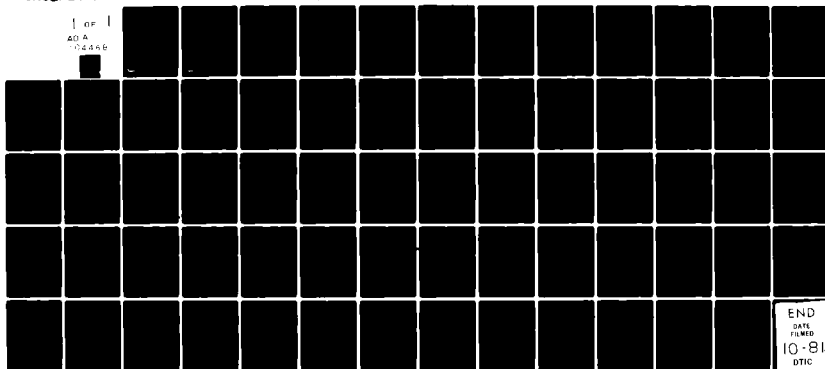


AD-A104 468

SRI INTERNATIONAL MENLO PARK CA COMPUTER SCIENCE LAB F/6 9/2
OBJ-1: A STUDY IN EXECUTABLE ALGEBRAIC FORMAL SPECIFICATION.(U)
JUL 81 J A GOGUEN, J MESEGUER N00014-80-C-0296
NL

UNCLASSIFIED

1 of 1
AD A
02456



END
DATE
FILMED
10-81
DTIC

FILE COPY



AD A104468

LEVEL



OBJ-1, A STUDY IN EXECUTABLE ALGEBRAIC FORMAL SPECIFICATION

Final Report, Fiscal Year 1980

July 1981

By: Joseph A. Goguen, Senior Computer Scientist
José Meseguer, Computer Scientist
Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

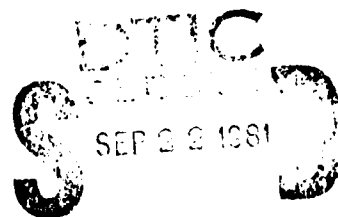
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

Attention: Dr. Robert Grafton

This document has been approved
for release by the Office of Naval Research
under the provisions of the
Automatic Desclassification and
Downgrading Schedule.

SRI Project 1350-100
Contract No. N00014-80-0296

NR-044-006



A

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-2046

SRI International



**OBJ-1, A STUDY IN EXECUTABLE
ALGEBRAIC FORMAL
SPECIFICATION**

Final Report, Fiscal Year 1980

July 1981

By: Joseph A. Goguen, Senior Computer Scientist
José Meseguer, Computer Scientist
Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

Attention: Dr. Robert Grafton

SRI Project 1350-100
Contract No. N00014-80-0296

Approved:

Jack Goldberg, Director
Computer Science Laboratory

David H. Brandin, Vice President and Director
Computer Science and Technology Division

Table of Contents

I. REVIEW OF PROBLEM AND APPROACH	1
II. PROGRESS	3
1. <u>Implementations</u>	3
2. <u>Applications and Examples</u>	3
3. <u>Theoretical Foundations</u>	4
4. <u>Work on Related Specification Languages</u>	5
5. <u>Publications</u>	6
III. RESULTS IN PROGRESS	7
1. <u>Implementations</u>	7
2. <u>Foundations</u>	8
APPENDICES	9
A OBJT SUGGESTIONS, BUGS AND COMMENTS	10
B PROGRAMMING LANGUAGE DEFINITION	15
C SYMBOLTREE SPECIFICATION	33
D SPECIFICATION OF GRAPHS AND PATHS	42
E TECHNIQUES FOR HIGHER ORDER SPECIFICATIONS AND OTHER SURPRISES	46
F PARTIAL ALGEBRAS WITH EQUATIONALLY DEFINED DOMAINS	53
G STRICT ERROR ALGEBRAS DEFINED BY TESTS	57
H MODEL-THEORETIC CHARACTERIZATION OF RELATIONAL CLASSES OF PROGRAM	60
SCHEME INTERPRETATIONS	
REFERENCES	60

Author	For
Project	☒
Task	☐
Phase	☐
Location	
<i>Letter on file</i>	
Classification	
Accession Codes	
Indexing	
Notes	
A	

I. REVIEW OF PROBLEM AND APPROACH

As hardware becomes less expensive, it is increasingly appropriate and important to put greater emphasis on reducing the cost of software. It is well known that a great deal of the cost of software arises in debugging, and particularly in debugging small changes in large programs during the process of maintaining them. Moreover, it seems clear that a major contributing factor to difficulties of this kind is the generally poor quality of documentation of such large programs, so that the programmer who has to do the maintenance has great difficulty in determining the effects of changes which he makes in the code. It is becoming increasingly clear to the computer science community that formal program specification is promising as a solution, and that specialized specification languages are helpful in expressing such program specifications.

The goal of this research has been to develop a formal and executable algebraic specification language which can be used to specify a variety of application programs, such as database systems, compilers and interpreters for programming languages, and business systems. An advantage of formality in this context is that each specification has a unique unambiguous meaning, so that it is actually meaningful to ask whether or not a given program in fact satisfies a given specification. An advantage of executability is that test cases can be run directly on the specification, to examine properties of programs before they are written, and to help in debugging specifications. The latter is important because large specifications, like large programs, are usually wrong as first written.

We have been investigating the utility of a number of potential advantages of the algebraic approach, including the following:

1. Achievement of a high level of modularity in a natural way;
2. Achievement of a high level of abstraction in a natural way;
3. The possibility of executing test cases;
4. User definition of data types and control structures, using any desired syntax, including pre-fix, post-fix, and "mix-fix" operators, as well as coercions;
5. The specification of error and exception conditions, as well as their handling, and recovery;

6. The use of parameterized abstract modules as a method for structuring specifications;
7. Algorithms for checking consistency and other desirable properties of specifications (e.g., the Knuth-Bendix algorithm); and
8. Provision of a completely rigorous semantics for all these features.

II. PROGRESS

This section describes the progress which we have made on this project.

1. Implementations

J. Tardo has provided a new implementation of OBJ, called OBJT20, which fixes most of the bugs discovered in his previous OBJT implementation. Both OBJT and OBJT20 are currently running on SRI's DEC-20 system, but we will soon retire the old OBJT, and rename the new OBJT20 to OBJT. The bugs found in OBJT are documented in a memorandum, reproduced in Appendix A here, based on our extensive experience using OBJT.

The new OBJT20 implementation runs faster, takes less space, permits lower case letters, supports TOPS20-style command completion with the <escape> character, and allows arbitrarily long file names. Moreover, as described in detail in Appendix A, it corrects many of the bugs which we found in OBJT. The only disadvantage is that it will not run on DEC-10 machines, but only on 20s. Joseph Tardo is a student at UCLA whose just completed thesis[Tardo 81] is on these implementations; he is now working at DEC.

2. Applications and Examples

J. Goguen and K. Parsaye-Ghomi have written a paper entitled "Algebraic Denotational Semantics using Parameterized Abstract Modules" which gives some new techniques for defining the semantics of programming languages, based on the features of OBJ. These techniques are illustrated with the definition of a strongly typed programming language having integer and boolean expressions, conditionals, iteration, block structure, and side-effect-only procedures which can also be passed as parameters. Because procedures can be passed as values, and because the language is strongly typed, the type system must be higher order.

One interesting result of this research was a correspondence between some of the basic constructions in denotational semantics, and some definitions in the OBJT library of basic parameterized objects. For example, the denotationalists' Cartesian product of domains corresponds to PAIR in <OBJT>LIB.OBJ, and the denotationalists function domain construction (usually denoted by "arrow") corresponds to ARRAY.

The paper was presented at the "International Conference on Formalization of Programming Concepts," held in Peniscola, Spain, this April, and has appeared

in the proceedings[Goguen & Parsaye-Ghomi 81]. An improved version of the language definition is reproduced in Appendix B of this report. This version shows better how the OBJ library of parameterized specifications is used; it better illustrates the compilation of an interpreter, and it has better mnemonics and better test cases. Many people have been surprised at how short and modular is the complete definition of this fairly nontrivial programming language, and how easy it is to modify it to get definitions of related languages.

Goguen has defined a new data type, called *symboltree*, in OBJ. The purpose of this data type is to provide for fast checking of certain information, such as the types of variables, during interactive editing. The definition, and a large number of test cases showing how the operations of the data type work, is given in Appendix C of this report.

K. Parsaye-Ghomi, with A. B. C. Sampaio of UCLA, has written a specification of a hardware multiplexor in OBJT; a rough draft paper exists.

In response to a challenge from Prof. H. Reichel (of Leipzig), we have shown how to define graphs, and paths in a graph, using the OBJ error algebra formalism, rather than his formalism using partially defined operations. The OBJT code is given in Appendix F. This example is also referred to in Appendix F.

We have discovered some rather tricky methods to achieve certain effects which it might seem cannot be done in OBJT, such as higher order operations, defining the natural numbers from the integers, and imposing new equations on old objects. These are illustrated in Appendix E to this report.

3. Theoretical Foundations

Meseguer and Goguen are working hard on the basic theory of error algebras which underlies OBJ's approach to error definition, handling, and recovery. Although a number of surprising and subtle difficulties have been uncovered, we are convinced that it will be possible to get correct versions of all the basic algorithms needed for OBJ-1. Current efforts are centered on underlying semantic issues, and on the relationship to partial algebras.

We have found a small but vicious flaw in the usual deductive system for many-sorted equational logic, as used for example in most work on abstract

data types; rather shockingly, this system is not sound. Not only have we given some new axioms which are sound and complete, but we have found sufficient conditions such that the old deduction system works anyway[Goguen & Meseguer 81]. Since there are so many people using equations now, it seems appropriate that this paper reach the fairly broad audience which reads SIGPLAN Notices. A full version of this paper, with all the proofs, is in an advanced stage of preparation.

Meseguer has written a deep paper entitled "A Birkhoff-like Theorem for Algebraic Classes on Interpretations of Program Schemes" which was also presented at the Peniscola Conference, and appears in its proceedings[Meseguer 81]. This work, which is summarized in Appendix G to this report, follows up an earlier paper "Varieties of Chain-complete Algebras" which appeared in the prestigious special issue of the Journal of Pure and Applied Algebra honoring the sixtieth birthday of Professor Saunders MacLane of the University of Chicago[Meseguer 80].

K. Parsaye-Ghomi has developed, as part of his nearly completed Ph.D. dissertation at UCLA, a method for extending OBJ to handle higher order operations and equations. We have found that this would be extremely useful in the specification of programming languages, as illustrated in[Goguen & Parsaye-Ghomi 81]. The thesis and some papers based on it should be available soon.

4. Work on Related Specification Languages

R. Burstall and J. Goguen have written an informal introduction to their powerful specification language CLEAR[Burstall & Goguen 81]. This paper, which will appear in the Academic Press book of Liege lectures, with papers of Dijkstra, Boyer and Moore, and Manna, includes a sophisticated specification of a garbage collector.

Burstall and Goguen are also working on the design of a much more user-oriented specification language with the same underlying semantics as CLEAR; this new language is to be called ORDINARY, as in some ways it also builds on the previous generation SRI specification language SPECIAL[Levitt, Robinson & Silverberg 79]. A draft report is available on this subject[Goguen & Burstall 80a].

Burstall and Goguen are also working on a program design system which will be

based on their previous work on specification. This system is called CAT, and a concept piece outlining their intentions for it is available as an SRI technical report[Goguen & Burstall 80b] and is included with this report. One general idea is to construct a program transformation system which explicitly addresses the very important fact that in order to verify correctness of an application of a transformation, it is necessary to know some parts of the theory of the program, and to understand how various parts of the theory fit together with various parts of the program.

[Burstall & Goguen 78] gives a complete formal semantics for CLEAR, the first time that denotational semantics has been given for a specification language. [Goguen & Burstall 78] gives the mathematical background on theories on which [Burstall & Goguen 78] is based; we have just completed final revisions of this paper for publication.

[Goguen 80] gives foundations of OBJ, in particular for the use of rewrite rules to implement initial algebras, and or using the Knuth-Bendix algorithm for automatic verification for algebraic specifications.

5. Publications

This subsection lists papers, either published or submitted, which have been supported in whole or in part by this project. It also includes Ph. D. theses.

1. Burstall, R. M., and Goguen, J. A. The Semantics of CLEAR, a Specification Language. In Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification, Lecture Notes in Computer Science, volume 86, pages 292-332. Springer-Verlag, 1980.
2. Burstall, R. M. and Goguen, J. A. An Informal Introduction to CLEAR, a Specification Language. In Boyer, R. and Moore, J (editor), The Correctness Problem in Computer Science, . Academic Press, 1981.
3. Goguen, J. A. How to Prove Algebraic Inductive Hypotheses without Induction: with applications to the correctness of data type representations. In W. Bibel and R. Kowalski (editor), Proceedings, 5th Conference on Automated Deduction, pages 356-373. Springer-Verlag, Lecture Notes in Computer Science, volume 87, 1980.
4. Goguen, J. A. and Burstall, R. M. Some Fundamental Properties of Algebraic Theories: a Tool for Semantics of Computation. Technical Report, Dept. of Artificial Intelligence, University of Edinburgh, 1978. DAI Research Report No. 5; to appear in Theoretical Computer

Science.

5. Goguen, J. A. and Burstall, R. M. An Ordinary Design. Technical Report, SRI International, 1980. Draft report.
6. Goguen, J. A., and Burstall, R. M. CAT, a System for the Structured Elaboration of Correct Programs from Structured Specifications. Technical Report, SRI, International; Computer Science Lab, 1980. Based on unpublished working draft, UCLA and SRI, 1979.
7. Goguen, J. A. and Meseguer, J. Completeness of Many-sorted Equational Logic. 1981. to appear, SIGACT Newsletter.
8. Goguen, J. A. and Parsaye-Ghomi, K. Algebraic Denotational Semantics using Parameterized Abstract Modules. In J. Diaz and I. Ramos (editor), Formalizing Programming Concepts, pages 292-309. Springer-Verlag, Peniscola, Spain, 1981. Lecture Notes in Computer Science, volume 107.
9. Meseguer, J. Varieties of Chain-Complete Algebras. Journal of Pure and Applied Algebra 19:347-383, 1980.
10. Meseguer, J. A Birkhoff-like Theorem for Algebraic Classes of Interpretations of Program Schemes. In J. Diaz and I. Ramos (editor), Formalization of Programming Concepts, pages 152-168. Springer-Verlag, Peniscola, Spain, 1981. Lecture Notes in Computer Science, volume 107.
11. Parsaye-Ghomi, K. Higher Order Data Types. PhD thesis, UCLA, Computer Science Department, 1981. Forthcoming.
12. Tardo, J. The Design, Specification and Implementation of OBJT: A Language for Writing and Testing Abstract Algebraic Program Specifications. PhD thesis, UCLA, Computer Science Department, 1981.

III. RESULTS IN PROGRESS

This section sketches some results which are now in progress.

1. Implementations

One of the basic ideas behind OBJ is to regard equations as rewrite rules, so that techniques such as the Knuth-Bendix algorithm can be used for execution and verification. D. Smallberg of UCLA is working on a version of the reduction and Knuth-Bendix algorithms which can handle so-called permuting axioms. This system, called KB, is being written in C, to run on VAX machines. It will not have all the features of OBJT, such as mix-fix syntax and error handling, but it will serve as testbed for the eventual integration of these ideas in an OBJ-1 system. The design and implementation of KB is

Smallberg's Ph.D. thesis topic.

Also, J. Weiner of the University of New Hampshire has begun work on an implementation of OBJ in Prolog. This should be more efficient than the current Rutgers-UCI LISP implementation, although it will not include all its features. It will also be highly portable.

2. Foundations

We are making significant progress on the difficult problems arising in the theory of error algebras, by making use of advanced mathematical ideas in works such as [Coste 77] and [Gabriel Ulmer 71]. Some of our ideas are described in Appendices F and G. If successful, it appears that this approach will subsume all the various algebraic models which have been so far given in the literature. A major difficulty which we foresee is attempting to convey the ideas in a way which will be understandable to the computer science community at large.

APPENDICES

It is intended that the material in these appendices should provide a good practical background for OBJT users, as supplements to[Goguen & Tardo 79] and[Tardo 81].

A OBJT SUGGESTIONS, BUGS AND COMMENTS

This is a compendium of various discoveries about OBJ and its implementations. Many comments (e.g., in 1.a below) relate to the implementations at SRI-KL, namely OBJT and OBJT20, and are so indicated, while others will apply to any implementation based on the same design. Also, note that many of the OBJT bugs have been corrected in OBJT20. Eventually OBJT20 will supersede OBJT and also acquire its name.

1. Input/Output

a. There is no way to see the result of performing an IMAGE. Perhaps there could be a flag which if set would cause the result of performing an IMAGE to be displayed whenever IMAGE is executed. This flag could also control whether or not an OUT file would include these result displays.

This raises the question of whether it is desirable to display built-in objects such as INT. One possibility is to display just the syntax, with a comment (***) that this is a built-in object; a second possibility is to display a set of equations which define the object, even though the object is not actually implemented by the corresponding rewrite rules.

b. The parsers provide no help when an expression cannot be parsed. It is a difficult but interesting problem to design a parser which will provide useful feedback to the user in such cases. Probably it should be interactive.

2. IMAGE

a. Why should it be permitted to apply IMAGE to built-in objects such as INT? (One explanation is that one could then define NAT from INT; but there is a better way to do that, e.g., using a unary prefix operator, such as #_ : INT -> NAT, and a suitable error equation.)

b. We could define a new OBJ "parameterized object" to have the form SORTS (<param-sort-list>) <new-sort-list> / <old-sort-list> [with the latter optional] for its SORTS declaration, and otherwise the same syntax as present OBJT objects. Furthermore, we should allow only such parameterized objects to be imaged, with only their parameter sorts being mapped. Furthermore, if not all parameter sorts are actually mapped, then the resulting IMAGE object is also parameterized, and this should be indicated in the same way, with parentheses.

c. It should not be permitted to have duplicate copies of ANY objects around, unless they have different names for the object and for new sorts. In particular, IMAGE should not create duplicate objects, either of BOOL or of any other objects.

3. RUM & PERMUTING

a. The present implementations have a bug somewhere; for example, ARRAY with ADD as a PERMUTING operator produces strange LISP-level error messages when SYMBOLTABLE (defined by IMAGE as an ARRAY of STACKs) is run. Similar strange things happen if it is RUMed without the PERMUTING declaration.

b. EQ is not implemented for permuting objects; it should not be very difficult to extend it to do so.

c. It would be useful to be able to set the environment of a RUN to any desired object. The syntax

RUN / <obj-list> <exp> NUR

is one possibility, with default (if there is no "/ <obj-list>") to the previous object, as at present.

4. Files

a. It would be nice to be able to save OBJT working states. (Using the operating system SAVE seems to use a lot of memory.) More generally, it would be nice to have an incremental object management facility.

5. Updating Objects

a. There are many cases where one wants to enrich an existing object with some new operations, or even some new sorts, and it is a drag to have to give a new name. Even worse, one might like to take an existing large definition and evolve it to serve some new purpose by changing old objects. It would be very nice if there were some systematic way of doing this. Here is a suggestion: OBJ <id> / <id> JBO, where <id> is an identifier, would create a new object <id> which enriches the old one, now designated <id>.-1; and all old references to the old object <id> now updated to become references to <id>.-1. This process could be repeated to yield objects <id>.-2, <id>.-3 etc.

6. Associativity

OBJT and OBJT20 implement something which is fairly close to associativity, but more efficient to run; thus, it is dangerous to rely on associativity working correctly in complex cases. An associative-operator, pattern-matching algorithm could be the basis of a new implementation. It should be possible to do this in such a way that the simple cases are still executed efficiently.

7. Knuth-Bendix Test

It is far from clear how to implement a Knuth-Bendix algorithm which handles all OBJ features, including errors, conditional equations, associativity and permutativity. The current implementation however, does not always run correctly even on classical cases which involve none of these features.

8. Syntax

a. It would be convenient for some applications to be able to define operations with result a tuple of values of specified sorts (this is called "co-arity" in the algebraic literature). One also needs to be able to "untuple" such values. For this, one might build in operations, denoted say 1^* , 2^* , 3^* , ... to extract (respectively) the first, second, third, ... component.

b. Equations of the form

AS S: (X = BAD IF P(X))

which would be very useful for defining subtypes, such as NAT as a subtype of INT, are not accepted. (This could be done by setting $S = \text{NAT}$, giving a coercion $\text{INT} \rightarrow \text{NAT}$, and letting $P(X)$ be $X < 0$.)

c. (HIDDEN) does not work correctly in OBJT; the first time that an operation declared HIDDEN is used inside the object where it is declared, one gets an "abnormal termination" warning and is thrown out of the object. (This has been corrected in OBJT20.)

d. It would be nice to have a way to encapsulate a number of previously defined objects, declaring all but some subset of their operations to be HIDDEN. (See ORDINARY for one approach.)

9. Built-in Objects

a. The type NAT of natural numbers is not built-in. It should be, with the obvious coercion from NAT to INT.

b. It would be nice if the quotes (e.g., 'A', 'B, etc.) were not necessary for type ID of identifiers.

c. OBJT evaluates some Boolean expressions wrong. For example,
RUN T AND (T AND T) NUR

yields the result "T AND T". Similarly for many other truth values and connectives. (This has been corrected in OBJT20.)

10. Trivial but Annoying

a. Input/Output

Spacing conventions are irregular and sometimes confusing. (These have all been corrected in OBJT20.)

(1). Comments received from a file are treated reasonably, but comments acquired directly from the user are not in OBJT (e.g., typing

```
>*** THIS IS A  
TWO LINE TEST OF COMMENTS. ***
```

at a terminal produces a rather odd distribution of characters).

(2). For input read from a file, warnings after a RUN skip a line, and the next RUN is right after the warning. (e.g.,

```
>RUN .....
```

```
warning: .....  
>RUN
```

which gives the impression that warning goes with the 2nd RUN).

(3). The built-in OBJT editor is not so easy to use; it would probably be better to use something like EMACS.

b. Syntax

(1). OBJ should not object to using numbers as operator symbols. Of course, ambiguities might arise, but overloading constants is no worse than overloading operators like + and *. (This is corrected in OBJT20.)

(2). It is actually difficult to use keywords spelled backwards as terminators, because one makes spelling errors; syntax like

```
OBJ ... ENDOBJ  
RUN ... ENDRUN  
IM ... ENDIM
```

would be easier to remember and to use correctly.

(3). The name "TEST" for the flag which causes OBJ to interpret all RUNs as RUMs is not very suggestive; for example, "RUM" would be better.

B PROGRAMMING LANGUAGE DEFINITION

This is an improved version of the programming language definition in [Goguen & Parsaye-Ghomi 81].

[PHOTO: Recording initiated Mon 6-Jul-81 5:52PM]

Link from GOGUEN, TTY 10

TOPS-20 Command processor 3A(37)-3
End of <GOGUEN>COMAND.CMD.2
@OBJT20

***OBJT 4/17/81

>
LISP

242 msec CPU (0 msec GC), 2831 msec clock, 64 conses

*
(REALLOC 10000 10000 10000 10000 310000)
OBJT20 Running at 411067 Load 1.57 Used 0:03:40.2 in 1:22:19

(SBEGIN)

>
IN LIB MOD NI

*** HERE IS A BASIC LIBRARY OF PARAMETERIZED TYPES ***

OBJ PAIR
SORTS PAIR LEFT RIGHT
OK-OPS
 <_;> : LEFT RIGHT -> PAIR
 LEFT_ : PAIR -> LEFT
 RIGHT_ : PAIR -> RIGHT
VARS
 LEFT : LEFT
 RIGHT : RIGHT
 P : PAIR
OK-EQNS
 (LEFT < LEFT ; RIGHT > = LEFT)
 (RIGHT < LEFT ; RIGHT > = RIGHT)
 (< LEFT P ; RIGHT P > = P)
JBO

```

OBJ ARRAY / BOOL
SORTS ARRAY INDEX ELEMENT
OK-OPS
  NIL-ARRAY : -> ARRAY
  PUT : INDEX ELEMENT ARRAY -> ARRAY
  _[_] : ARRAY INDEX -> ELEMENT
  _IN_ : INDEX ARRAY -> BOOL
ERR-OPS
  UNDEF : INDEX -> ELEMENT
VARS
  A : ARRAY
  I I' : INDEX
  ELM : ELEMENT
OK-EQNS
  (PUT(I,ELM,A)[ I ] = ELM)
  (PUT(I,ELM,A)[ I' ] = A [ I' ] IF NOT I == I')
  (I IN NIL-ARRAY = F)
  (I IN PUT(I',ELM,A)= I == I' OR I IN A)
ERR-EQNS
  (A [ I ] = UNDEF(I)IF NOT I IN A)
JBO

```

```

OBJ LIST / BOOL
SORTS LIST ELEMENT
OK-OPS
  NIL : -> LIST
  _ : ELEMENT -> LIST
  _;_ : LIST LIST -> LIST (ASSOCIATIVE)
  EMPTY? : LIST -> BOOL
  FIRST : LIST -> ELEMENT
  REST : LIST -> LIST
ERR-OPS
  NO-FIRST : -> LIST
  NO-REST : -> LIST
VARS
  L : LIST
  E E' : ELEMENT
OK-EQNS
  (NIL ; L = L)
  (L ; NIL = L)
  (EMPTY?(NIL)= T)
  (EMPTY?(E)= F)
  (EMPTY?(L ; E)= F)
  (FIRST(E ; L)= E)
  (REST(E ; L)= L)
  (FIRST(E)= E)
  (REST(E)= NIL)
ERR-EQNS
  (FIRST(NIL)= NO-FIRST)
  (REST(NIL)= NO-REST)

```

JBO

```
OBJ STACK / BOOL
SORTS STACK ELEMENT
OK-OPS
  EMPTY : -> STACK
  POP_ : STACK -> STACK
  PUSH : ELEMENT STACK -> STACK
  TOP_ : STACK -> ELEMENT
  EMPTY?_ : STACK -> BOOL
ERR-OPS
  UNDERFLOW : -> STACK
  NO-TOP : -> ELEMENT
VARS
  ELM : ELEMENT
  S : STACK
OK-EQNS
  (POP PUSH(ELM,S)= S)
  (TOP PUSH(ELM,S)= ELM)
  (EMPTY? PUSH(ELM,S)= F)
  (EMPTY? EMPTY = T)
ERR-EQNS
  (POP EMPTY = UNDERFLOW)
  (TOP EMPTY = NO-TOP)
JBO
```

=End of file=

*** WE BEGIN BY DEFINING THE BASIC COMPONENTS OF STATES

*** THE STORABLE VALUES OF MODEST ARE TYPE-VALUE PAIRS
WHICH ARE CALLED ITEMS ***

```
IM (PAIR => ITEM1)
SORTS (PAIR => ITEM)
  (LEFT => TYPE)
  (RIGHT => VALUE)
OPS  (<_> : LEFT RIGHT -> PAIR => <_:>)
  (LEFT_ : PAIR -> LEFT => TYPE-OF_)
  (RIGHT_ : PAIR -> RIGHT => VALUE-OF_)
MI
```

```
OBJ ITEM / ITEM1
OK-OPS
  UNDEFINED-TYPE : -> TYPE
  UNDEFINED-VALUE : -> VALUE
  UNDEFINED-ITEM : -> ITEM
```

JBO

*** A TAPE IS A LIST OF ITEMS ***

```
IM (LIST => TAPE)/ ITEM BOOL
SORTS (LIST => TAPE)
      (ELEMENT => ITEM)
OPS   (NO-REST : -> LIST => END-OF-TAPE)
MI
```

```
IM (PAIR => I/O-TAPES)/ TAPE
SORTS (PAIR => I/O-TAPES)
      (LEFT => TAPE)
      (RIGHT => TAPE)
OPS   (LEFT_ : PAIR -> LEFT => INPUT-OF_)
      (RIGHT_ : PAIR -> RIGHT => OUTPUT-OF_)
MI
```

*** FROM THE INTEGERS, WE CONSTRUCT FIRST AN ENRICHMENT
WITH MORE OPERATIONS, AND THEN AN
ABSTRACTION, WITH FEWER ***

```
OBJ INTE / INT BOOL
OK-OPS
  _<=_ : INT INT -> BOOL
  _=>_ : INT INT -> BOOL
VARS
  I J : INT
OK-EQNS
  (I <= J = NOT(I > J))
  (I => J = NOT(I < J))
JBO
```

```
OBJ LOC / INT
SORTS LOC
OK-OPS
  _ : INT -> LOC
  NEXT_ : LOC -> LOC
VARS
  I : INT
OK-EQNS
  (NEXT I = I + 1)
JBO
```

```
IM (ARRAY => STORE)/ LOC ITEM BOOL
SORTS (ARRAY => STORE)
```

```

        (ELEMENT => ITEM)
        (INDEX => LOC)
OPS   (NIL-ARRAY : -> ARRAY => NIL-STORE)
MI

```

```

IM (PAIR => STATE)/ STORE I/O-TAPES BOOL
SORTS (PAIR => STATE)
      (LEFT => STORE)
      (RIGHT => I/O-TAPES)
OPS   (LEFT_ : PAIR -> LEFT => MEMERY-OF_)
      (RIGHT_ : PAIR -> RIGHT => TAPE-OF_)
MI

```

```

OBJ MAKE-STATE / STATE
OK-OPS
  PUT : LOC ITEM STATE -> STATE
  [_] : STATE LOC -> ITEM
  _IN_ : LOC STATE -> BOOL
  NIL-STATE : -> STATE
VARS
  ITEM : ITEM
  STORE : STORE
  L : LOC
  TAPE : I/O-TAPES
OK-EQNS
  (PUT(L,ITEM,< STORE ; TAPE >)= < PUT(L,ITEM,
    STORE); TAPE >)
  (< STORE ; TAPE > [ L ] = STORE [ L ])
  (L IN < STORE ; TAPE > = L IN STORE)
  (NIL-STATE = < NIL-STORE ; < NIL ; NIL > >)
JBO

```

```

OBJ I/O / MAKE-STATE
OK-OPS
  READ-NEXT-INPUT : STATE -> ITEM
  WRITE-NEXT-OUTPUT : ITEM STATE -> STATE
  INITIAL-STATE : TAPE -> STATE
  SET-INPUT : STATE -> STATE
VARS
  STORE : STORE
  IN-TAPE OUT-TAPE : TAPE
  STATE : STATE
  ITEM : ITEM
OK-EQNS
  (READ-NEXT-INPUT(< STORE ; < IN-TAPE ; OUT-TAPE
    > >)= FIRST(IN-TAPE))
  (WRITE-NEXT-OUTPUT(ITEM,< STORE ; < IN-TAPE ;
    OUT-TAPE > >)= < STORE ; < IN-TAPE ;(

```

```

        OUT-TAPE ; ITEM)> >)
    (INITIAL-STATE(IN-TAPE)= < NIL-STORE ; < IN-TAPE
      ; NIL > >)
    (SET-INPUT(< STORE ; < IN-TAPE ; OUT-TAPE > >)=
      < STORE ; < REST(IN-TAPE); OUT-TAPE > >)
JBO

```

OBJ ALLOCATION / I/O

OK-OPS

```

    ALLOCATE : STATE -> LOC
    INITIALIZE : TYPE STATE -> STATE
    INITIALIZE : ITEM STATE -> STATE
    FIND-NEXT : LOC STATE -> LOC

```

VARs

```

    TYPE : TYPE
    STATE : STATE
    ITEM : ITEM
    LOC : LOC

```

OK-EQNS

```

    (ALLOCATE(STATE)= FIND-NEXT(1,STATE))
    (FIND-NEXT(LOC,STATE)= IF NOT(LOC IN STATE)THEN
      LOC ELSE FIND-NEXT((NEXT LOC),STATE)FI)
    (INITIALIZE(TYPE,STATE)= PUT(ALLOCATE(STATE),<
      TYPE : UNDEFINED-VALUE >,STATE))
    (INITIALIZE(ITEM,STATE)= PUT(ALLOCATE(STATE),
      ITEM,STATE))

```

JBO

IM (ARRAY => LAYER)/ ID LOC BOOL

SORTS (ARRAY => LAYER)

(INDEX => ID)

(ELEMENT => LOC)

OPS (NIL-ARRAY : -> ARRAY => NIL-LAYER)

MI

IM (STACK => SYMBOL-TABLE)/ LAYER BOOL

SORTS (STACK => ENV)

(ELEMENT => LAYER)

OPS (EMPTY : -> STACK => NIL-ENV)

(POP_ : STACK -> STACK => EXITBLOCK_)

MI

IM (LIST => ID-LIST)/ ID BOOL

SORTS (LIST => ID-LIST)

(ELEMENT => ID)

MI

```

OBJ ENVIRONMENT / SYMBOL-TABLE ID-LIST ALLOCATION
OK-OPS
  ENTERBLOCK_ : ENV -> ENV
  GET : ENV ID -> LOC
  RETRIEVE : ID ENV STATE -> ITEM
  BIND : ID-LIST ENV STATE -> ENV
ERR-OPS
  UNDECL : ID -> LOC
  _ALREADY-DECLARED-IN-BLOCK : ID -> ENV
VARS
  ENV : ENV
  ID : ID
  LAY : LAYER
  STATE : STATE
  ID-L : ID-LIST
OK-EQNS
  (ENTERBLOCK ENV = PUSH(NIL-LAYER,ENV))
  (GET(ENV,ID)=(TOP ENV)[ ID ] IF ID IN TOP ENV)
  (GET(ENV,ID)= GET(EXITBLOCK ENV,ID)IF(NOT(ID IN
    TOP ENV)))
  (RETRIEVE(ID,ENV,STATE)= STATE [ GET(ENV,ID)])
  (BIND(ID,PUSH(LAY,ENV),STATE)= PUSH(PUT(ID,
    ALLOCATE(STATE),LAY),ENV))
  (BIND(ID ; ID-L,ENV,STATE)= BIND(ID-L,BIND(ID,
    ENV,STATE),(INITIALIZE(UNDEFINED-ITEM,STATE
    ))))
ERR-EQNS
  (GET(NIL-ENV,ID)= UNDECL(ID))
  (BIND(ID,PUSH(LAY,ENV),STATE)= ID
    ALREADY-DECLARED-IN-BLOCK IF(ID IN LAY))
JBO

```

*** WE NOW DEFINE INTEGER AND BOOLEAN EXPRESSIONS ***

```

OBJ EXPRESSION / ENVIRONMENT
SORTS EXP
OK-OPS
  _ : ID -> EXP
  VALUE : EXP ENV STATE -> ITEM
  TYPE : EXP ENV STATE -> TYPE
VARS
  EXP : EXP
  ENV : ENV
  STATE : STATE
  ID : ID
  I J : ITEM
OK-EQNS
  (VALUE(ID,ENV,STATE)= RETRIEVE(ID,ENV,STATE))
  (TYPE(EXP,ENV,STATE)= TYPE-OF VALUE(EXP,ENV,
    STATE))

```

JBO

OBJ INT-EXP / EXPRESSION

OK-OPS

```
INT : -> TYPE
_ : INT -> VALUE
INT-VAL : ITEM -> INT
_ : INT -> EXP
+_ : EXP EXP -> EXP
-_ : EXP EXP -> EXP
*_ : EXP EXP -> EXP
```

ERR-OPS

```
_DOES-NOT-MATCH_ : TYPE TYPE -> VALUE
```

VARS

```
I : INT
ENV : ENV
STATE : STATE
EXP EXP' : EXP
TYPE : TYPE
VALUE : VALUE
```

OK-EQNS

```
(INT-VAL(< INT : I >)= I)
(VALUE(I,ENV,STATE)= < INT : I >)
(VALUE(EXP + EXP',ENV,STATE)= < INT :(INT-VAL(
    VALUE(EXP,ENV,STATE))+ INT-VAL(VALUE(EXP',
    ENV,STATE))))>)
(VALUE(EXP - EXP',ENV,STATE)= < INT :(INT-VAL(
    VALUE(EXP,ENV,STATE))- INT-VAL(VALUE(EXP',
    ENV,STATE))))>)
(VALUE(EXP * EXP',ENV,STATE)= < INT :(INT-VAL(
    VALUE(EXP,ENV,STATE))* INT-VAL(VALUE(EXP',
    ENV,STATE))))>)
```

ERR-EQNS

```
(INT-VAL(< TYPE : VALUE >)= TYPE DOES-NOT-MATCH
INT IF(NOT(TYPE == INT)))
```

JBO

OBJT20 Running at 405005 Load 3.87 Used 0:04:44.9 in 1:25:16

OBJ BOOL-EXP / INTE INT-EXP

OK-OPS

```
BOOL : -> TYPE
_ : BOOL -> VALUE
BOOL-VAL : ITEM -> BOOL
_ : BOOL -> EXP
_AND_ : EXP EXP -> EXP
_OR_ : EXP EXP -> EXP
NOT_ : EXP -> EXP
_EQ_ : EXP EXP -> EXP
<=_ : EXP EXP -> EXP
>_ : EXP EXP -> EXP
```

```

ERR-OPS
  TYPE-CONFLICT : -> ITEM
VARS
  B : BOOL
  TYPE : TYPE
  ENV : ENV
  STATE : STATE
  VALUE : VALUE
  EXP EXP' : EXP
  I J : INT
OK-EQNS
  (BOOL-VAL(< BOOL : B >)= B)
  (VALUE(B,ENV,STATE)= < BOOL : B >)
  (VALUE(EXP AND EXP',ENV,STATE)= < BOOL : (
    BOOL-VAL(VALUE(EXP,ENV,STATE))AND BOOL-VAL(
    VALUE(EXP',ENV,STATE)))>)
  (VALUE(EXP OR EXP',ENV,STATE)= < BOOL : (BOOL-VAL
    (VALUE(EXP,ENV,STATE))OR BOOL-VAL(VALUE(
    EXP',ENV,STATE)))>)
  (VALUE(NOT EXP,ENV,STATE)= < BOOL : (NOT(BOOL-VAL
    (VALUE(EXP,ENV,STATE))))>)
  (VALUE(EXP EQ EXP',ENV,STATE)= < BOOL : (VALUE(
    EXP,ENV,STATE)== VALUE(EXP',ENV,STATE)))>)
  (VALUE(EXP <= EXP',ENV,STATE)= < BOOL : (INT-VAL(
    VALUE(EXP,ENV,STATE))<= INT-VAL(VALUE(EXP',
    ENV,STATE))))>)
  (VALUE(EXP > EXP',ENV,STATE)= < BOOL : (INT-VAL(
    VALUE(EXP,ENV,STATE))> INT-VAL(VALUE(EXP',
    ENV,STATE))))>)
ERR-EQNS
  (BOOL-VAL(< TYPE : VALUE >)= TYPE DOES-NOT-MATCH
    BOOL IF NOT(TYPE == BOOL))
  (VALUE(EXP EQ EXP',ENV,STATE)= TYPE-CONFLICT IF
    NOT(TYPE(EXP,ENV,STATE)== TYPE(EXP',ENV,
    STATE)))
  (VALUE(EXP <= EXP',ENV,STATE)= TYPE-CONFLICT IF
    NOT(TYPE(EXP,ENV,STATE)== INT AND TYPE(EXP',
    ENV,STATE)== INT))
  (VALUE(EXP > EXP',ENV,STATE)= TYPE-CONFLICT IF
    NOT(TYPE(EXP,ENV,STATE)== INT AND TYPE(EXP',
    ENV,STATE)== INT))
JBO

```

*** WE NOW DEFINE VARIOUS STATEMENTS AND THEIR MEANINGS

```

IM (LIST => STMT-LIST)/ BOOL
SORTS (LIST => STMT-LIST)
  (ELEMENT => STMT)
MI

```

```

OBJ EXECUTION / STMT-LIST ENVIRONMENT I/O
SORTS PROGRAM
OK-OPS
    EXECUTE_ : PROGRAM -> TAPE
    EVAL : STMT-LIST ENV STATE -> STATE
    _WITH-INPUT_ : STMT-LIST TAPE -> PROGRAM
VARS
    TAPE : TAPE
    STMT-L : STMT-LIST
OK-EQNS
    (EXECUTE(STMT-L WITH-INPUT TAPE)= OUTPUT-OF(
        TAPE-OF(EVAL(STMT-L,NIL-ENV,INITIAL-STATE(
            TAPE))))))
JBO

```

*** FIRST SEMICOLON ***

```

OBJ SEMICOLON / EXECUTION
VARS
    STMT : STMT
    STMT-L : STMT-LIST
    ENV : ENV
    STATE : STATE
OK-EQNS
    (EVAL(STMT ; STMT-L,ENV,STATE)= EVAL(STMT-L,ENV,
        EVAL(STMT,ENV,STATE)))
JBO

```

*** DEFINE BLOCK STRUCTURE ***

```

IM (LIST => DECL-LIST)/ BOOL
SORTS (LIST => DECL-LIST)
    (ELEMENT => DECLARATION)
OPS (NIL : -> LIST => NILDECL)
MI

```

```

OBJ DECLARATION / DECL-LIST BOOL-EXP
OK-OPS
    _:_ : ID TYPE -> DECLARATION
    DECLARE-ENV : DECL-LIST ENV STATE -> ENV
    DECLARE : DECL-LIST STATE ENV -> STATE
VARS
    D : DECLARATION
    DL : DECL-LIST
    ENV : ENV
    ID : ID
    TYPE : TYPE
    ID-L : ID-LIST
    STATE : STATE
OK-EQNS

```

```

(DECLARE(D ; DL,STATE,ENV)= DECLARE(DL,DECLARE(D
,STATE,ENV),DECLARE-ENV(D,ENV,STATE)))
(DECLARE-ENV(D ; DL,ENV,STATE)= DECLARE-ENV(DL,
DECLARE-ENV(D,ENV,STATE),INITIALIZE(
UNDEFINED-ITEM,STATE)))
(DECLARE(ID : TYPE,STATE,ENV)= INITIALIZE(TYPE,
STATE))
(DECLARE-ENV(ID : TYPE,ENV,STATE)= BIND(ID,ENV,
STATE))

```

JBO

OBJ BLOCK / DECLARATION EXECUTION

SORTS BLOCK

OK-OPS

```

_:=_ : DECL-LIST STMT-LIST -> BLOCK
BEGIN_END : BLOCK -> STMT

```

VARs

```

DCL-L : DECL-LIST
STMT-L : STMT-LIST
ENV : ENV
STATE : STATE

```

OK-EQNS

```

(EVAL(BEGIN DCL-L ; STMT-L END,ENV,STATE)= EVAL(
STMT-L,DECLARE-ENV(DCL-L,ENTERBLOCK ENV,
STATE),DECLARE(DCL-L,STATE,ENTERBLOCK ENV))
)

```

JBO

*** DEFINE ASSIGNMENT ***

OBJ ASSIGNMENT / EXECUTION BOOL-EXP

OK-OPS

```

_:=_ : ID EXP -> STMT
ASSIGN : ID ITEM ENV STATE -> STATE

```

ERR-OPS

```

TYPE-OF_CONFLICTS_ : ID ITEM -> STATE

```

VARs

```

ID : ID
EXP : EXP
ENV : ENV
STATE : STATE
ITEM : ITEM

```

OK-EQNS

```

(EVAL(ID := EXP,ENV,STATE)= ASSIGN(ID,VALUE(EXP,
ENV,STATE),ENV,STATE))
(ASSIGN(ID,ITEM,ENV,STATE)= PUT(GET(ENV,ID),ITEM
,STATE))

```

ERR-EQNS

```

(ASSIGN(ID,ITEM,ENV,STATE)= TYPE-OF ID CONFLICTS
ITEM IF NOT(TYPE-OF RETRIEVE(ID,ENV,STATE)
== TYPE-OF ITEM))

```

JBO

*** DEFINE READ AND PRINT STATEMENTS ***

```
OBJ INPUT-OUTPUT / EXECUTION ASSIGNMENT
OK-OPS
  READ_ : ID -> STMT
  PRINT_ : EXP -> STMT
ERR-OPS
  NO-INPUT-AVAILABLE-FOR_ : ID -> STATE
VARS
  ID : ID
  ENV : ENV
  STATE : STATE
  EXP : EXP
  STORE : STORE
  OUT-TAPE : TAPE
OK-EQNS
  (EVAL(READ ID,ENV,STATE)= ASSIGN(ID,
    READ-NEXT-INPUT(STATE),ENV,SET-INPUT(STATE)
  ))
  (EVAL(PRINT EXP,ENV,STATE)= WRITE-NEXT-OUTPUT(
    VALUE(EXP,ENV,STATE),STATE))
ERR-EQNS
  (EVAL(READ ID,ENV,< STORE ; < NIL ; OUT-TAPE > >
    )= NO-INPUT-AVAILABLE-FOR ID)
JBO
```

*** DEFINE CONDITIONAL ***

```
OBJ CONDITIONAL / EXECUTION BOOL-EXP STMT-LIST
OK-OPS
  IF:_THEN_ELSE_:FI : EXP STMT-LIST STMT-LIST ->
    STMT
VARS
  EXP : EXP
  STMT-L STMT-L' : STMT-LIST
  ENV : ENV
  STATE : STATE
OK-EQNS
  (EVAL(IF: EXP THEN STMT-L ELSE STMT-L' :FI,ENV,
    STATE)= EVAL(STMT-L,ENV,STATE)IF BOOL-VAL(
    VALUE(EXP,ENV,STATE))== T)
  (= EVAL(STMT-L',ENV,STATE)IF BOOL-VAL(VALUE(EXP,
    ENV,STATE))== F)
JBO
```

*** DEFINE ITERATION ***

```
OBJ ITERATION / EXECUTION BOOL-EXP STMT-LIST
OK-OPS
```

```

    WHILE_DO_OD : EXP STMT-LIST -> STMT
VARS
    EXP : EXP
    STMT-L : STMT-LIST
    ENV : ENV
    STATE : STATE
OK-EQNS
    (EVAL(WHILE EXP DO STMT-L OD,ENV,STATE)= EVAL(
        STMT-L ; WHILE EXP DO STMT-L OD,ENV,STATE)
        IF BOOL-VAL(VALUE(EXP,ENV,STATE))= T)
    (= STATE IF(BOOL-VAL(VALUE(EXP,ENV,STATE))= F))
JBO

```

*** THIS OBJECT SUMMARIZES ALL STATEMENT DEFINITIONS ***

```

OBJ STATEMENTS / EXECUTION SEMICOLON BLOCK
    ASSIGNMENT CONDITIONAL INPUT-OUTPUT
    ITERATION
JBO

```

*** WE NOW BEGIN DEFINING PROCEDURES ***

```

IM (PAIR => PROC-DEFN)/ STMT-LIST ID-LIST BOOL
SORTS (PAIR => PROC-DEFN)
    (LEFT => ID-LIST)
    (RIGHT => STMT-LIST)
OPS
    (LEFT_ : PAIR -> LEFT => FORMALS-OF_)
    (RIGHT_ : PAIR -> RIGHT => STMT-OF_)
MI

```

```

IM (PAIR => CONTOUR)/ PROC-DEFN ENVIRONMENT BOOL
SORTS (PAIR => CONTOUR)
    (LEFT => PROC-DEFN)
    (RIGHT => ENV)
OPS
    (LEFT_ : PAIR -> LEFT => DEFN-OF_)
    (RIGHT_ : PAIR -> RIGHT => ENV-OF_)
MI

```

```

IM (LIST => PARAM-DECL-LIST)/ BOOL
SORTS (LIST => PARAM-DECL-LIST)
    (ELEMENT => PARAM-DECL)
MI

```

```

IM (LIST => TYPE-LIST)/ ITEM BOOL
SORTS (LIST => TYPE-LIST)
    (ELEMENT => TYPE)

```

OPS (NIL : -> LIST => VOID)
MI

IM (LIST => EXP-LIST)/ BOOL-EXP BOOL
SORTS (LIST => EXP-LIST)
(ELEMENT => EXP)
MI

OBJ PROCEDURES / CONTOUR TYPE-LIST EXP-LIST
PARAM-DECL-LIST DECLARATION

SORTS PROC-DECL

OK-OPS

PROC[_] : TYPE-LIST -> TYPE
_ : CONTOUR -> VALUE
_ : PROC-DECL -> DECLARATION
CONTOUR-VAL : ITEM -> CONTOUR
PROC[_]_END : ID PARAM-DECL-LIST STMT-LIST ->
PROC-DECL
: : ID TYPE -> PARAM-DECL
CALL[_] : ID EXP-LIST -> STMT

VARS

TYPE-L : TYPE-LIST
C : CONTOUR
TYPE : TYPE
VALUE : VALUE

OK-EQNS

(CONTOUR-VAL(< PROC[TYPE-L] : C >)= C)

ERR-EQNS

(CONTOUR-VAL(< TYPE : VALUE >)= TYPE
DOES-NOT-MATCH PROC[VOID] IF((TYPE == INT
)OR(TYPE == BOOL)))

JBO

OBJ PROC-DECLARATION / PROCEDURES

OK-OPS

GET-ID : PARAM-DECL-LIST -> ID-LIST
GET-TYPE : PARAM-DECL-LIST -> TYPE-LIST

VARS

ID : ID
PM-L : PARAM-DECL-LIST
STMT-L : STMT-LIST
ENV : ENV
STATE : STATE
PM : PARAM-DECL
TYPE : TYPE

OK-EQNS

(DECLARE-ENV(PROC ID [PM-L] STMT-L END,ENV,
STATE)= BIND(ID,ENV,STATE))

```

(DECLARE(PROC ID [ PM-L ] STMT-L END, STATE, ENV)=
  INITIALIZE(((PROC[ GET-TYPE(PM-L)]):((
    GET-ID(PM-L); STMT-L >); DECLARE-ENV(PROC
    ID [ PM-L ] STMT-L END, ENV, STATE)>>)), STATE
  ))
(GET-TYPE(PM ; PM-L)= GET-TYPE(PM); GET-TYPE(
  PM-L))
(GET-TYPE(ID : TYPE)= TYPE)
(GET-ID(PM ; PM-L)= GET-ID(PM); GET-ID(PM-L))
(GET-ID(ID : TYPE)= ID)
JBO

```

OBJ PARAM-PASS-BY-VALUE / PROC-DECLARATION

OK-OPS

```

PASS-ENV : ID ENV STATE -> ENV
PASS : EXP-LIST STATE ENV -> STATE
GET-ENV : ID ENV STATE -> ENV
GET-PARAMS : ID ENV STATE -> ID-LIST

```

VARs

```

EXP : EXP
EXP-L : EXP-LIST
ID : ID
STMT-L : STMT-LIST
STATE : STATE
ENV : ENV

```

OK-EQNS

```

(PASS(EXP ; EXP-L, STATE, ENV)= PASS(EXP-L, PASS(
  EXP, STATE, ENV), ENV))
(PASS(EXP, STATE, ENV)= INITIALIZE(VALUE(EXP, ENV,
  STATE), STATE))
(PASS-ENV(ID, ENV, STATE)= BIND(GET-PARAMS(ID, ENV,
  STATE), GET-ENV(ID, ENV, STATE), STATE))
(GET-PARAMS(ID, ENV, STATE)= FORMALS-OF(DEFN-OF
  CONTOUR-VAL(VALUE(ID, ENV, STATE))))
(GET-ENV(ID, ENV, STATE)= ENTERBLOCK ENV-OF
  CONTOUR-VAL(VALUE(ID, ENV, STATE)))

```

JBO

OBJ CALL-BY-VALUE / PARAM-PASS-BY-VALUE EXECUTION

OK-OPS

```

CALL-OK? : ID EXP-LIST ENV STATE -> BOOL
LIST-TYPE : EXP-LIST ENV STATE -> TYPE-LIST
GET-STMT : ID ENV STATE -> STMT-LIST

```

ERR-OPS

```

PARAMS-OF_MISMATCH : ID EXP-LIST -> BOOL

```

VARs

```

ID : ID
EXP-L : EXP-LIST
EXP : EXP
ENV : ENV

```

```

STATE : STATE
OK-EQNS
  (EVAL(CALL ID [ EXP-L ],ENV,STATE)= EVAL(
    GET-STMT(ID,ENV,STATE),PASS-ENV(ID,ENV,
      STATE),PASS(EXP-L,STATE,ENV))IF CALL-OK?(ID
        ,EXP-L,ENV,STATE))
  (CALL-OK?(ID,EXP-L,ENV,STATE)= T IF(TYPE(ID,ENV,
    STATE)== PROC[ LIST-TYPE(EXP-L,ENV,STATE)])
    )
  (GET-STMT(ID,ENV,STATE)= STMT-OF DEFN-OF
    CONTOUR-VAL(VALUE(ID,ENV,STATE)))
  (LIST-TYPE(EXP,ENV,STATE)= TYPE(EXP,ENV,STATE))
  (LIST-TYPE(EXP ; EXP-L,ENV,STATE)= TYPE(EXP,ENV,
    STATE); LIST-TYPE(EXP-L,ENV,STATE))
ERR-EQNS
  (CALL-OK?(ID,EXP-L,ENV,STATE)= PARAMS-OF ID
    MISMATCH EXP-L IF(NOT(TYPE(ID,ENV,STATE)==
      PROC[ LIST-TYPE(EXP-L,ENV,STATE)])))
JBO

```

*** WE NOW SUM UP ALL THE FEATURES OF MODEST ***

```

OBJ MODEST / STATEMENTS CALL-BY-VALUE
JBO

```

=End of file=

EXIT

```

@
@;WE NOW SAVE THE RESULTS OF OBJT20'S PROCESSING OF THIS
@;DEFINITION IN A FILE TO WHICH WE CAN RETURN LATER FOR
@;EXECUTION. IN EFFECT, WE HAVE COMPILED AN INTERPRETER
@;FOR THE LANGUAGE MODEST.
@

```

```

@SAVE MOD.EXE
MOD.EXE.3 Saved
@

```

```

@VDI MOD.EXE

```

```

PS:<OBJT>
MOD.EXE.3;P775200      268 137216(36)  6-Jul-81 18:02:16 GOGUEN

```

```

@;THUS FILE MOD.EXE IS IN DIRECTORY <OBJT> , 268 PAGES LONG

```

```

@R MOD
[CHKPOINT:(7 6 81)AT(18 2 10)]

```

IN MOTEST NI

*** TEST PROGRAMS FOR THE MODEST DEFINITION ***
 *** FIRST TESTS FOR CONDITIONAL ***

```
RUN EXECUTE((BEGIN 'A : INT ; READ 'A ;(IF: 'A <= 4 THEN
    PRINT('A + 1)ELSE PRINT('A * 2):FI)END)
    WITH-INPUT(< INT : 2 >)) NUR
AS TAPE: (< INT : 3 >)
```

```
RUN EXECUTE((BEGIN 'A : INT ; READ 'A ;(IF: 'A <= 4 THEN
    PRINT('A + 1)ELSE PRINT('A * 2):FI)END)
    WITH-INPUT(< INT : 5 >)) NUR
AS TAPE: (< INT : 10 >)
```

*** A TEST FOR WHILE ***

```
RUN EXECUTE((BEGIN 'A : INT ; 'S : INT ; READ 'A ; 'S :=
    0 ; WHILE('A > 0)DO('S :=('A + 'S); 'A :=(
    'A - 1); PRINT 'S)OD END)WITH-INPUT(< INT :
    2 >)) NUR
AS TAPE: ((< INT : 2 >);(< INT : 3 >))
```

*** A TEST FOR BLOCKS ***

```
RUN EXECUTE((BEGIN 'A : INT ; 'B : INT ;(READ 'A ; 'B :=(
    'A + 11); PRINT 'B);(BEGIN 'A : INT ; READ
    'A ; 'B :=('A + 5); PRINT 'B END); 'B :=('A
    + 22); PRINT 'B END)WITH-INPUT(< INT : 11
    > ; < INT : 5 >)) NUR
AS TAPE: ((< INT : 22 >);(< INT : 10 >);(< INT : 33 >))
```

*** TESTS FOR RECURSION, FIRST FACTORIAL ***

```
RUN EXECUTE((BEGIN 'S : INT ; 'M : INT ; PROC 'P [ 'A :
    INT ]('S :=('S * 'A))END ; 'S := 1 ; READ
    'M ; WHILE('M > 1)DO(CALL 'P [ 'M ] ; 'M :=
    ('M - 1))OD ; PRINT 'S END)WITH-INPUT(< INT
    : 2 >)) NUR
AS TAPE: (< INT : 2 >)
```

```
RUN EXECUTE((BEGIN('A : INT ;(PROC 'P [ 'B : INT ](IF:('B
    <= 3)THEN(CALL 'P [ 'B + 1 ])ELSE(PRINT('B
    + 1)):FI)END));(READ 'A ; CALL 'P [ 'A ])
    END)WITH-INPUT < INT : 1 >) NUR
AS TAPE: (< INT : 15 >)
```

*** TESTS FOR PROCEDURES AS PARAMETERS ***

```
RUN EXECUTE((BEGIN('A : INT ; 'Q : PROC[ INT ] ;(PROC 'S
    [ 'C : INT ](PRINT 'C)END);(PROC 'P [ 'R :
    PROC[ INT ] ; 'D : INT ](CALL 'R [ 'D ])END
    ));(READ 'A ; 'Q := 'S ;(CALL 'P [ 'Q ; 'A
    ]))END)WITH-INPUT(< INT : 22 >)) NUR
AS TAPE: (< INT : 22 >)
```

```
RUN EXECUTE((BEGIN('A : INT ; 'Q : PROC[(PROC[ INT ] ;
    INT ; INT)] ;(PROC 'P [( 'R : PROC[ INT ] ;
    'B : INT ; 'C : INT)](CALL 'R [( 'B + 'C)])
    END);(PROC 'S [ 'A : INT ](PRINT('A + 11))
    END));('Q := 'P ; READ 'A ;(CALL 'Q [ 'S ;
    'A ; 'A ]))END)WITH-INPUT(< INT : 11 >))
    NUR
AS TAPE: (< INT : 33 >)
```

=End of file=

EXIT
@
@POP

[PHOTO: Recording terminated Mon 6-Jul-81 6:18PM]

C SYMBOLTREE SPECIFICATION

[PHOTO: Recording initiated Tue 9-Dec-80 11:32PM]

Link from OBJT, TTY 31

TOPS-20 Command processor 3A(30)-3
@OBJT

***OBJT 12/2/79

IN NAT PBOJS INDEX TREE STREE NI

*** NAT.OBJ : DEFINES NATURAL NUMBER FROM INTEGER USING
CONSTRUCTOR OPERATION (# N) AND ERROR CONDITION (N < 0
) ***

```
OBJ NAT / INT
SORTS NAT
OK-OPS
  #_ : INT -> NAT
  INC : NAT -> NAT
  DEC : NAT -> NAT
  POS : NAT -> BOOL
ERR-OPS
  NEG : -> NAT
VARS
  N : INT
EQNS
  (DEC(# N)= # DEC(N))
OK-EQNS
  (INC(# N)= # INC(N))
  (POS(# N)= N > 0)
ERR-EQNS
  (# N = NEG IF N < 0)
JBO
```

*** TEST CASES FOR NAT ***
RUN INC(INC(# 4)) NUR
AS NAT: (# 6)

RUN DEC(DEC(# 1)) NUR
AS NAT: >>ERROR>> NEG

RUN INC(DEC(DEC(# 1))) NUR
AS NAT: >>ERROR>> INC(NEG)

RUN DEC(DEC(DEC(# 1))) NUR

AS NAT: >>ERROR>> DEC(NEG)

RUN DEC(INC(DEC(# 1))) NUR
AS NAT: (# 0)

=End of file=

*** POBJS.OBJ : DEFINES PARAMETERIZED OBJECTS ARRAY AND
PAIR ***

OBJ ARRAY / BOOL
SORTS ARRAY INDEX ELEM / BOOL
OK-OPS
 NILARRAY : -> ARRAY
 PUT : INDEX ELEM ARRAY -> ARRAY
 [_] : ARRAY INDEX -> ELEM
 IN : INDEX ARRAY -> BOOL
ERR-OPS
 UNDEF : INDEX -> ELEM
VARS
 A : ARRAY
 I I' : INDEX
 V V' : ELEM
OK-EQNS
 (PUT(I,V,A)[I'] = V IF I == I')
 (= A [I'] IF NOT I == I')
 (I IN NILARRAY = F)
 (I IN PUT(I',V,A) = I == I' OR I IN A)
ERR-EQNS
 (A [I] = UNDEF(I) IF NOT I IN A)
JBO

OBJ PAIR
SORTS PAIR C1 C2
OK-OPS
 <_> : C1 C2 -> PAIR
 #1_ : PAIR -> C1
 #2_ : PAIR -> C2
VARS
 C1 : C1
 C2 : C2
 P : PAIR
OK-EQNS
 (#1 < C1 ; C2 > = C1)
 (#2 < C1 ; C2 > = C2)
 (< #1 P ; #2 P > = P)
JBO

=End of file=

*** INDEX.OBJ : DEFINES INDEX = POINTER TO A NODE IN A
TREE ***

```

OBJ INDEX / NAT
SORTS INDEX
OK-OPS
  INDEX : -> INDEX
  POP_ : INDEX -> INDEX
  _ : INDEX NAT -> INDEX
  NEXT_ : INDEX -> INDEX
  PREV_ : INDEX -> INDEX
ERR-OPS
  UNDEF : -> INDEX
  NO-PREV : -> INDEX
VARS
  P : INDEX
  N : NAT
OK-EQNS
  (POP(P . N)= P)
  (NEXT(P . N)= P . INC(N))
  (PREV(P . N)= P . DEC(N))
ERR-EQNS
  (POP INDEX = UNDEF)
  (P . N = NO-PREV IF ERR(N))
JBO

```

*** TEST CASES FOR INDEX ***

```

OBJ INDEXTEST / INDEX
OK-OPS
  P1 : -> INDEX
EQNS
  (P1 = INDEX . # 1 . # 2 . # 3)
JBO

```

```

RUN P1 NUR
AS INDEX: (((INDEX .(# 1)).(# 2)).(# 3))

```

```

RUN POP P1 NUR
AS INDEX: ((INDEX .(# 1)).(# 2))

```

```

RUN POP POP P1 NUR
AS INDEX: (INDEX .(# 1))

```

```

RUN POP POP POP P1 NUR
AS INDEX: INDEX

```

```

RUN POP POP POP POP P1 NUR

```

AS INDEX: >>ERROR>> UNDEF

RUN NEXT P1 NUR
AS INDEX: (((INDEX .(# 1)).(# 2)).(# 4))

RUN PREV P1 NUR
AS INDEX: (((INDEX .(# 1)).(# 2)).(# 2))

=End of file=

*** TREE.OBJ : DEFINES LABELED TREE AS INDEXED ARRAY ,
WITH LABEL AS A PARAMETER ***

IM (ARRAY => LTREE1)/ INDEX
SORTS (ARRAY => LTREE)
 (ELEM => LABEL)
OPS (NILARRAY : -> ARRAY => NILTREE)
MI

OBJ LTREE / LTREE1
OK-OPS

ARITY : INDEX LTREE -> INT
BREADTH : INDEX LTREE -> INT
FIRST : INDEX LTREE -> INDEX
LAST : INDEX LTREE -> INDEX
PUSH : INDEX LTREE -> INDEX
PUSH : INDEX LABEL LTREE -> LTREE

ERR-OPS

BAD-INDEX : -> INT
BAD-INDEX : INDEX -> INDEX
BAD-INDEX : INDEX -> LTREE

VARs

T : LTREE
P : INDEX
N : NAT
L : LABEL
I : INT

OK-EQNS

(BREADTH(P . # 1,T)= 0 IF NOT(P . # 1)IN T)
(BREADTH(P . N,T)= BREADTH(P . INC(N),T)IF(P .
 INC(N))IN T)
(BREADTH(P . # I,T)= I IF(P . # I)IN T AND NOT(P
 . # INC(I))IN T)
(FIRST(P,T)= P . # 1 IF(P . # 1)IN T)
(ARITY(P,T)= BREADTH(P . # 1,T))
(LAST(P,T)= P . # ARITY(P,T))
(PUSH(P,T)= P . # INC(ARITY(P,T)))

```

(PUSH(P,L,T)= PUT(PUSH(P,T),L,T)IF P IN T)
ERR-EQNS
(BREADTH(P . # 0,T)= BAD-INDEX)
(BREADTH(P . N,T)= BAD-INDEX IF NOT P IN T OR
  ERR(N))
(FIRST(P,T)= BAD-INDEX(P)IF NOT(P . # 1)IN T)
(LAST(P,T)= BAD-INDEX(P)IF NOT(P . # ARITY(P,T))
  IN T)
(PUSH(P,L,T)= BAD-INDEX(P)IF NOT P IN T)
JBO

```

*** INSTANTIATE LABEL TO INTEGER IN LTREE AND TEST ***

```

IM (LTREE => INTREE)/ INDEX
SORTS (LABEL => INT)
MI

```

```

OBJ LTREETEST / INTREE
OK-OPS
  P_ : INT -> INDEX
  T_ : INT -> LTREE
EQNS
  (P 0 = INDEX)
  (T 0 = PUT(P 0,0,NILTREE))
  (P 1 = P 0 . # 1)
  (T 1 = PUT(P 1,1,T 0))
  (P 2 = NEXT P 1)
  (T 2 = PUT(P 2,2,T 1))
  (P 3 = P 2 . # 1)
  (T 3 = PUT(P 3,3,T 2))
  (P 4 = NEXT P 3)
  (T 4 = PUT(P 4,4,T 3))
  (P 5 = NEXT P 4)
  (T 5 = PUT(P 5,5,T 4))
  (T 6 = PUSH(P 0,6,T 5))
  (P 6 = PUSH(P 0,T 5))
  (T 7 = PUSH(P 2,7,T 5))
  (P 7 = PUSH(P 2,T 5))
  (T 8 = PUSH(P 3,8,T 5))
  (P 8 = PUSH(P 3,T 5))
  (T 9 = PUSH(P 5,9,T 5))
  (P 9 = PUSH(P 5,T 5))
JBO

```

```

RUN ARITY(P 0,T 5) NUR
AS INT: 2

```

```

RUN ARITY(P 2,T 5) NUR
AS INT: 3

```

```

(PUSH(P,L,T)= PUT(PUSH(P,T),L,T)IF P IN T)
ERR-EQNS
(BREADTH(P . # 0,T)= BAD-INDEX)
(BREADTH(P . N,T)= BAD-INDEX IF NOT P IN T OR
ERR(N))
(FIRST(P,T)= BAD-INDEX(P)IF NOT(P . # 1)IN T)
(LAST(P,T)= BAD-INDEX(P)IF NOT(P . # ARITY(P,T))
IN T)
(PUSH(P,L,T)= BAD-INDEX(P)IF NOT P IN T)
JBO

```

*** INSTANTIATE LABEL TO INTEGER IN LTREE AND TEST ***

```

IM (LTREE => INTREE)/ INDEX
SORTS (LABEL => INT)
MI

```

```

OBJ LTREETEST / INTREE
OK-OPS
P_ : INT -> INDEX
T_ : INT -> LTREE
EQNS
(P 0 = INDEX)
(T 0 = PUT(P 0,0,NILTREE))
(P 1 = P 0 . # 1)
(T 1 = PUT(P 1,1,T 0))
(P 2 = NEXT P 1)
(T 2 = PUT(P 2,2,T 1))
(P 3 = P 2 . # 1)
(T 3 = PUT(P 3,3,T 2))
(P 4 = NEXT P 3)
(T 4 = PUT(P 4,4,T 3))
(P 5 = NEXT P 4)
(T 5 = PUT(P 5,5,T 4))
(T 6 = PUSH(P 0,6,T 5))
(P 6 = PUSH(P 0,T 5))
(T 7 = PUSH(P 2,7,T 5))
(P 7 = PUSH(P 2,T 5))
(T 8 = PUSH(P 3,8,T 5))
(P 8 = PUSH(P 3,T 5))
(T 9 = PUSH(P 5,9,T 5))
(P 9 = PUSH(P 5,T 5))
JBO

```

```

RUN ARITY(P 0,T 5) NUR
AS INT: 2

```

```

RUN ARITY(P 2,T 5) NUR
AS INT: 3

```

RUN ARITY(P 3,T 5) NUR
AS INT: 0

RUN ARITY(P 4,T 5) NUR
AS INT: 0

RUN ARITY(P 5,T 5) NUR
AS INT: 0

RUN T 5 [P 0] NUR
AS INT: 0

RUN T 5 [P 1] NUR
AS INT: 1

RUN T 5 [P 2] NUR
AS INT: 2

RUN T 5 [P 3] NUR
AS INT: 3

RUN T 5 [P 4] NUR
AS INT: 4

RUN T 5 [P 5] NUR
AS INT: 5

RUN T 9 [P 0] NUR
AS INT: 0

RUN T 9 [P 1] NUR
AS INT: 1

RUN T 9 [P 2] NUR
AS INT: 2

RUN T 9 [P 3] NUR
AS INT: 3

RUN T 9 [P 4] NUR
AS INT: 4

RUN T 9 [P 5] NUR
AS INT: 5

RUN T 9 [P 6] NUR
AS INT: >>ERROR>> UNDEF((INDEX .(# 3)))

RUN T 9 [P 7] NUR
AS INT: >>ERROR>> UNDEF(((INDEX .(# 2)).(# 4)))

RUN T 9 [P 8] NUR
AS INT: >>ERROR>> UNDEF((((INDEX .(# 2)).(# 1)).(# 1)))

RUN T 9 [P 9] NUR
AS INT: 9

=End of file=

*** STREE.OBJ : DEFINES SYMBOLTREE , PARAMETERIZED BY
LABEL ***

IM (PAIR => STREE1)/ LTREE
SORTS (PAIR => STREE)
 (C1 => INDEX)
 (C2 => LTREE)
MI

OBJ STREE / STREE1
OK-OPS

NILSTREE : -> STREE
PUSH : STREE LABEL -> STREE
LABEL : STREE LABEL -> STREE
VAL_ : STREE -> LABEL
POP_ : STREE -> STREE
NEXT_ : STREE -> STREE
PREV_ : STREE -> STREE

VARs

ST : STREE
L : LABEL

OK-EQNS

(NILSTREE = < INDEX ; NILTREE >)
(PUSH(ST,L)= < #1 ST . # INC(ARITY(#1 ST,#2 ST))

```

; PUSH(#1 ST,L,#2 ST)>>
(LABEL(ST,L)= < #1 ST ; PUT(#1 ST,L,#2 ST)>>)
(VAL ST = #2 ST [ #1 ST ])
(POP ST = < POP #1 ST ; #2 ST >)
(NEXT ST = < NEXT #1 ST ; #2 ST >)
(PREV ST = < PREV #1 ST ; #2 ST >)
JBO

```

*** INSTANTIATE LABEL TO INT AND TEST ***

```

IM (STREE => INTSTREE)/ STREE1
SORTS (LABEL => INT)
MI

```

```

OBJ STREETEST / INTSTREE
OK-OPS
  T_ : INT -> STREE
EQNS
  (T 0 = LABEL(NILSTREE,0))
  (T 1 = PUSH(T 0,1))
  (T 2 = PUSH(POP T 1,2))
  (T 3 = PUSH(POP T 2,3))
  (T 4 = PUSH(POP T 3,4))
  (T 5 = PUSH(T 4,5))
  (T 6 = PUSH(POP T 5,6))
  (T 7 = POP T 6)

```

JBO

```

RUN VAL T 0 NUR
AS INT: 0

```

```

RUN VAL T 1 NUR
AS INT: 1

```

```

RUN VAL T 2 NUR
AS INT: 2

```

```

RUN VAL T 3 NUR
AS INT: 3

```

```

RUN VAL T 4 NUR
AS INT: 4

```

```

RUN VAL T 5 NUR
AS INT: 5

```

RUN VAL T 6 NUR
AS INT: 6

RUN VAL T 7 NUR
AS INT: 4

RUN VAL PREV T 7 NUR
AS INT: 3

RUN VAL PREV T 6 NUR
AS INT: 5

RUN VAL PREV T 4 NUR
AS INT: 3

=End of file=

>
>
EXIT
e

@POP

[PHOTO: Recording terminated Tue 9-Dec-80 11:45PM]

D SPECIFICATION OF GRAPHS AND PATHS

[PHOTO: Recording initiated Mon 6-Jul-81 5:04PM]

Link from GOGUEN, TTY 10

TOPS-20 Command processor 3A(37)-3

@OBJT20

***OBJT 4/17/81

>

IN GRAPH NI

OBJT20 Running at 404602 Load 2.23 Used 0:02:25.8 in 0:34:03

*** DEFINITIONS OF GRAPH AND PATH, TO BE LATER SPECIALIZED
TO PARTICULAR GRAPHS ***

OBJ GRAPH
SORTS NODE ARC
OK-OPS
 BEGIN_ : ARC -> NODE
 END_ : ARC -> NODE
JBO

OBJ PATH / GRAPH BOOL
SORTS PATH
OK-OPS
 SOURCE_ : PATH -> NODE
 TARGET_ : PATH -> NODE
 NIL : NODE -> PATH
 APPEND : PATH ARC -> PATH
ERR-OPS
 ERRORPATH : NODE NODE -> PATH
VARS
 N N' : NODE
 P : PATH
 A : ARC
OK-EQNS
 (SOURCE NIL(N)= N)
 (TARGET NIL(N)= N)
 (SOURCE APPEND(P,A)= SOURCE P)
 (TARGET APPEND(P,A)= END A)
ERR-EQNS
 (APPEND(P,A)= ERRORPATH(SOURCE P,END A) IF NOT

TARGET P == BEGIN A)

JBO

OBJ GRAPH1 / PATH

OK-OPS

N1 : -> NODE

N2 : -> NODE

N3 : -> NODE

N4 : -> NODE

A1 : -> ARC

A2 : -> ARC

A3 : -> ARC

OK-EQNS

(BEGIN A1 = N1)

(BEGIN A2 = N2)

(BEGIN A3 = N4)

(END A1 = N2)

(END A2 = N3)

(END A3 = N3)

JBO

*** NOW SOME TEST CASES ***

RUN APPEND(NIL(N1),A1) NUR
AS PATH: APPEND(NIL(N1),A1)

RUN APPEND(APPEND(NIL(N1),A1),A2) NUR
AS PATH: APPEND(APPEND(NIL(N1),A1),A2)

RUN APPEND(APPEND(APPEND(NIL(N1),A1),A2),A3) NUR
AS PATH: >>ERROR>> ERRORPATH(N1,N3)

RUN APPEND(NIL(N1),A2) NUR
AS PATH: >>ERROR>> ERRORPATH(N1,N3)

RUN APPEND(APPEND(NIL(N1),A1),A3) NUR
AS PATH: >>ERROR>> ERRORPATH(N1,N3)

*** WE NOW DEFINE ANOTHER GRAPH ***

OBJ GRAPH2 / PATH

OK-OPS

```

N1 : -> NODE
N2 : -> NODE
N3 : -> NODE
N4 : -> NODE
A1 : -> ARC
A2 : -> ARC
A3 : -> ARC
A4 : -> ARC

```

OK-EQNS

```

(BEGIN A1 = N2)
(BEGIN A2 = N2)
(BEGIN A3 = N3)
(BEGIN A3 = N1)
(END A1 = N1)
(END A2 = N3)
(END A3 = N4)
(END A3 = N4)
(END A4 = N1)

```

JBO

*** NOW SOME TEST CASES FOR THE SECOND GRAPH ***

```

RUN APPEND(NIL(N1),A1) NUR
AS PATH: >>ERROR>> ERRORPATH(N1,N1)

```

```

RUN APPEND(NIL(N2),A1) NUR
AS PATH: APPEND(NIL(N2),A1)

```

```

RUN APPEND(APPEND(NIL(N1),A1),A2) NUR
AS PATH: >>ERROR>> ERRORPATH((SOURCE ERRORPATH(N1,N1)),N3
)

```

```

RUN APPEND(APPEND(NIL(N2),A1),A4) NUR
AS PATH: >>ERROR>> ERRORPATH(N2,N1)

```

```

RUN APPEND(APPEND(APPEND(NIL(N2),A1),A4),A3) NUR
AS PATH: >>ERROR>> ERRORPATH((SOURCE ERRORPATH(N2,N1)),N4
)

```

```

RUN APPEND(NIL(N1),A2) NUR
AS PATH: >>ERROR>> ERRORPATH(N1,N3)

```

RUN APPEND(APPEND(NIL(N1),A4),A3) NUR
AS PATH: >>ERROR>> ERRORPATH((SOURCE ERRORPATH(N1,N1)),N4
)

=End of file=

>
EXIT
@
@POP

[PHOTO: Recording terminated Mon 6-Jul-81 5:05PM]

E TECHNIQUES FOR HIGHER ORDER SPECIFICATIONS AND OTHER SURPRISES

HERE ARE SOME INTERESTING THINGS YOU MIGHT NOT HAVE REALIZED
COULD BE DONE WITH OBJT: DEFINE THE NATURALS FROM THE INTEGERS;
GET THE EFFECT OF HIGHER ORDER PARAMETERIZED TYPES; AND DEFINE
PARAMETERIZED TYPES WHICH DO NOT PRESERVE THEIR ARGUMENT OBJECTS,
BUT RATHER DIVIDE THEM BY 5.

[PHOTO: Recording initiated Tue 9-Dec-80 11:32PM]

Link from OBJT, TTY 31

TOPS-20 Command processor 3A(30)-3
@OBJT

***OBJT 12/2/79

IN NAT POBJS INDEX TREE STREE NI

*** NAT.OBJ : DEFINES NATURAL NUMBER FROM INTEGER USING
CONSTRUCTOR OPERATION (# N) AND ERROR CONDITION (N < 0
) ***

```
OBJ NAT / INT
SORTS NAT
OK-OPS
  #_ : INT -> NAT
  INC : NAT -> NAT
  DEC : NAT -> NAT
  POS : NAT -> BOOL
ERR-OPS
  NEG : -> NAT
VARS
  N : INT
EQNS
  (DEC(# N)= # DEC(N))
OK-EQNS
  (INC(# N)= # INC(N))
  (POS(# N)= N > 0)
ERR-EQNS
  (# N = NEG IF N < 0)
JBO
```

*** TEST CASES FOR NAT ***
RUN INC(INC(# 4)) NUR
AS NAT: (# 6)

RUN DEC(DEC(# 1)) NUR

AS NAT: >>ERROR>> NEG

RUN INC(DEC(DEC(# 1))) NUR
AS NAT: >>ERROR>> INC(NEG)

RUN DEC(DEC(DEC(# 1))) NUR
AS NAT: >>ERROR>> DEC(NEG)

RUN DEC(INC(DEC(# 1))) NUR
AS NAT: (# 0)

=End of file=

[PHOTO: Recording initiated Mon 13-Apr-81 6:57PM]

Link from OBJT, TTY 1

TOPS-20 Command processor 3A(32)-3
End of COMAND.CMD.1
@OBJT

***OBJT 12/2/79

>IN MAPLIST.OBJ NI

*** MAPLIST AS PARAMETERIZED OBJECT ***

OBJ MAPLIST

SORTS E L

OK-OPS

_ : E -> L

NIL : -> L

_. : L L -> L (ASSOCIATIVE)

FE : E -> E

FL : L -> L

VARs

L L' : L

E : E

OK-EQNS

(NIL . L = L)

(L . NIL = L)

(FL(E)= FE(E))

(FL(NIL)= NIL)

(FL(E . L)= FE(E). FL(L))

JBO

*** WE NOW INSTANTIATE IT TO A SQUARING FUNCTION ON LISTS
, BUT MUST FIRST DEFINE THE SQUARING FUNCTION ***

```
OBJ INTSQ / INT
OK-OPS
  SQ : INT -> INT
VARS
  N : INT
OK-EQNS
  (SQ(N)= N * N)
JBO
```

```
IM (MAPLIST => NATLISTSQ)/ INTSQ
SORTS (E => INT)
      (L => INTLIST)
OPS   (FE : E -> E => SQ)
MI
```

```
RUN FL(1 . 2 . 3) NUR
AS INTLIST: (1 . 4 . 9)
```

```
RUN FL(3 . 6 . 17) NUR
AS INTLIST: (9 . 36 . 289)
```

*** WE NOW INSTANTIATE THIS TO AN ADD1 FUNCTION ON LISTS

```
IM (MAPLIST => INTLISTINC)/ INT
SORTS (E => INT)
      (L => INTLIST)
OPS   (FE : E -> E => INC)
MI
```

```
RUN FL(1 . 2 . 3) NUR
AS INTLIST: (2 . 3 . 4)
```

```
RUN FL(3 . 6 . 17) NUR
AS INTLIST: (4 . 7 . 18)
```

```
=End of file=
>EXIT
@POP
```

[PHOTO: Recording terminated Mon 13-Apr-81 6:58PM]

[PHOTO: Recording initiated Mon 13-Apr-81 7:04PM]

Link from OBJT, TTY 1

TOPS-20 Command processor 3A(32)-3
End of COMAND.CMD.1
@OBJT

***OBJT 12/2/79

>IN NAT5 NI

*** NATURAL NUMBERS ***

OBJ NAT

SORTS N

OK-OPS

 Z : -> N

 S : N -> N

 + : N N -> N

VARs

 M P : N

OK-EQNS

 (Z + M = M)

 (S(M)+ P = S(M + P))

JBO

 RUN Z NUR

AS N: Z

 RUN S(Z) NUR

AS N: S(Z)

 RUN S(S(Z)) NUR

AS N: S(S(Z))

 RUN S(S(S(Z))) NUR

AS N: S(S(S(Z)))

 RUN S(S(S(S(S(Z)))) NUR

AS N: S(S(S(S(S(Z))))

 RUN S(S(S(S(S(S(Z)))))) NUR

AS N: S(S(S(S(S(S(Z))))))

 RUN S(S(S(S(S(S(S(Z))))))) NUR
AS N: S(S(S(S(S(S(S(Z)))))))

 RUN S(S(S(S(S(S(S(S(Z))))))) NUR
AS N: S(S(S(S(S(S(S(S(Z)))))))

 RUN S(S(S(S(S(S(S(S(S(Z))))))) NUR
AS N: S(S(S(S(S(S(S(S(S(Z)))))))

 RUN S(S(Z))+ S(S(Z)) NUR
AS N: S(S(S(S(Z))))

 RUN S(S(Z))+ S(S(S(Z))) NUR
AS N: S(S(S(S(S(Z)))))

*** THIS PARAMETERIZED OBJECT IDENTIFIES 5 AND 0 ***

OBJ MOD5
SORTS N
OK-OPS
 Z : -> N
 S : N -> N
VARS
 N : N
OK-EQNS
 (S(S(S(S(S(Z))))))= Z)
JBO

*** WE NOW INSTANTIATE IT TO THE NATURALS AS ABOVE ***

IM (MOD5 => NAT5)
SORTS (N => NAT)
MI

 RUN Z NUR
AS NAT: Z

 RUN S(Z) NUR
AS NAT: S(Z)

 RUN S(S(Z)) NUR

AS NAT: S(S(Z))

 RUN S(S(S(Z))) NUR
AS NAT: S(S(S(Z)))

 RUN S(S(S(S(S(Z)))) NUR
AS NAT: Z

 RUN S(S(S(S(S(S(Z)))))) NUR
AS NAT: S(Z)

 RUN S(S(S(S(S(S(S(Z))))))) NUR
AS NAT: S(S(Z))

 RUN S(S(S(S(S(S(S(S(Z)))))))) NUR
AS NAT: S(S(S(Z)))

 RUN S(S(S(S(S(S(S(S(S(Z)))))))) NUR
AS NAT: S(S(S(Z)))

 RUN S(S(Z))+ S(S(Z)) NUR

?Warning: EXPRESSION CANNOT BE PARSED
 RUN S(S(Z))+ S(S(S(Z))) NUR

?Warning: EXPRESSION CANNOT BE PARSED

*** WHAT HAPPENS IF WE TRY IT ON INT? ***

 IM (MOD5 => INT5)
 SORTS (N => INT)
 MI

 RUN Z NUR
AS INT: Z

 RUN S(Z) NUR
AS INT: S(Z)

 RUN S(S(Z)) NUR
AS INT: S(S(Z))

 RUN S(S(S(Z))) NUR

AS INT: S(S(S(Z)))

RUN S(S(S(S(S(Z)))))) NUR
AS INT: Z

RUN S(S(S(S(S(S(Z)))))) NUR
AS INT: S(Z)

RUN S(S(S(S(S(S(S(Z))))))) NUR
AS INT: S(S(Z))

RUN S(S(S(S(S(S(S(S(Z))))))) NUR
AS INT: S(S(S(Z)))

RUN S(S(S(S(S(S(S(S(S(Z))))))) NUR
AS INT: S(S(S(S(Z)))

RUN S(S(Z))+ S(S(Z)) NUR

?Warning: EXPRESSION CANNOT BE PARSED
RUN S(S(Z))+ S(S(S(Z))) NUR

?Warning: EXPRESSION CANNOT BE PARSED

=End of file=
>IN NAT5S NI

OBJ NAT5S / INT
OK-EQNS

(5 = 0)

?Warning: INPUT ABNORMALLY TERMINATED

>EXIT

@POP

[PHOTO: Recording terminated Mon 13-Apr-81 7:05PM]

F PARTIAL ALGEBRAS WITH EQUATIONALLY DEFINED DOMAINS

We have investigated partial algebras such that the domains of definition of their operations can be equationally defined. These are very close in spirit to error algebras[Goguen 77], on which the semantics of OBJ is based; the main difference is that, for exceptional cases, an operation is just not defined, instead of producing an error message as in the case of error algebras.

Partial algebras, with equationally defined domains of definition for their operations have a strong expressive power. Moreover, they have initial algebras, relatively free algebras, and all limits and colimits. This supports an initial algebra semantics and many useful constructions.

We shall illustrate the concept with two examples. The first is a version of stack of integers as a partial algebra with equationally defined domains. We give the specification in an OBJ-like style[Goguen & Tardo 79], and will explain our notation below.

```
OBJ    STACK / INTEGER BOOL
SORTS  STACK
VARS   S: STACK N: INTEGER
OPS    EMPTY : -> STACK
        IEMPTY : STACK -> BOOL
        PUSH : INTEGER, STACK -> STACK
        POP : STACK : S : (ISEMPTY(S) = FALSE) -> INTEGER
        TOP : STACK : S : (ISEMPTY(S) = FALSE) -> INTEGER
EQNS   IEMPTY(EMPTY) = T
        IEMPTY(PUSH(N,S)) = F
        POP(PUSH(S,N)) = S
        TOP(PUSH(S,N)) = N
JBO
```

The only operations that are partial are POP and TOP. Both are defined exactly on those values of the STACK variable S such that the equation ISEMPTY(S) = FALSE holds. this is represented by the notation
POP : STACK : S : (ISEMPTY(S) = FALSE) -> INTEGER

and similarly for TOP.

our next definition is the specification of the data type path.

```
OBJ    PATH
SORTS  NODE PATH
VARS   A B C : NODE
        F G H : PATH
OPS    ID : NODE -> PATH
        DOM : PATH -> NODE
        COD : PATH -> NODE
        -- : PATH : G , PATH : F : (DOM(G) = COD(F)) -> PATH
```

EQNS $\text{DOM}(\text{ID}(A)) = A$
 $\text{COD}(\text{ID}(A)) = A$
 $F \cdot \text{ID}(A) = F$
 $\text{ID}(A) \cdot F = F$
 $(F \cdot G) \cdot H = F \cdot (G \cdot H)$

JBO

The only partial operation here is the composition of paths $\cdot$, which is defined exactly for those values of the variables F, G such that the equation
 $\text{DOM}(G) = \text{COD}(F)$

holds. (Note that the initial algebra of this specification is empty, but constants of types NODE and PATH can be added. Such constraints will specify a given graph, and then the initial algebra is the algebra of paths on that graph.)

In general, not only equations, but also Horn-like conditional equations are allowable. There is a precise formal definition of the concept of partial algebra with equationally defined domains satisfying a certain set of Horn axioms which can be given in terms of "essentially algebraic theories" as defined below [Gabriel Ulmer 71]. These generalize total algebras, and use the algebraic theory approach [Lawvere 63]. Algebraic theories have also been used by [Burstall & Goguen 77, Burstall & Goguen 80] for the semantics of their specification language CLEAR.

Definition 1: A small category T is an essentially algebraic theory if T has all finite limits (i.e. finite products and equalizers). A functor $H: T \rightarrow T'$ between two essentially algebraic theories is a morphism of essentially algebraic theories if and only if it preserves limits.

For instance, the specification of the data type PATH above corresponds to a theory T_{PATH} , in which each object is a finite limit of the objects NODE and PATH. The object COMPOSABLE (where the operation of path composition is defined) is the equalizer

$$\begin{array}{ccccc} \text{COMPOSABLE} & \longrightarrow & \text{PATH} \times \text{PATH} & \begin{array}{c} \xrightarrow{\pi_1} \text{PATH} \xrightarrow{\text{DOM}} \\ \xrightarrow{\pi_2} \text{PATH} \xrightarrow{\text{COD}} \end{array} & \text{PATH} \end{array}$$

and the equations correspond to forcing certain diagrams to commute in T .

Definition 2: A category B is a category of partial algebras with equationally defined domains if and only if B is equivalent to the category $\text{Alg}_T = \lim \text{Funct}(T, \text{Set})$ of (finite) limit preserving functors from T to the category of sets, for T some essentially algebraic theory.

If $S \subset \text{Ob}(T)$ is a set of objects of T such that any other object is a finite limit of objects in S , we say that S generates T . Then we can define a forgetful functor U_S from Alg_T to the category $S\text{-Set}$ of S -sorted sets, given on objects by $U_S(A)_s = A(s)$, for every s in S . For example, $S = \{\text{NODE}, \text{PATH}\}$ generates T_{PATH} , and U_S is in this case the functor sending each PATH data type to its NODE and PATH sets.

Theorem 3: For T an essentially algebraic theory, Alg_T is complete and cocomplete (in particular it has an initial object). If $S \subset \text{Ob}(T)$ generates T , then the functor $U_S: \text{Alg}_T \rightarrow S\text{-Set}$ has a left adjoint; i.e., there are free algebras.

There are also free extensions. For instance the specification

```

OBJ  GRAPH
SORTS NODE EDGE
OPS   DOM : EDGE -> NODE
      COD : EDGE -> NODE
JBO

```

has an associated (essentially algebraic) theory T_{GRAPH} which can be embedded in T_{PATH} , so that the injection $J: T_{\text{GRAPH}} \rightarrow T_{\text{PATH}}$ is a morphism of theories. This induces a functor $U_J: \text{Alg}_T \rightarrow \text{Alg}_T$ by composition with $J: A \mapsto A.J$; this functor considers every PATH -algebra as a GRAPH -algebra by forgetting about the operations of composition and identity. U_J has a left adjoint F_J , which gives the free extension of each graph to its algebra of paths. In general we have

Theorem 4: Let $H: T \rightarrow T'$ be a morphism of theories, and let $U_H: \text{Alg}_T \rightarrow \text{Alg}_{T'}$ be the functor induced by composition with H . Then U_H has a left adjoint. Our ultimate goal in this area is to extend the results on rewrite-rules known

for total algebras to the case of partial algebras with equationally defined domains, as a first step toward further extending these results to error algebras, and in particular obtaining for error algebras the powerful structural results mentioned above. This will provide theoretical support for some of the more experimental features of OBJ.

Using results of [Coste 77] we have isolated a deductive system for partial algebras with equationally defined domains which is both consistent and complete, and we are in the process of integrating a version of that deductive system with OBJ-style specifications like those given above. We also plan to study the relationship to the work of [Kaphengst & Reichel 77, Reichel & Hupbach & Kaphengst 80], and to the categorical work of [Gabriel Ulmer 71]. We believe that many of their infinitary constructions will specialize to the finitary case, which is important for the semantics of partial algebras with equationally defined domains, and for error algebras.

G STRICT ERROR ALGEBRAS DEFINED BY TESTS

We assume familiarity with error algebras[Goguen 77]. For the case of strict error algebras, i.e., error algebras where all the error elements reduce to an error constant E, the meaning of OK and ERROR equations can be captured as ordinary equations on the whole algebra if we introduce a ternary operation $\text{IFE}(_,_,_)$, such that $\text{IFE}(a,b,c)$ equals b in case a is the error constant and equals c otherwise. For example, if we want the OK part to satisfy the axioms of group theory we can impose equations such as

$$x \cdot (x)^{-1} = \text{IFE}(x,E,1)$$

$$x \cdot 1 = x$$

which correspond to the axiom about the inverse and neutral elements respectively. If we had just imposed

$$x \cdot x^{-1} = 1$$

as an ordinary equation, this would have given

$$E = E \cdot E^{-1} = 1$$

and then every element would collapse to the error element, because

$$x \cdot 1 = x \cdot E = E.$$

For any set $\bar{\Gamma}$ of OK-equations and ERROR equations encoded in this form, we have given a free construction, which associates to each strict error Σ -algebra another one satisfying $\bar{\Gamma}$, and having the obvious universal property. We have also proved that this process is conservative in that, if the Σ -operations restrict to the OK part, and we only have OK-equations, this construction is equivalent to imposing first the OK equations as ordinary equations on the OK part, and then adding an error element. The following is a summary of our definitions and results.

Definition 1: A strict (error) set is a pair (A,E) , where E belongs to A. A function $f:A \rightarrow B$ between two strict error sets is a strict (error) function if and only if either:

1. $|A| = |B| = 1$;
2. $|B| = 1$; or
3. none of the above holds but $f^{-1}\{E\} = \{E\}$.

For an index set S of sorts, an S-sorted strict error set is a family $\{(A_s, E)\}_{s \in S}$ of strict error sets, and similarly an S-sorted strict error function

is a family of strict error functions.

Definition 2: For Σ an S-sorted signature of operations, a strict (error) Σ -algebra is an S-sorted strict error set $A = \{(A_s, E)\}_{s \in S}$ together with a Σ -algebra structure on A such that every operation $\sigma \in \Sigma_{w,1}$ gives as result the error element of A_1 whenever one of its operands is an error element; i.e.,

$$(1) \quad \sigma(x_1, \dots, E, \dots, x_n) = E$$

for E in any place i, for $1 \leq i \leq n$. A strict (error) Σ -homomorphism is a Σ -homomorphism that is also a strict error function.

Note that every strict error Σ -algebra can be made into a Σ_{IFE} -algebra, for Σ_{IFE} the signature obtained from Σ by adding a constant E for each sort s in S, and a ternary operation IFE: $s, t, t \rightarrow t$ for each pair of sorts s, t in S. Given a strict error Σ -algebra, we interpret the constants E as error elements, and we define $IFE(a, b, c)$ to be equal to b if $A = E$, and to be equal to c otherwise. With this definition every strict error homomorphism becomes a Σ_{IFE} -homomorphism.

Theorem 1: Let $\bar{\Gamma}$ be a set of Σ_{IFE} equations, and for every Σ_{IFE} algebra A let $A/\bar{\Gamma}$ be the quotient algebra obtained from A by imposing the equations in $\bar{\Gamma}$. Then if A is a strict error Σ -algebra $A/\bar{\Gamma}$ is also a strict error Σ -algebra, the canonical quotient $q: A \rightarrow A/\bar{\Gamma}$ is a strict error homomorphism, and for any strict error Σ -homomorphism $f: A \rightarrow B$ such that B satisfies $\bar{\Gamma}$ as a Σ_{IFE} -algebra, there exists a unique strict error homomorphism $\bar{f}: A/\bar{\Gamma} \rightarrow B$ such that $\bar{f}.q = f$.

To each ordinary Σ -algebra A we can associate a strict error Σ -algebra A_E by adding a new element E to each sort A_s , and extending the operation σ in Σ in the obvious way. The following theorem shows that the construction of Theorem 1 can be seen as a "conservative extension" of the construction of imposing ordinary Σ -equations when only OK-equations are imposed.

Theorem 2: Let Δ be a set of Σ -equations such that, for each equation, the set of variables occurring in its right hand side is contained in the set of variables occurring in its left hand side. Let $\bar{\Gamma}$ be the set of Σ_{IFE} equations obtained from those in Δ as follows: for each $u = v$ in Δ , if the variables occurring in u and v are the same, $u = v$ belongs to $\bar{\Gamma}$; otherwise $u = \text{IFE}(u, E, v)$ belongs to $\bar{\Gamma}$. Then for every Σ -algebra A there exists a natural isomorphism of strict error Σ -algebras

$$(A/\Delta)_E \cong (A_E)/\bar{\Gamma}.$$

H MODEL-THEORETIC CHARACTERIZATION OF RELATIONAL CLASSES OF PROGRAM SCHEME INTERPRETATIONS

Making operations and tests into elements of a signature of operations Σ , any interpretation of a program scheme can be seen as a continuous Σ -algebra, i.e., as an ω -cpo together with a Σ -algebra structure such that the operations preserve limits of ω -chains [Goguen, Thatcher, Wagner & Wright 77, Nivat 75]. For the equivalence problem of program schemes to be tractable, it is convenient to consider not all interpretations, but just those in a class defined by some natural property.

One such case of particular interest, because it allows proofs by computation induction, is the relational classes, i.e., the classes of algebras A such that there is some set of pairs of finitary Σ -expressions (u,v) such that

$$u(a_1, \dots, a_n) \leq v(a_1, \dots, a_n)$$

for every $(a_1, \dots, a_n) \in A^n$ (where n is the number of distinct variables occurring in u or v) for each (u,v) . In [Meseguer 81] those classes have been characterized model-theoretically by the following "Birkhoff-like" result:

Theorem 1: A class of continuous Σ -algebras is relational if and only if it is closed under

- (i) products
- (ii) (continuous) full subalgebras
- (iii) (continuous) quotients, and
- (iv) algebraic completions.

REFERENCES

- [Burstall & Goguen 77]
Burstall, R. M. and Goguen, J. A.
Putting Theories together to Make Specifications.
Proc. 5th Int. Joint Confr. on Artificial Intelligence, 1977.
- [Burstall & Goguen 78]
Burstall, R. M. and Goguen, J. A.
Semantics of CLEAR.
1978.
unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warszawa.

- [Burstall & Goguen 80]
 Burstall, R. M., and Goguen, J. A.
 The Semantics of CLEAR, a Specification Language.
 In Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification, Lecture Notes in Computer Science, volume 86, pages 292-332. Springer-Verlag, 1980.
- [Burstall & Goguen 81]
 Burstall, R. M. and Goguen, J. A.
 An Informal Introduction to CLEAR, a Specification Language.
 In Boyer, R. and Moore, J., editor, The Correctness Problem in Computer Science, . Academic Press, 1981.
- [Coste 77]
 Coste, M.
Localisation dans les Categories de Modeles.
 PhD thesis, Universite Paris Nord, 1977.
- [Gabriel Ulmer 71]
 Gabriel, P. and Ulmer, F.
Lokal Prasentierbare Kategorien.
 Springer-Verlag, 1971.
 Springer Lecture Notes in Mathematics, vol. 221.
- [Goguen & Burstall 78]
 Goguen, J. A. and Burstall, R. M.
Some Fundamental Properties of Algebraic Theories: a Tool for Semantics of Computation.
 Technical Report, Dept. of Artificial Intelligence, University of Edinburgh, 1978.
 DAI Research Report No. 5; to appear in Theoretical Computer Science.
- [Goguen & Burstall 80a]
 Goguen, J. A. and Burstall, R. M.
An Ordinary Design.
 Technical Report, SRI International, 1980.
 Draft report.
- [Goguen & Burstall 80b]
 Goguen, J. A., and Burstall, R. M.
CAT, a System for the Structured Elaboration of Correct Programs from Structured Specifications.
 Technical Report, SRI, International; Computer Science Lab, 1980.
 Based on unpublished working draft, UCLA and SRI, 1979.
- [Goguen & Meseguer 81]
 Goguen, J. A. and Meseguer, J.
 Completeness of Many-sorted Equational Logic.
 1981.
 to appear, SIGACT Newsletter.

- [Goguen & Parsaye-Ghomi 81]
 Goguen, J. A. and Parsaye-Ghomi, K.
 Algebraic Denotational Semantics using Parameterized Abstract
 Modules.
 In J. Diaz and I. Ramos, editor, Formalizing Programming
 Concepts, pages 292-309. Springer-Verlag, Peniscola, Spain,
 1981.
 Lecture Notes in Computer Science, volume 107.
- [Goguen & Tardo 79]
 Goguen, J. A. and Tardo, J.
 An Introduction to OBJ-T.
 In Specification of Reliable Software, pages 170-189. IEEE,
 1979.
 Cambridge, Mass., April 1979.
- [Goguen, Thatcher, Wagner & Wright 77]
 Goguen, J. A. , Thatcher, J. W., Wagner, E. and Wright, J. B.
 Initial Algebra Semantics and Continuous Algebras.
Journal of the Association for Computing Machinery 24(1),
 January, 1977.
- [Goguen 77]
 Goguen, J. A.
 Abstract Errors for Abstract Data Types.
 In Working Conference on Formal Description of Programming
 Concepts. IFIP, 1977.
 Also published by North-Holland, 1979, edited by P. Neuhold.
- [Goguen 80]
 Goguen, J. A.
 How to Prove Algebraic Inductive Hypotheses without Induction:
 with applications to the correctness of data type
 representations.
 In W. Bibel and R. Kowalski, editor, Proceedings, 5th
 Conference on Automated Deduction, pages 356-373.
 Springer-Verlag, Lecture Notes in Computer Science, volume
 87, 1980.
- [Kaphengst & Reichel 77]
 Kaphengst, H. and Reichel, H.
 Initial Algebraic Semantics for Non-Context-Free Languages.
 In Springer-Verlag Lecture Notes in Computer Science, vol. 56,
 pages 120-126. Springer-Verlag, 1977.
- [Lawvere 63]
 Lawvere, F. W.
 Functorial Semantics of Algebraic Theories.
 (Proc. Nat. Acad. Sciences) 50, 1963.
 Summary of Ph.D. Thesis, Columbia University.
- [Levitt, Robinson & Silverberg 79]
 Levitt, K., Robinson, L. and Silverberg, B.
The HDM Handbook, vols. I,II,III.
 Technical Report, SRI, International; Computer Science Lab,
 1979.

[Meseguer 80]

Meseguer, J.
Varieties of Chain-Complete Algebras.
Journal of Pure and Applied Algebra 19:347-383, 1980.

[Meseguer 81]

Meseguer, J.
A Birkhoff-like Theorem for Algebraic Classes of
Interpretations of Program Schemes.
In J. Diaz and I. Ramos, editor, Formalization of Programming
Concepts, pages 152-168. Springer-Verlag, Peniscola, Spain,
1981.
Lecture Notes in Computer Science, volume 107.

[Nivat 75]

Nivat, M.
On the Interpretation of Polyadic Recursive Schemes.
In Symposia Mathematica, 15. Academic Press, 1975.

[Reichel & Hupbach & Kaphengst 80]

Reichel, H., Hupbach, U. R., and Kaphengst, H.
Initial Algebraic Specifications of Data Types, Parameterized
Data Types, and Algorithms.
Technical Report, VEB Robotron, Zentrum fur Forschung und
Technik, 1980.

[Tardo 81]

Tardo, J.
The Design, Specification and Implementation of OBJT: A
Language for Writing and Testing Abstract Algebraic Program
Specifications.
PhD thesis, UCLA, Computer Science Department, 1981.

DATE
FILME