      break
            default:   report too many possible clauses
                      break
        }
    }
    build the object for the word
    return
}
```

END

7-87

DTIC