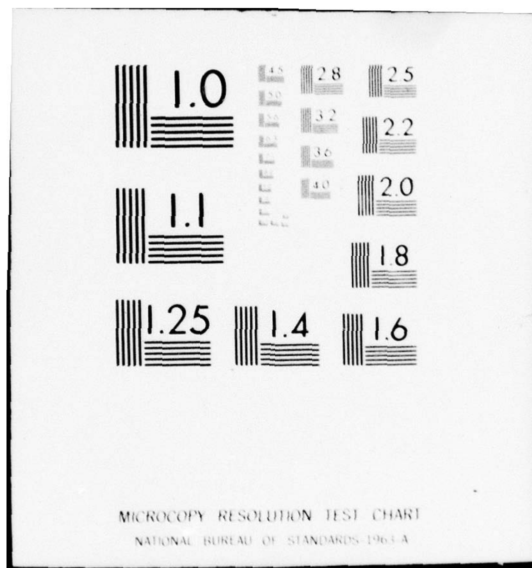
AD-A054 282 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
GENESIS: A LANGUAGE FOR DESCRIBING THE DEVELOPMENT OF PROGRAMS.(U)
FEB 78 W B TYLER, W M MCKEEMAN N00014-76-C-0682
UNCLASSIFIED TR-78-2-002 NL

| OF |
AD
A054282

END
DATE
FILMED
6-78
DDC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A054282

GENESIS: A LANGUAGE FOR DESCRIBING THE DEVELOPMENT OF PROGRAMS

by

W. B. Tyler

W. M. McKeeman

Technical Report No. 78-2-002

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

AD No. ____
DDC FILE COPY

INFORMATION SCIENCES
UNIVERSITY OF CALIFORNIA
SANTA CRUZ, CALIFORNIA 95064

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD A054282

DDC FILE COPY

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 78-2-002	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) GENESIS: A LANGUAGE FOR DESCRIBING THE DEVELOPMENT OF PROGRAMS.		5. TYPE OF REPORT & PERIOD COVERED Technical rept.
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) W. B. Tyler, W. M. McKeeman		8. CONTRACT OR GRANT NUMBER(s) ONR N00014-76-C-0682
9. PERFORMING ORGANIZATION NAME AND ADDRESS William M. McKeeman and Sharon Sickel, Information Sciences, UCSC, Rm. 239 AS, Santa Cruz, California 95064		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, Arlington, Virginia 22217		12. REPORT DATE Feb 1978
		13. NUMBER OF PAGES 27 p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research, University of California, 553 Evans Hall, Berkeley, California 94720		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

14 TR-78-2-002

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DDC RECEIVED MAY 25 1978 A

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A language, named GENESIS, is proposed for describing the program development process and bootstrapped programs in particular. It is concise (a seven line grammar) and general. The report contains syntax, semantics, examples and an approach to a formal semantics for the language.

DD FORM 1473, 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410 350

GENESIS: A LANGUAGE FOR DESCRIBING THE DEVELOPMENT OF PROGRAMS

by

W. B. Tyler

W. M. McKeeman

January 31, 1978

Board of Studies in Information Sciences
University of California at Santa Cruz
Santa Cruz, California 95064

Research was supported by the Office of Naval Research under Contract N00014-76-C-0682.

CONTENTS

ABSTRACT

1. THE PROBLEM OF DESCRIBING BOOTSTRAPPING

2. LITERATURE ON THE DESCRIPTION OF BOOTSTRAPPING

3. SYNTAX OF GENESIS

4. SEMANTICS OF GENESIS

5. PROGRAM GRAPHS FOR GENESIS

6. EXAMPLES OF GENESIS

7. AN AXIOMATIC APPROACH TO BOOTSTRAPPING

ACKNOWLEDGEMENT

REFERENCES

ABSTRACT

A language, named GENESIS, is proposed for describing the program development process and bootstrapped programs in particular. It is concise (a seven line grammar) and general. The report contains syntax, semantics, examples and an approach to a formal semantics for the language.

ACCESSION for
NTIS White Section ☒
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
Letter on file
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

1. The Problem of Describing Bootstrapping

The development of a compiler is usually a complicated process. This is especially so when the compiler is "bootstrapped", that is, written in the same language which it compiles. Bootstrapping involves several versions of the compiler. The first version must be written in an existing language. This may then be used to compile subsequent versions, and so on.

Our goal is to develop a suitable notation (which we call GENESIS) for describing such processes clearly and concisely. Although motivated by the bootstrapping problem, our notation is also suitable for describing many other processes involving execution of computer programs. A GENESIS program documents the steps taken in constructing a particular program. Were we to write a GENESIS program a priori, then executing it would carry out the construction steps. If we wish to send someone a bootstrapped program we may either send the completed product or the files from which it was constructed and the GENESIS program will recreate it. This is particularly useful if the bootstrap system is more concise than the translated version of the developed program.

This paper is organized in 7 sections. The third and fourth give the syntax and semantics of the proposed language. The fifth introduces a graphical interpretation for the language,and the sixth gives examples. The reader may well want to scan sections 5 and 6 before seriously reading the definitions in sections 3 and 4. The seventh section provides an outline of a formal semantics for the language.

2. Literature on the Description of Bootstrapping

The literature on bootstrapping is not extensive. A brief discussion may be found in McKeeman [70]. McKeeman [70] uses the T-diagrams of Bratman [61]. Earley and Sturgis [70] extends T-diagrams to cover interpreters and machines. Our notation is an outgrowth of some earlier work of Fay [76]. It is more general than T-diagrams, but does not display the interlocking of the various translators in as vivid a manner. We are free to use T-diagrams to illustrate all or part of a GENESIS program.

3. Syntax of GENESIS

3.1 Grammar

```
GENESIS-PROG = GENESIS-STMT (';' GENESIS-STMT)*
GENESIS-STMT = FILE-NAME* '←' TUPLE
             | PROC-NAME '(' NAME* ')' '=' TUPLE
             | FILE-NAME* '←' PROC-NAME '(' NAME* ')' '='
               '[' ( 'GLOBAL' NAME* ';' )? GENESIS-PROG ']'
TUPLE        = ( '·'? FILE-NAME | '·'? PROC-NAME '(' TUPLE ')' )*
NAME         = FILE-NAME | PROC-NAME
```

3.2 Remarks on Lexical Conventions

NAMEs are not formally defined. The user should feel free to use any naming conventions he finds convenient. We will make free use of subscripts, hyphenated names, or other suffixes, usually to name the language in which a program is written. The suffix is simply a part of the name, is not grammatically defined, and has no semantic significance in GENESIS. NAMEs may be

delimited by blanks or by GENESIS operators. Commas will be considered as equivalent to blanks and may be freely used to increase readability. We will use the following comment conventions:

1. comments are begun by the '¢' character
2. comments are ended by '¢' or the end of the line on which they are written, whichever comes first.
3. comments are equivalent to blanks.

4. Semantics of GENESIS

4.1 Objects

GENESIS deals with two fundamental kinds of objects -- files and processors. Names may be used to refer to either kind of object. GENESIS programs are generally considered to be embedded in an environment containing some "primitive" files and processors.

4.1.1 Files

A GENESIS file is a sequence of data objects. A file may be either primitive or the output of exactly one process. One file may be used as input by any number of processes. Files are written and read strictly sequentially. A file may be written and read concurrently as long as the reading process does not read past the end of the information being written. A special empty file called NULL may be used to discard unwanted outputs or supply empty inputs.

4.1.2 Processors

A processor is an object which maps a (possibly empty) tuple of input files to a (possibly empty) tuple of output files. The image of an input tuple may

not be uniquely defined. That is, a processor is a function from tuples to sets of tuples. The combination of a processor with specific input and output files will be called a process. Any given process will produce as output some tuple in the set of tuples consituting the image of the input. We use the term "processor" to emphasize that in general the "execution" of a process takes time. Further, it is meaningful to consider a "partially executed" state of a process in which part of the input has been read and part of the output has been written.

A computer may be a GENESIS processor. The combination of a computer and operating system may also be a processor, though at a different level. One processor to which we will frequently refer is a human being, usually named BYHAND.

4.1.3 Names

Names are used to refer to files and processors. At any given time a name may be bound to at most one object. However, an object may be bound to many names.

4.2 GENESIS Program Meaning

The meaning of GENESIS programs is defined in terms of bindings and processor executions. The following sections give the meaning of each GENESIS construct in terms of the bindings it performs and the processes it creates. The reader may wish to refer to the examples in Section 6.

4.2.1 GENESIS-PROG = GENESIS-STMT (';' GENESIS-STMT)*

A GENESIS program is a sequence of statements, each of which performs bindings. There are two kinds of statements. The first (Section 4.2.2) binds names to files. The second (Sections 4.2.3 and 4.2.4) defines a processor and

binds a name to it. Zero or more processes may be created by a GENESIS statement. We will often describe the execution of GENESIS programs as though there were no parallelism in processor execution, but in general it will be advantageous to allow for parallelism in execution. If the sequence of binding is unchanged, parallelism will not alter the effect of a GENESIS program.

4.2.2 GENESIS-STMT = FILE-NAME* '←' TUPLE

The TUPLE is evaluated, then the FILE-NAMES are bound to the files constituting the TUPLE in order from left to right. When the number of files does not match the number of names a GENESIS interpreter should warn the user. If more FILE-NAMEs than files are present the excess names are bound to NULL. If more files than names are present the excess files are not bound. Mismatches may occur when a processor produces outputs not needed by the user (e.g., a trace of parser activity) or when the user wishes to allow (e.g., in a procedure declaration) for outputs which may not be available in all cases.

4.2.3 GENESIS-STMT = PROC-NAME '(' NAME* ')' '=' TUPLE

This statement defines a processor and binds the PROC-NAME to the processor. (See Section 6.1.1.) The NAMEs in parentheses are formal input parameters to the processor. These formal inputs may be either files or processors. Once defined, the processor may be supplied with actual input parameters and used as a TUPLE. (See Section 4.2.5.) When this is done, two types of substitution are performed. First, actual parameters are substituted for formal parameters in the processor body (the TUPLE in the definition); second, the body with actual parameters is substituted for the processor call. This is equivalent to the use of name parameters in ALGOL 60. Dots in the processor definition modify this substitution algorithm as follows.

As can be seen from the above, if a procedure body contains names other than formal parameter names the bindings of those names will be evaluated at the time the processor is used, each time it is used. The dot, or "value of", symbol may be used to override this. If a name in a processor definition is immediately preceded by a dot, the binding of the name is evaluated at the time the processor is defined, rather than when it is executed. This could be implemented in the following procedure, which could be carried out at the time the processor is defined. For every dotted name in the processor body, create a new, unique name. This name should be one which cannot be written by a GENESIS programmer. Bind the new name to the same object to which the dotted name is bound, and replace the dot and name in the processor body with the new name.

Dotting may be used to increase efficiency as shown in the example of Section 6.5. On the other hand, undotted names may also be useful. For example, if a processor uses a compiler, leaving the compiler name undotted allows an improved compiler to be substituted by simply rebinding the compiler name. This involves no change to any processor using the compiler name.

4.2.4 GENESIS-STMT = FILE-NAME* '←' PROC-NAME '(' NAME* ')' '='

'[' ('GLOBAL' NAME* ';')? GENESIS-PROG ']'

This form of statement also defines a processor and binds the PROC-NAME to it. The main differences between this form and the form of Section 4.2.3 are that this form has formal output parameters as well as formal inputs, and the body of this form of processor is a bracketed GENESIS program.

When a processor defined as in this section is used in a TUPLE (See Section 4.2.5) its value is computed by executing the bracketed GENESIS program constituting the body of the processor, after substituting actual for formal inputs. The

body should contain statements binding the formal output names (the NAMEs to the left of the '←' symbol in the processor definition) to files. The output of the processor is then the tuple of files bound to the formal output names.

All names occurring in the body including formal inputs and outputs are strictly local to the processor unless they are listed in the GLOBAL part. (See Section 4.3.) However names occurring in the GLOBAL part are known outside the procedure. The effect of dotted names is the same for this form of procedure declaration as for the form of Section 4.2.3.

4.2.5 TUPLE = ('.'? FILE-NAME | '.'? PROC-NAME '(' TUPLE ')')*

A TUPLE is an ordered set of files. Each file is designated either by name or as part of the output of a process.

4.2.5.1 '.' ? FILE-NAME

The FILE-NAME names a file which is a component of the TUPLE. The dot is meaningful only if the TUPLE is part of the body of a processor definition. (Dots are explained in Section 4.2.3.)

4.2.5.2 '.'? PROC-NAME '(' TUPLE ')'

The value of this construct is the output TUPLE of the processor, applied to the actual inputs specified by the input TUPLE. To eliminate possible ambiguities, the effect must be equivalent to first evaluating the input TUPLE, next making the needed substitutions in the processor body, and finally executing the processor.

Recall that this construct is used as part of a TUPLE. (Section 4.2.5.) The output of the processor is generally more than one file. Each file output

by the processor becomes a component of the TUPLE, rather than having the set of outputs as a single "structured" component.

4.2.6 NAME = FILE-NAME | PROC-NAME

NAME is used in the grammar in declarations where either a file name or a processor name may be appropriate.

4.3 Scope of Names

The following rules define the scope of names occurring in GENESIS programs.

1. Any names used in a GENESIS program are local to the program unless affected by one or more of the rules below.
2. In a processor definition the formal input and output names have the processor definition as their entire scope. If processor definitions are nested, the innermost definition applicable controls the name.
3. In a bracketed GENESIS program (see 4.2.4) names listed after the word GLOBAL are known in the containing program. The containing program may declare these names global to itself also, so that the scope of a name is the smallest containing program which does <u>not</u> declare the name to be global.

5. Program Graphs for GENESIS

We will find it useful to represent GENESIS programs by directed graphs. The nodes represent files and processes; edges represent data flow. At times the files between two processes may be omitted. We will use rectangles for processor nodes and circles for file nodes.

6. Examples of GENESIS

The use of GENESIS is illustrated by several examples.

6.1 Simplified Compiler

A GENESIS program and graphs are shown illustrating the construction of a simplified compiler from component pieces.

6.1.1 GENESIS Program

```
COMPILE(SOURCE) = EMIT(PARSE(SCAN(SOURCE)))
```

6.1.2 Remarks

The GENESIS program defines a *processor* named COMPILE. The result of applying COMPILE to a source file (formal input SOURCE) is the same as that of applying EMIT to the result of applying PARSE to the result of applying SCAN to SOURCE, where EMIT, PARSE, and SCAN are all names of primitive (in this *application*) processors. If this statement were embedded in a larger GENESIS program the names EMIT, SCAN, and PARSE would not need to be bound to processors at the time the definition was executed. But when COMPILE is actually applied to an input they must have been bound and their bindings will then be used. This program defines the two graphs of figures 6.1.2a and 6.1.2-b to be equivalent.

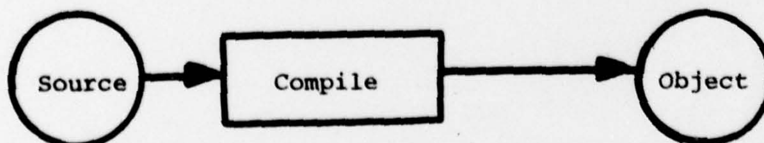


Figure 6.1.2-a

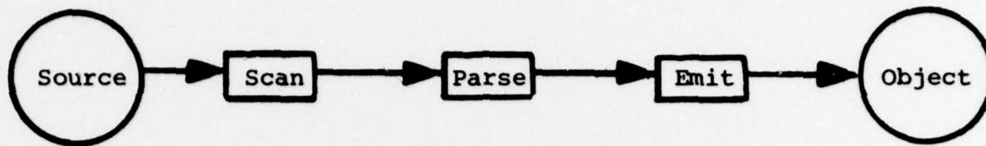


Figure 6.1.2-b

6.2 Zilog Z-80 Assembler History

6.2.1 GENESIS Program

DATE	GENESIS Statement
9/75	$\text{ASM}_{\text{PL/I}} \leftarrow \text{BYHAND}();$
9/75	$\text{ASM}_{\text{S370}} \leftarrow \text{PL/I}(\text{ASM}_{\text{PL/I}});$
11/75	$\text{ASM}(\text{X}) = \text{S370}(\text{ASM}_{\text{S370}},\text{X});$
12/75	$\text{ASM}_{\text{ASM}} \leftarrow \text{BYHAND}(\text{ASM}_{\text{PL/I}});$
12/75	$\text{ASM}_{\text{Z80}} \leftarrow \text{ASM}(\text{ASM}_{\text{ASM}});$
4/76	$\text{ASM}(\text{X}) = \text{Z80}(\text{ASM}_{\text{Z80}},\text{X});$ ¢ redefining ASM
6/76	$\text{ASM}_{\text{Z80}} \leftarrow \text{ASM}(\text{ASM}_{\text{ASM}});$ ¢ the bootstrap step
11/76	$\text{REL}_{\text{ASM}} \leftarrow \text{BYHAND}(\text{ASM}_{\text{ASM}});$ ¢ relocatable ASM
1/77	$\text{REL}_{\text{Z80}} \leftarrow \text{ASM}(\text{REL}_{\text{ASM)}};$ ¢ bootstrapping again
2/77	$\text{REL}(\text{X}) = \text{Z80}(\text{REL}_{\text{Z80}},\text{X});$
3/77	$\text{LINK}_{\text{ASM}} \leftarrow \text{BYHAND}();$ ¢ Linker for relocatable form of mach lang
?	$\text{LINK}_{\text{Z80}} \leftarrow \text{ASM}(\text{LINK}_{\text{ASM}});$
?	$\text{LINK}(\text{X}) = \text{Z80}(\text{LINK}_{\text{Z80}},\text{X});$
?	$\text{LINK}_{\text{REL}} \leftarrow \text{BYHAND}(\text{LINK}_{\text{ASM}});$
?	$\text{LINK}_{\text{Z80}} \leftarrow \text{LINK}(\text{REL}(\text{LINK}_{\text{REL}}));$ ¢ bootstrapping yet again

DATE	GENESIS Statement
?	$LINK(X) = Z80(LINK_{Z80},X)$; ¢ redefining LINK
	$REL_{REL} \leftarrow BYHAND(REL_{ASM})$; ¢ possible future effort
	$REL(X) = Z80(.REL(.REL_{REL}),X)$; ¢ possible future effort

6.2.2 Remarks

Several primitive processors are assumed: BYHAND, a human or humans; S370, an IBM 370 computer; PL/I, a compiler for PL/I into S370 machine code; and Z80, a Zilog Z-80 computer. Names of program files have been subscripted with the names of the languages in which they are expressed.

6.3 Description of Compiler Structure

This section uses GENESIS to describe the compiler structure presented in McKeeman [72].

6.3.1 Graphical Description

We will generate a GENESIS program describing the graph of Figure 6.3.1-a. The processor named SYNTHESIS is expanded in Figure 6.3.1-b. Figure 6.3.1-c shows the completed processor, COMPILE.

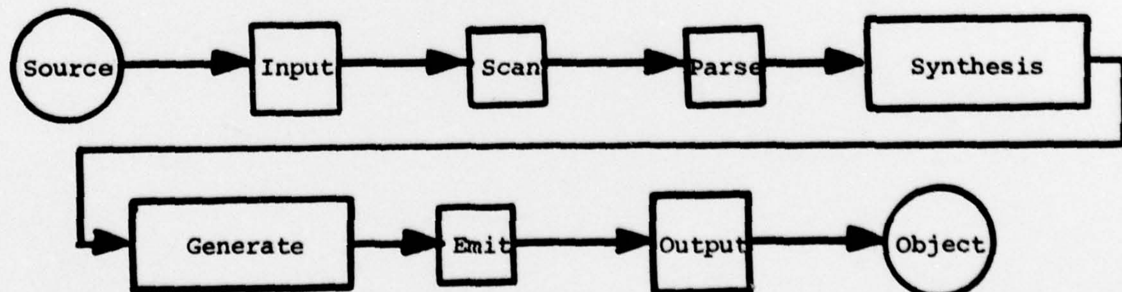


Figure 6.3.1-a

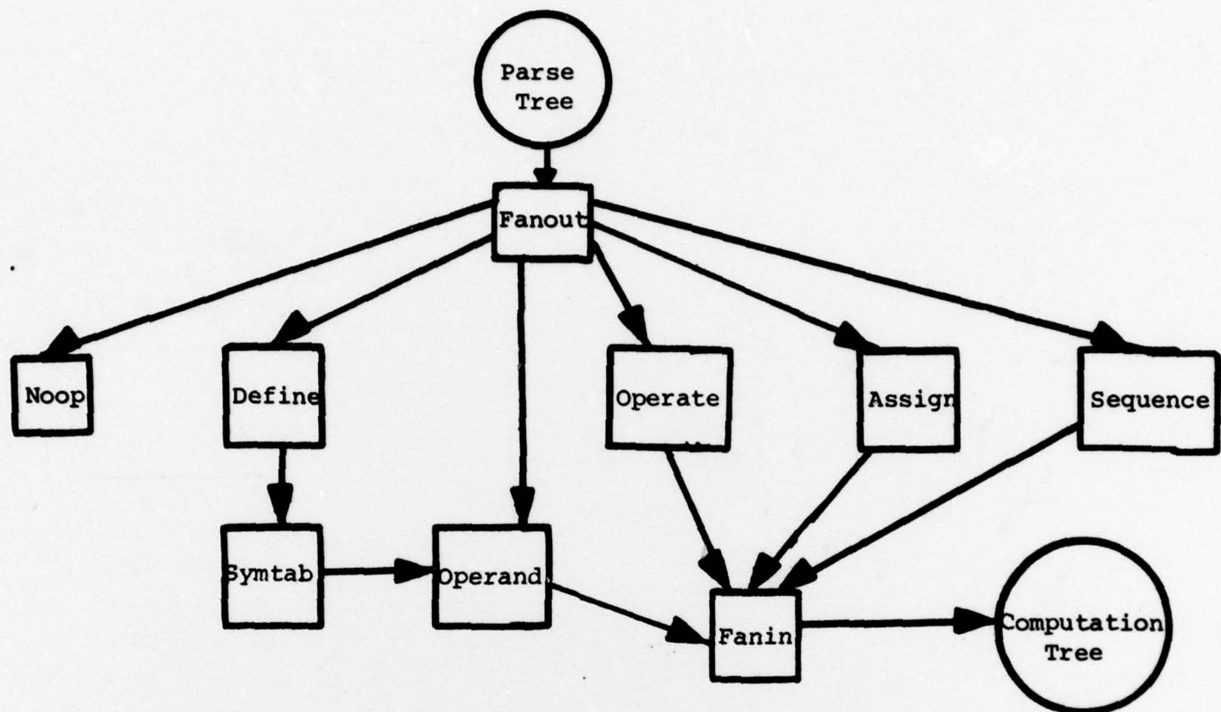


Figure 6.3.1-b

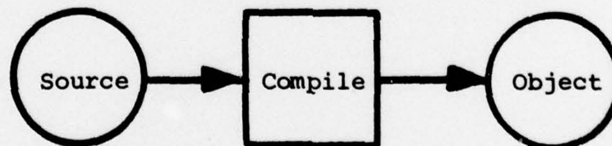


Figure 6.3.1-c

6.3.2 GENESIS Program

```
COMPILE(SOURCE) = OUTPUT(EMIT(GENERATE(SYNTHESIS(
                    PARSE(SCAN(INPUT(SOURCE)))))));
COMPUTATIONTREE ← SYNTHESIS(PARSETREE) =
      [NULL,F2,F3,F4,F5,F6 ← FANOUT(PARSETREE);
         COMPUTATIONTREE ← FANIN(OPERAND(SYMTAB(DEFINE(F2)),
                                                F3),
                                 OPERATE(F4),
                                 ASSIGN(F5),
                                 SEQUENCE(F6) )]
```

6.3.3 Remarks

As it stands, the program above is not bound to specific processors, only to processor names. Thus if EMIT, for example, is changed in the environment the processor COMPILE is implicitly changed.

6.4 XCOM Bootstrapping from McKemman [70]

6.4.1 GENESIS Program

$XCOM1_{ALGOL}$ ← BYHAND();

$XCOM1_{XPL}$ ← BYHAND($XCOM1_{ALGOL}$);

$XCOM1_{B5500ML}$ ← ALGOL($XCOM1_{ALGOL}$);

XCOM1(X) = B5500MCP($XCOM1_{B5500ML}$,X);

$XCOM2_{XPL}$ ← BYHAND($XCOM1_{XPL}$);

$XCOM2_{S360ML} \leftarrow XCOM1(XCOM2_{XPL});$

$XCOMSM_{S360ML} \leftarrow ASSEMBLE(BYHAND());$

$XCOMSM(X,Y) = S360(XCOMSM_{S360ML},X,Y);$

$XCOM2(X) = XCOMSM(XCOM2_{S360ML},X);$

$XCOM3_{XPL} \leftarrow BYHAND(XCOM2_{XPL});$

$XCOM3_{S360ML} \leftarrow XCOM2(XCOM3_{XPL});$

$XCOM3(X) = XCOMSM(XCOM3_{S360ML},X);$

$USERCOM_{XPL} \leftarrow BYHAND();$

$USERCOM_{S360ML} \leftarrow XCOM3(USERCOM_{XPL});$

$USERCOM(X) = XCOMSM(USERCOM_{S360ML},X)$

6.4.2 Remark

The T-diagram of Figure 6.4.2-a, which is taken from McKeeman [70], describes the program of 6.4.1.

6.5 Example of Dot Use

Suppose that a subroutine library for an Algol-like language is implemented by surrounding every user program by an outer block containing procedure declarations for the library. The structure is shown below

```
Begin
    procedure declaration for library
    user's program
End
```

A HISTORY OF THE XPL SYSTEM

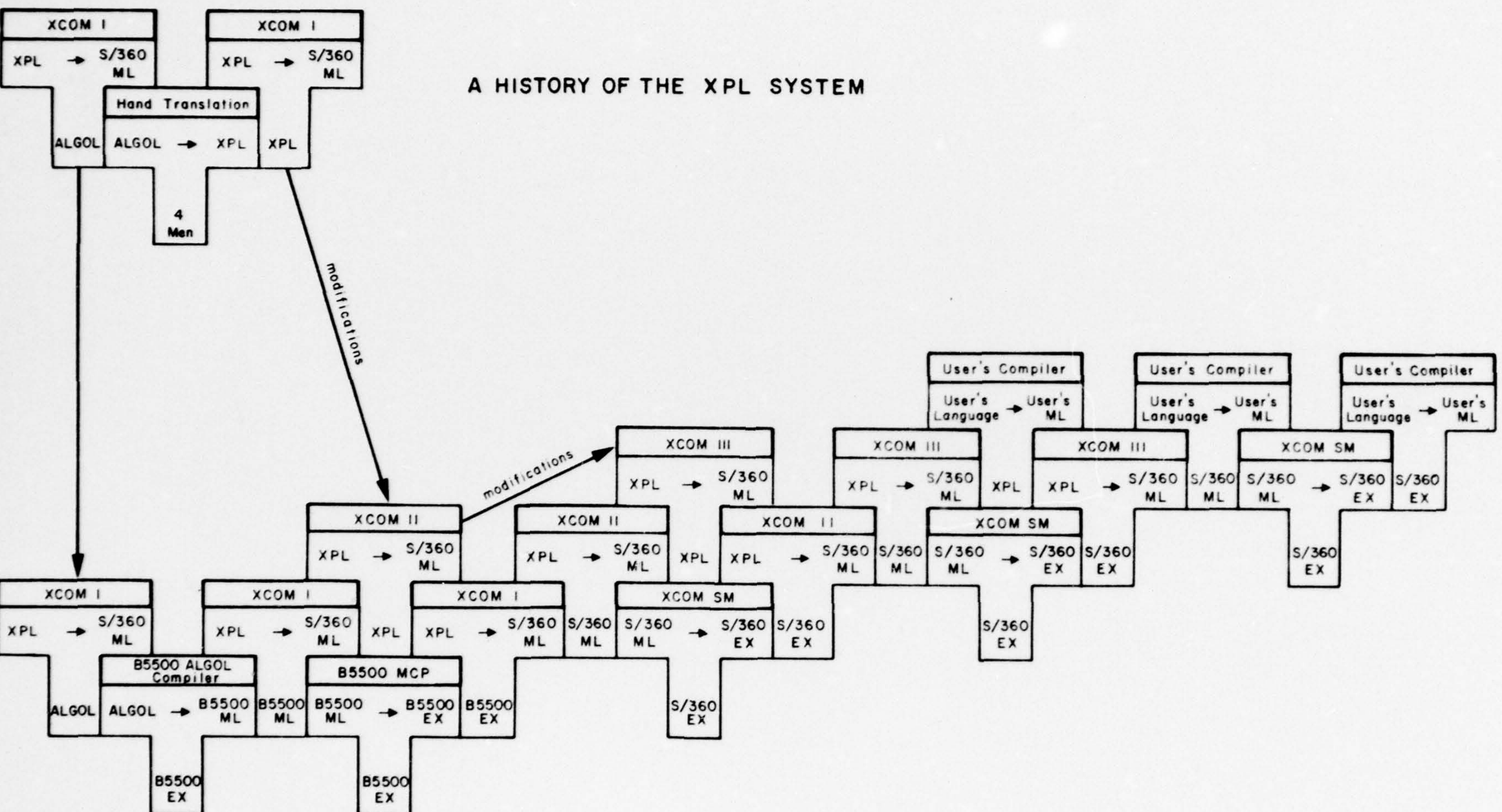


Figure 6.4.2-a

This might be accomplished by two processors: the compiler proper and a concatenator where the concatenator produces the structure shown above from the user program and the procedure library. A GENESIS declaration for this system is

```
COMPILE(USERPROG) = PCOMPILE(CONCAT(LIB,USERPROG))
```

Note that the library must be recompiled each time COMPILE is used. However if COMPILE were declared as

```
COMPILE(USERPROG) = .PCOMPILE(.CONCAT(.LIB,USERPROG))
```

this could be avoided. Since the processor names in the definition as well as the library file name are dotted, the processing can be done once -- at the time of definition -- stopping only when input from USERPROG is needed. This partially executed process can then be saved to be bound to many different user programs. But the work of compiling the procedure library is done only once.

7. An Axiomatic Approach to Bootstrapping

This section develops several axioms describing relations between processors, languages, and translators. The axioms are related to GENESIS programs in such a way that constructing a proof of correctness of some bootstrapping steps is equivalent to constructing a GENESIS program for carrying out those steps. Our axioms include the semantic information expressed by T-diagrams.

7.1 Processors and Programs

Until now, we have treated all inputs to a processor as being more or less the same sort of thing. We now find it useful to distinguish a special class of processors which we will call "programmable". A programmable processor is one which is intended to simulate any of a variety of processors. The particular processor simulated depends on an input file called the "program". When a program is bound to a programmable processor the result is a new processor. It should be clear that any processor may be considered to be programmable and any of its inputs may be considered to be the program. There is really a complete symmetry between program and data. However we will find it useful to arbitrarily call some processors programmable. We will use the convention that the leftmost input parameter to a programmable processor is its program.

7.2 Languages and Translators

A language is a set of valid programs together with a mapping from programs in the set into processors. Two programs will be said to be equivalent if they map to equivalent processors. (Two processors are equivalent if they perform the same mapping of inputs to outputs. It may be useful to define classes of "erroneous" inputs and define processors to be equivalent if their mappings for non-erroneous inputs are the same.) A translator is a processor which maps

programs in one language into equivalent programs in another language.

7.3 Some Predicates

We explain some predicates used in the axioms of Section 7.4.

7.3.1 M(m,e)

This predicate is true if and only if m is a programmable processor accepting language *e*. That is, if m is bound to a program p in e the resulting processor is equivalent to the one defined by the program→processor map of e.

7.3.2 T(t,e,e')

T is true if and only if t is a process which maps programs in language e into equivalent programs into equivalent programs in language e'.

7.3.3 TP(c,e_c,e,e')

TP is true if and only if c is a program in language e_c and the processor described by c is a translator from e to e'. That is, c is a compiler.

7.4 The Axioms

We give our axioms in the form

preconditions

GENESIS statement

result

The meaning of the form is that if the preconditions are met, then after executing the GENESIS statement the result will be true.

7.4.1 Processor Construction

This axiom describes the construction of a processor for programs in language e given the existence of a processor for programs in language e' and a translator t from e to e'.

preconditions

M(m',e') & T(t,e,e')

GENESIS statement

$m(p,n_1,n_2,...) = .m'(.t(p),n_1,n_2,...)$

result

M(m,e)

The symbols n_1, n_2 are arbitrary names which may be present.

7.4.2 *Translator Construction*

A method of constructing a translator from a translator program and a processor is shown in this axiom.

preconditions

$TP(c,e_c,e,e')$ & $M(m,e_c)$

GENESIS statement

t(p) = .m(.c,p)

result

T(t,e,e')

7.4.3 Program Translation

This axiom describes the translation of a program p expressed in language e into a program p' expressed in e'. The predicate P whose truth is described as being preserved by the axiom takes a program text and language as parameters. P may be any predicate whose truth depends only on the equivalence class of the

processor onto which the program is mapped by the language's mapping.

<u>preconditions</u>

$P(p,e)$ & $T(t,e,e')$

<u>GENESIS statement</u>

$p' \leftarrow t(p)$

<u>result</u>

$P(p',e')$

Note that the predicate TP is a valid replacement for P if its third and fourth parameters are held constant. Thus we have the useful result

<u>preconditions</u>

$TP(p,e_p,e,e')$ & $T(t,e_p,e_q)$

<u>GENESIS statement</u>

$q \leftarrow t(p)$

<u>result</u>

$TP(q,e_q,e,e')$

7.5 Example

Suppose that we have a compiler, $PASCAL_{ALGOL}$, written in ALGOL, which translates PASCAL into ML, a machine language. We also have $ALGOL_{ML}$, an ALGOL compiler expressed in ML and translating ALGOL into ML. Finally we have MAC, a computer which executes ML programs. These facts can be summarized as follows:

$TP(PASCAL_{ALGOL},ALGOL,PASCAL,ML)$

$TP(ALGOL_{ML},ML,ALGOL,ML)$

$M(MAC,ML)$

We wish to generate a processor, PASCALX, to execute PASCAL programs. With the preconditions above we execute the following GENESIS program. Comments give the result of each statement.

$ALGOL(p) = .MAC(.ALGOL_{ML},p);$

¢ T(ALGOLT, ALGOL, ML)

$PASCAL_{ML} \leftarrow ALGOLT(PASCAL_{ALGOL});$

¢ TP($PASCAL_{ML}$, ML, PASCAL, ML)

$PASCALT(p) = .MAC(.PASCAL_{ML},p);$

¢ T(PASCALT, PASCAL, ML)

$PASCALX(p, n_1, n_2, \ldots) = .MAC(.PASCALT(p), n_1, n_2, \ldots);$

¢ M(PASCALX, PASCAL)

Acknowledgement

The authors would like to thank Michael Fay, David Jacobson, Doug Michels and Dan Ross for several helpful discussions.

Bratman, H., [61]. "An Alternate Form of the UNCOL Diagram," _Communications of the Association for Computing Machinery_, 4: 3(1961), 142.

Earley, J., and H. Sturgis, [70]. "A Formalism for Translator Interactions," _Communications of the Association for Computing Machinery_, 13:10 (1970), 607-617.

Fay, M., [76]. Unpublished manuscript.

McKeeman, W. M., et al., [70]. _A Compiler Generator_, Prentice-Hall, Englewood Cliffs, N. J. (1970).

McKeeman, W. M., [72]. "Compiler Structure" in _Proceedings of the First USA-JAPAN Computer Conference_, Tokyo, 1972.