Inter-Office Memorandum

To	Mesa Group	Date	October 25, 1977
From	Barbara Koalkin	Location	Palo Alto
Subject	Debugger Interpreter	Organization	SDD/SD

XEROX

Filed on: [MAXC]<KOALKIN>DInterpreter.BRAVO

DRAFT

This memo represents a preliminary attempt at specifying what the proposed debugger interpreter will look like. A full interpreter at this point seems unreasonable and probably of marginal value. However, a minimal subset of the language would be a valuable extension to the current debugger command language.

We have specified the following subset of the Mesa TYPE calculus as being acceptable to this interpreter:

--dot notation: *a.b.c*
--assignment: ←
--dereference: ↑
--indexing: *[ ]*
--addressing: "*@expression*"
--*LOOPHOLE*

With the help of some of the compiler's modules we will be able to enforce strong type-checking in the interpreter.

The proposed interpreter should help to alleviate many of the problems regarding displaying and assigning values to complicated data structures that now force the user to go down to octal level debugging.

In terms of the formal Mesa syntax the grammar for the proposed interpreter should include the following expressions:

```
Expression    ::= AssignmentExpr | Disjunction
AddingOp      ::= + | -
AssignmentExpr ::= LeftSide ← RightSide
Conjunction   ::= Negation | Conjunction AND Negation
Disjunction   ::= Conjunction | Disjunction OR Conjunction
Factor        ::= - Primary | Primary
IndexedAccess ::= ( Expression ) [ Expression ] | Variable [ Expression ]
IndirectAccess ::= ( Expression ) ↑ | Variable ↑
LeftSide      ::= identifier |                          -- Call in Statement
                  IndexedAccess | QualifiedAccess | IndirectAccess |
                  LOOPHOLE [ Expression ] |
                  LOOPHOLE [ Expression , TypeSpecification ]
Literal       ::= numericLiteral |      -- all defined outside the grammar
```

XEROX SDD ARCHIVES
I have read and understood
Pages_____To_____
Reviewer_____Date_____
# of Pages_____Ref. 11SDD-357

```
                    stringLiteral |
                    characterLiteral
MultiplyingOp   ::= * | / | MOD
Negation        ::= Relation | Not Relation
Not             ::= ~ | NOT
Primary         ::= Variable | Literal | ( Expression ) | @ LeftSide
Product         ::= Factor | Product MultiplyingOp Factor
QualifiedAccess ::= ( Expression ) . identifier | Variable . identifier
Relation        ::= Sum | Sum RelationTail
RelationalOp    ::= # | = | < | <= | > | >=
RelationTail    ::= RelationalOp Sum | Not RelationalOp Sum |
                    IN SubRange | Not IN Subrange
RightSide       ::= Expression
Subrange        ::= SubrangeTC | TypeIdentifier          -- SubrangeTC, TypeIdentifier in TypeSpecification
Sum             ::= Product | Sum AddingOp Product
Variable        ::= LeftSide
```

There are some questions in my mind about including the following expressions (we should discuss these further):

```
Expression      ::= IfExpr
IfExpr          ::= IF Expression THEN Expression ELSE Expression
BuiltinCall     ::= MIN [ ExpressionList ] | MAX [ ExpressionList ] | ABS [ Expression ] |
                    LENGTH [ Expression ] | BASE [ Expression ] |
                    TypeOp [ TypeSpecification ] |
                    DESCRIPTOR [ Expression ] |
                    DESCRIPTOR [ Expression , Expression ] |
                    DESCRIPTOR [ Expression , Expression , TypeSpecification ]
Component       ::= empty | Expression
ComponentList   ::= KeywordComponentList | PositionalComponentList
Constructor     ::= OptionalTypeId [ ComponentList ]
ExpressionList  ::= Expression | ExpressionList , Expression
FunctionCall    ::= BuiltinCall | Call
KeywordComponent    ::= identifier : Component
KeywordComponentList    ::=
                            KeywordComponent |
                            KeywordComponentList , KeywordComponent
LeftSide        ::= Call | MEMORY [ Expression ] | REGISTER [ Expression ]
PositionalComponentList    ::=
                            Component |
                            PositionalComponentList , Component
Primary         ::=  FunctionCall | Constructor
TypeOp          ::= SIZE | FIRST | LAST
```

The following expressions seem to be of marginal value to consider including:

```
Expression      ::= NewExpr | SelectExpr
NewExpr         ::= NEW Variable OptCatchPhrase
SelectExpr      ::= SelectExprSimple | SelectExprVariant
SelectExprSimple    ::= SELECT LeftItem FROM                -- LeftItem in Statement
                            ExprChoiceList
                            ENDCASE => Expression
SelectExprVariant   ::= WITH OpenItem SELECT TagItem FROM-- OpenItem, TagItem in
ChoiceList      ::= AdjectiveList => Expression , |         -- AdjectiveList in Statement
                    ChoiceList AdjectiveList => Expression ,
```

```
ExprChoiceList ::= TestList => Expression , |                -- TestList in Statement
                   ExprChoiceList TestList => Expression ,
```

Remaining Questions:
--whether the interpreter should use the same scanning mechanism as the compiler; the current thought seems to be to keep it a separate mechanism and have it build its own trees with information relevant to interpreting the value of expressions
--what sort of user interface to have for the interpreter; whether the present set of Interpet commands should be replaced simply by one INTERPRET command or accept interpreted values as input for all commands
--what kind of procedure calls to allow, if any - for instance, how about interpret call of nested procedures and returning large parameter records
--whether we should allow user-defined temporary variables
--the above specified grammar is an expression evaluator - what about evaluating statements (and multiple statements)
--what context to evaluate in (current module, current configuration, defs.foo)
--expandint to conditional breakpoints

Distribution:
Mesa Group
Ed Satterthwaite
John Weaver