Internal Memo

XEROX

To From

Mimosa Users Russ Atkinson
PARC/CSL

Subject Date

Cedar/Mesa language changes February 6, 1990

Revised on May 15, 1991.
Minor clarifications, Christian Jacobi, May 6, 1993.

Cedar/Mesa Language Changes

Overview

This note describes the changes to the Cedar/Mesa language that are either completed or proposed for the Mimosa compiler. The extensions to the Cedar/Mesa language are intended to foster portability of code from one target architecture to another.

There are several compilers mentioned in this note:

> ***XDE compiler*** - the compiler that is in the current release of XDE (Xerox Development environment). This compiler compiles Mesa into the XDE version of the PrincOps architecture, which runs on Xerox product hardware. The term *XDE compiler* is meant to apply to all compilers for product Mesa software.
>
> ***Cedar 7.0 compiler*** - the compiler for with the Cedar 7.0 release. This compiler compiles the Cedar language into the CSL version of the PrincOps architecture. It also contains some extensions that are compatible with Mimosa.
>
> ***Mimosa compiler*** - a family of retargetable compilers. The Sun version of the Mimosa compiler translates Cedar into C code suitable for execution on either Sun 3 or Sun 4 machines using SunOS*, a variant of UNIX*. Other targets are under consideration.
>
> ***MesaPort compiler*** - This compiler compiles extended Mesa into the Intel 80386 architecture. It is not under active development at this time.

---

* SunOS is a trademark of Sun Microsystems. UNIX is a trademark of AT&T.

Parametrization

The language and compiler have been parametrized. That is, some features of the language are defined as machine dependent. The basic types have machine dependent specifications, as do certain compiler limitations. This section does not describe all of the parameters [to be covered in the Mimosa compiler internal documentation], but it covers the parameters of interest to most programmers.

It is worth noting that sufficiently similar targets may have identical parameters. For example, the Sun 3 (68020) target and Sun 4 (Sparc) target have been given identical parametrization to encourage portability. This has the further benefit of making the C code generated for these targets identical.

bitOrder

The target bit numbering order is either msBit (most significant bit is bit zero) or lsBit (least significant bit is bit zero). Examples of msBit architectures are PrincOps, IBM 360 & 370, Motorola 680x0 & 88000, and Sun Sparc. Examples of lsBit architectures are Intel 80x86 and DEC VAX. For the purposes of Mimosa, bit order and byte order are taken to be identical.

bitsPerWord & bitsPerLongWord

These quantities give the natural word width of the target. A few targets, notably PrincOps, have two "natural" word widths, based on their instruction sets. Most targets, notably the Sun Sparc and Motorola 68020, have only one.

These numbers determine the size of some data types, determine how tightly records are packed, and determine the amount of padding for local variables and arguments. In Mimosa, both bitsPerWord and bitsPerLongWord are assumed to be powers of two.

bitsPerAU (addressing granularity)

The number of bits per addressing unit is mostly used when generating addresses. It also determines the units used for the SIZE operator. The supported values for bitsPerAU are 8, 16, and 32.

bitsPerProc

The number of bits per procedure descriptor is given by bitsPerProc.

Some parameters for various targets

	PrincOps	80386	68020
bitOrder	msBit	lsBit	msBit
bitsPerWord	16	32	32
bitsPerLongWord	32	32	32
bitsPerAU	16	8	8
bitsPerProc	16/32*	32	32

* PrincOps is a generic name used to describe a number of Xerox architectures. The architectural limits have changed

with time. XDE uses a PrincOps with 32 bit procedure descriptors, and PARC/CSL uses an older PrincOps with 16 bit procedure descriptors.

Basic types

Parametrized types

These basic types and type constructors are parametrized according to the target architecture.

UNIT: bitsPerAU

INTEGER, CARDINAL, WORD, UNSPECIFIED: bitsPerWord

NAT, NATURAL: bitsPerWord - 1

-- intersection of INTEGER and CARDINAL

PROC, PROCEDURE: bitsPerProc

SIGNAL, ERROR: bitsPerSignal

POINTER: bitsPerPtr

STRING: bitsPerPtr

REF, LIST: bitsPerRef

DESCRIPTOR: bitsPerDesc

INT, CARD: bitsPerLongWord

REAL: bitsPerReal

DCARD, DINT, DWORD: 2*bitsPerLongWord

DREAL: 2*bitsPerReal

The names of the types listed above are chosen as a compromise between historical naming and regular naming. For example, it is far from pleasing for NAT to have bitsPerWord - 1 bits, while CARD and INT have bitsPerLongWord bits. However, other choices would tend to invalidate existing code, which would be even farther from pleasing.

If bitsPerAU = bitsPerWord, then UNIT is the same type as WORD. Otherwise, UNIT is a subrange of WORD.

Only the Mimosa compiler supports arithmetic of more than 32 bits in precision.

LONG is used to convert from parametrized types with sizes equal to bitsPerWord to parametrized types with sizes equal to bitsPerLongWord. If bitsPerWord = bitsPerLongWord, then LONG has no effect. For example, when bitsPerWord = bitsPerLongWord, then the following pairs of types are equivalent:

CARDINAL, LONG CARDINAL

INTEGER, LONG INTEGER

STRING, LONG STRING

POINTER TO T, LONG POINTER TO T

DESCRIPTOR FOR T, LONG DESCRIPTOR FOR T

For dubious compatibility reasons, the use of LONG REAL is equivalent to DREAL, but the latter form is strongly encouraged. The support for LONG REAL may be withdrawn as a stronger encouragement.

Explicit precision types

To facilitate conversions that require preservation of type size, a group of built in types has been

defined that do not have parametrized sizes. The names of the types indicate the the number of bits occupied before any applicable padding. These basic types and type constructors are:

NAT15

INT16, CARD16, WORD16, UNSPEC16

NAT31

INT32, CARD32, REAL32, WORD32, UNSPEC32

INT64, CARD64, REAL64

Other new predefined types

A few other types have been predefined in the Mimosa compiler for convenience and consistency:

BIT:	WORD [0..1]
BYTE:	WORD [0..255]

New predefined types included in the Cedar 7.0 compiler

The following predefined types have been included in the Cedar 7.0 compiler to assist in writing portable code:

INT16:	16 bits signed
INT32:	32 bits signed
CARD16:	16 bits unsigned
CARD32:	32 bits unsigned
NAT15:	intersection of INT16 & CARD16
NATURAL:	alias for NAT
BIT:	equivalent to CARDINAL[0..1]
BYTE:	equivalent to CARDINAL[0..255]
UNIT:	equivalent to WORD for PrincOps
REAL32:	32 bits floating point

Addressability

Addresses of fields

The Cedar 7.0 and XDE Mesa compilers have long held that addressing of record fields or array elements was only legal when the address was the address of a word. In the PrincOps world, the addressing granularity was also a word, or 16 bits. For Mimosa, addressable fields must be an integral number of words long, even when the primitive addressing granularity of the machine would permit machine addresses to point at smaller units (such as a byte or halfword).

The rationale for this decision is twofold. First, fewer assumptions about addressing granularity make code more portable. Second, the compiler is unable to distinguish the address of a field from the address of a simple variable of the same type, and the representation of a simple variable requires padding to a word boundary.

Padding of variables to word boundaries is typically done for performance. A simple local variable declared to hold a subrange from [0..10), for example, occupies a whole word, and is padded with 0 bits. This enables the compiler to avoid using subword operations, which frequently results in

more efficient code.

The Mimosa compiler enforces the constraint that valid Mesa addresses can only be formed for variables that are on word boundaries, and occupy an integral number of words.

For further explanation, see the section "Addressability rationale and discussion" later in this document.

SIZE[T]

SIZE[*T*] returns the number of addressing units occupied by simple variables of type *T*. This is *not* the same as the minimal number of addressing units for a packed field of type *T*! For example, if the number of bits per addressing unit is 8, and the number of bits per word is 32, then SIZE[BYTE] = 4.

> Some PrincOps dependent programs will not work when they assume that SIZE[*T*] returns the number of words occupied by variables of type *T*. Such programs should use WORDS[*T*] instead.

Currently (and correctly) SIZE is used for two main reasons: pointer arithmetic and allocation. As mentioned in the section on Addressing, a Mesa pointer is treated as the address of a variable that has been padded to word boundaries where necessary. The compiler has traditionally followed this approach, and we see no reason to change it now (it is not easy to change). Therefore, SIZE must give the number of addressing units in the padded variable. For example, assume that we have the following code:

```
index: Index;
varA: ARRAY Index OF T;
ptrA: POINTER TO T = @varA[index];
ptrB: POINTER TO T = @varA[index+1];
ptrC: POINTER TO T = ptrA + SIZE[T];
```

We expect it to be true that ptrB = ptrC, and Mimosa makes that guarantee (provided that index+1 is still a valid value for Index). However, for time efficient access, the elements of varA are padded to word boundaries. Therefore, the definition of SIZE must be consistent with this usage.

Address arithmetic

It is strongly recommended that all addressing arithmetic be performed using the SIZE operator, as in the following examples:

```
ptr ← ptr + SIZE[T]
ptr ← ptr + SIZE[T]*7
ptr ← ptr + SIZE[T1]+SIZE[T2]
```

Other forms of address arithmetic may not be portable.

Arithmetic changes

Signed vs. unsigned

The arithmetic operators for signed and unsigned arithmetic are assumed to have different

properties. In cases where the operands are mixed signed and unsigned numbers the operation to be used, and the interpretation of the results, may be ambiguous, and require some decisions by the compiler.

As far as practical the Mimosa compiler makes signed/unsigned decisions based on range analysis and the target type. For example, if we have the following assignment:

```
Z := X / Y
```

and X, Y, and Z are single precision fixed point variables, then the decision about whether to use signed or unsigned division is based on whether or not X and Y can assume negative values. If either can, then we use signed division, other wise we use unsigned division. If we use signed division and either X or Y can assume unsigned values outside of

The case of addition is slightly more complex, since we can use the same machine addition operator for either signed or unsigned arithmetic. Here we try to resolve the ambiguity of the result based on the target. In most cases this becomes a simple case of bounds checking.

<<We need a better description here>>

Mixed precisions

In operations where the precisions of the operands differ the maximum precision operand determines the precision of the result. For example, if one operand of an addition is INT64 and the other is REAL32, then the operands are first converted to type REAL64, then a REAL64 operation is performed, and the result is also REAL64. This rule applies to all arithmetic operations (+, -, *, /, MOD, **) and comparisons (<, >, <=, >=, =, #).

Arithmetic defined by host

For efficiency, the exact definitions for arithmetic is supplied by the target machine. If Mimosa is generating C code, for example, operators like signed division operate according to whatever C does on the target machine.

Floating point

For efficiency, floating point exceptions need not be raised by the basic operators unless that target machine supports them as the default. Since many implementations of C, for example, do not have floating point exceptions, code generated by Mimosa need not support them either.

Relaxed limits

Variable sizes

A variable of known size must be strictly less than LAST[INT]+1 (currently 2**31 bits, or 2**28 bytes) in size. This is substantially larger than the previous limit (2**16 words for some types, 2**16 bits for other types). This limit does not apply to sequences, although attempting to evaluate BITS[T[N]] for a large sequence type T[N] may fail if the result would be larger than LAST[INT].

Enumeration types

An enumeration may contain values such that 0 <= ORD[value] <= LAST[CARDINAL]. For enumeration types ORD[e] is of type CARDINAL. The previous limit in Mesa was that an enumeration had to be no more than bitsPerWord in precision.

Array indexes

Arrays can be indexed by any indexing type so long as the resulting array does not exceed the maximum variable size. An indexing type is signed (INTEGER), unsigned (CARDINAL), an enumeration, or a subrange of an indexing type.

Subrange types

A subrange type may be formed from any indexing type up to bitsPerLongWord in precision. The previous limit in Mesa was that a subrange type had to be no more than bitsPerWord in precision.

Other limits

Other limits, such as the size of global frames and local frames, may be extended on a machine dependent basis. Internally, the Mimosa compiler must be prepared to handle limits up to LAST[INT], but not all targets will support this size.

Record alignment and layout

Normal records

The language is defined so that the compiler can reorder fields at will, provided that the decisions are consistent for given types. However, the Mimosa compiler does not take advantage of this freedom *at this time*, and it is quite likely that some code depends on the current strategy for record layout.

Records may be packed or unpacked. Unpacked records are laid out so that each field takes an integral number of words. Packed records are laid out such that fields not larger than a word do not cross word boundaries, and that fields larger than a word start on word boundaries and are padded to occupy an integral number of words.

For the Sun targets, normal records are unpacked by default. Applications that need to optimize space over time should use packed records, as in:

```
PackedRec: TYPE = PACKED RECORD [a, b: CARD16, c: CARD32];
```

Variables of type PackedRec occupy 64 bits. As discussed earlier, for 32 bit word targets, the compiler does not allow taking the address of either the a or b field, although the c field is addressable. Few applications should be affected by this constraint.

MACHINE DEPENDENT records

This design is taken from the MesaPort compiler with minor changes.

The syntax has been expanded to include some new options for machine dependent records:

typeCons ::= MACHINE DEPENDENT RECORD ...
typeCons ::= depOptions MACHINE DEPENDENT RECORD ...
depOptions ::= unitOption bitOrderOption
depOptions ::= bitOrderOption unitOption
depOptions ::= unitOption
depOptions ::= bitOrderOption
unitOption ::= WORD8 | WORD16 | WORD32 | WORD64
bitOrderOption ::= MSBIT | LSBIT | NATIVE

The unitOption specifies the number of bits in the major unit for field position specifications. If it is not given, then the number of bits in the addressing unit of the target is used. The unitOption is also used to determine the alignment constraints for a machine dependent record.

> The MesaPort compiler defines the default for the unitOption as WORD16. The MesaPort compiler also does not support the WORD64 option.

The bitOrderOption specifies the numbering of bits within a unit of the size specified by unitOption. If it is not given, or if it is given as NATIVE, then the default bit order for the target machine is used.

Machine dependent record declarations nested within other machine dependent record specifications by default use the unit and bit order specifications of the containing record declaration.

MACHINE DEPENDENT records (MDRs) are always tightly packed, with warnings being issued for gaps between fields, and errors for overlapping fields. Fields in MDRs have no required alignment, so they can span word boundaries without warnings. The only exception is that REF containing fields must be word aligned.

> The MesaPort compiler issues errors for some fields that cross word boundaries. The Mimosa compiler is prepared to be more permissive.

Field positions can be either explicit or implicit. Machine dependent record types with explicit field positions have a complicated set of options described below. Machine dependent record types with implicit field positions ignore these options, and are laid out from most significant to least significant bit within a word, and in increasing address order within a record.

> This description will need to change when the lsBit option is supported. The lsBit option is not yet supported for implicit field positions, and fields may not overlap word boundaries if the lsBit option is used.

The meaning of the field specifications for a machine dependent record is defined in terms of canonical bit number. The canonical bit number for Mimosa is defined as having 0 as the most significant bit in the first word, with bit N-1 being the least significant, where N is the number of bits in the word.

To translate a field specification for some record component into the canonical bit numbers, assume the following:

G = the number of bits for the unitOption (8, 16, 32, or 64)
B = the number of bits for the component type
N = the number of bits in the target machine word
C[X] = G*(X XOR N-1) -- *XOR is exclusive OR*

Then the canonical bit numbers (First bit# through Last bit#) are given by the following table.

Bit Order	Specification	First bit#	Last bit#
MSBIT	(U: L..R)	U*G+L	U*G+R

MSBIT	(U)	U*G	U*G+B-1
LSBIT	(U: L..R)	C[U]-L	C[U]-R
LSBIT	(U)	C[U]-B+1	C[U]

As an example, the following record declarations are equivalent in the sense that the fields have identical types and span identical ranges of bits. The bit numbers given by Type1 are the same as the canonical bit numbering, and are the same as the positions assigned if the field specifications were implicit.

```
Type1: TYPE = MSBIT WORD32 MACHINE DEPENDENT RECORD [
    f1 (0: 0..7): CHAR,
    f2 (0: 8..15): BYTE,
    f3 (0: 16..31): INT16,
    f4 (0: 32..63): INT32];

Type2: TYPE = MSBIT WORD8 MACHINE DEPENDENT RECORD [
    f1 (0: 0..7): CHAR,
    f2 (0: 8..15): BYTE,
    f3 (0: 16..31): INT16,
    f4 (4): INT32];

Type3: TYPE = LSBIT WORD16 MACHINE DEPENDENT RECORD [
    f1 (1: 15..8): CHAR,
    f2 (1: 7..0): BYTE,
    f3 (0: 15..0): INT16,
    f4 (2): INT32];

Type4: TYPE = LSBIT WORD8 MACHINE DEPENDENT RECORD [
    f1 (3: 7..0): CHAR,
    f2 (2: 7..0): BYTE,
    f3 (0: 15..0): INT16,
    f4 (4): INT32];
```

It is worth remembering that for LSBIT specifications the unit specified by the unitOption specifies the unit holding bit with the highest canonical bit number, while for MSBIT specifications the unit specified by the unitOption specifies the unit holding the bit with the lowest canonical bit number.

The intention of this baroque design is to allow descriptions of records to be specified relative to one architecture and to be compiled correctly for a variety of architectures. However, since the layout of the basic types may differ from machine to machine, the intent is not quite realized. In particular, quantities more than one byte long cannot be shipped reliably from one machine to another without conventions about whether the bytes are part of arithmetic quantities or byte strings. Most often, networks preserve byte order at some protocol level, and the client must rearrange bytes to recover arithmetic quantities.

This design allows exact specification of which bits are used to store a field. However, it does not specify what bit pattern is used to represent a value within the field, which always uses the natural bit order.

Field alignment

Different machines have different alignment constraints. For example, in PrincOps local and global frames are aligned on quad word (8 byte) boundaries. For the Sun 3 there are very few

alignment constraints, but for the Sun 4 all loads (and stores) of N bytes (N = 1, 2, 4, 8) using address A must have A MOD N = 0. Mimosa has chosen to make the Sun 3 and Sun 4 targets obey the same alignment constraints so as to have more portable code.

For a basic type T we normally assume that POINTER TO T will have addressing alignment such that access to the value will be efficient, *but no worse than word aligned!* This has its largest effect on record layout, where fields are padded and aligned to put the fields on boundaries that permit efficient access. Records also inherit their alignment from the strongest alignment of their fields. For example, for the Sun 4 target, consider the following record type declaration:

```
RT: TYPE = RECORD [a: INT16, b: INT32, c: INT64];
```

The components will be laid out with the following bit offsets:

```
RT: TYPE = RECORD [
    a: INT16,          -- start: 0, bits: 32 (padded)
    b: INT32,          -- start: 32, bits: 32
    c: INT64]          -- start: 64, bits: 64
```

The whole record will be 128 bits long, and will have an alignment constraint of 64 bits.

> As of this writing the compiler does not support 64 bit alignment constraints, and the generated C code is such that 64 bit alignment is never required. This state of affairs may well become the official truth, since changing it requires significant compiler work and imposes more constraints on the programmer for dubious efficiency benefits.

Records that are machine dependent are both useful and dangerous. They are useful because the client can put the bits anywhere they are desired, even with small fields crossing word boundaries. They are dangerous because forming the addresses of fields within machine dependent records can be used to get around the constraint checking that the compiler tries to enforce.

> As of this writing the compiler does not support subword fields crossing word boundaries for LSBIT machines or records. Although we may remedy this situation, the remedy is likely to take some time to come about.

Painted record types

In the Cedar 7.0 compiler, normal record types are *painted* in definitions modules and *unpainted* in implementation modules. This means that in implementation modules record type equivalence is determined structurally, whereas for definitions modules record type equivalence is determined by the occurrence of the type constructor. For example, given two type declarations:

```
RecordTypeA: TYPE = RECORD [a, b: INT];
RecordTypeB: TYPE = RECORD [a, b: INT];
```

For the Cedar 7.0 compilers these two types are different (and not mutually assignable) if they occur in a definitions module, but are equivalent if they appear in an implementation module.

For Mimosa and XDE Mesa, normal record types are always painted, so that for the above example the record types are not equivalent, and variables of those types cannot be freely assigned to each other. This makes the semantics of type equivalence the same for definitions and implementations modules. It also has some practical benefits for the runtime support, because descriptions of types can be much smaller.

New features

BITS[*T*], BYTES[T], UNITS[T], and WORDS[T]

Additional types and type operators have been added to facilitate the writing of portable code. For BITS and BYTES, a minimal result is required for subrange and enumeration types (e.g. BITS [BOOL] = 1, BITS[BYTE] = 8, BYTES[BYTE] = 1), and for MACHINE DEPENDENT RECORD types. Rounding to packed sizes is required for records containing packed sequences. For all other types, the compiler is permitted to round up the number of bits or bytes to an addressing unit boundary.

Operation	Units	Unit definition
BITS	BIT	[0..1]
BYTES	BYTE	[0..255]
SIZE ! UNITS	UNIT	addressing unit
WORDS	WORD	machine word

Again, note that SIZE, UNITS, and WORDS give their answers for variables padded to word boundaries, while BITS and BYTES give minimal answers. Although not symmetric, these definitions are believed to result in the fewest practical problems.

Exponentiation operation

X ** Y is a new expression that returns the number X raised to the Y power. The obvious definition will be used for signed and unsigned arithmetic (note: 0 ** 0 causes a signal). The following Mesa procedures provide correct (although not necessarily efficient) definitions of exponentiation for the various arithmetic classes:

```
CardPower: PROC [x, y: CARD] RETURNS [CARD] = {
    SELECT y FROM
        0 => IF x = 0 THEN ERROR Undefined ELSE RETURN [1];
        1 => RETURN [x];
        ENDCASE =>
            IF y MOD 2 # 0
                THEN RETURN [x * CardPower[x*x, y / 2]]
                ELSE RETURN [CardPower[x*x, y / 2]];
    };

IntPower: PROC [x, y: INT] RETURNS [INT] = {
    SELECT x FROM
        0 => IF y <= 0 THEN ERROR Undefined ELSE RETURN [0];
        1 => RETURN [1];
        -1 => IF y MOD 2 # 0 THEN RETURN [-1] ELSE RETURN [1];
        ENDCASE;
    SELECT y FROM
        < 0 => RETURN [0]
        0 => RETURN [1];
        1 => RETURN [x];
        ENDCASE =>
            IF y MOD 2 = 1
                THEN RETURN [x * IntPower[x*x, y / 2]]
                ELSE RETURN [IntPower[x*x, y / 2]];
    };
```

```
RealPower: PROC [x, y: REAL] RETURNS [REAL] = {
    SELECT x FROM
        0.0 => IF y = 0.0 THEN ERROR Undefined ELSE RETURN [0.0];
        ENDCASE => RETURN [RealFns.Exp[y*RealFns.Ln[x]];
    };
```

Extended precision exponentiation is handled by analogous routines.

> Previous Mesa and Cedar compilers did not support exponentiation. The syntax X**Y was chosen to avoid overloading the dereference (^) operator.

MACHINE CODE

The treatment of MACHINE CODE procedures has changed radically, since the "machine code" for the C target is the C programming language, not the binary code or assembly code for a particular machine. Rather, the syntax of MACHINE CODE procedures has been altered to require a string literal as the body of the procedure. This string literal is interpreted as the name of a C procedure, optionally prefixed by the name of the C file that must be included. As an example, consider the definition of logical AND as provided by Basics.mesa:

```
BITAND: BitOp = TRUSTED MACHINE CODE {
    "Basics.XRM_BITAND" -- AND
    };
```

The name *following the last dot* is the procedure name, and the characters preceding the last dot are taken to be the included file name, which will be passed through to C as:

```
#include <cedar/Basics.h>
```

> As a special case, if the procedure name is empty, and the invocation has no arguments or returns, the invocation is ignored. This allows one to force #include statements without actually having to call a specific procedure.

If the prefix starts with a double quote (") or left angle (<) then the prefix will be used literally in the #include statement emitted (the assumption is that the programmer is asking for something special).

This definition for MACHINE CODE was chosen to enable Cedar/Mesa programs to easily call procedures (and macros) written in C. This definition encourages portable code, since the machine dependencies are hidden from the Mesa code. The compatibility of data types, however, is strictly up to the programmer.

Following the UNIX C semantics, #include <cedar/Basics.h> specifies the file /usr/include/cedar/Basics.h. The existence (and contents) of this and any other files requested by the #include directive is not checked by Mimosa.

There are other extensions to MACHINE CODE covered in a separate document: C2CInterLanguageDoc.tioga.

Zones

The treatment of ZONE and UNCOUNTED ZONE has changed slightly, partly due to the distinction between bitsPerAU and bitsPerWord (assumed to be equal in PrincOps), and partly to make these zones have more commonality within the compiler.

The following definitions for zone implementations are compatible with the definitions assumed by the Mimosa compiler:

```
Type: TYPE = MACHINE DEPENDENT {null (0), last (177777B)};
Zone: TYPE = REF ZoneRecord;
ZoneRecord: TYPE = MACHINE DEPENDENT RECORD [
    new: PROC [self: Zone, units: CARD32, type: Type]
            RETURNS [REF],
    free: PROC [self: Zone, object: REF]
    ];
UZone: TYPE = POINTER TO UZoneRecord;
UZoneRecord: TYPE = MACHINE DEPENDENT RECORD [
    new: UNSAFE PROC [self: UZone, units: CARD32]
            RETURNS [POINTER],
    free: UNSAFE PROC [self: UZone, object: POINTER]
    ];
```

The procedures take a number of addressing units as arguments. For a counted zone:

```
zone.NEW[T]
```

is equivalent in function (but not type) to

```
zone.new[zone, SIZE[T], CODE[T]]
```

and, for an uncounted zone:

```
uzone.NEW[T]
```

is equivalent in function (but not type) to

```
uzone.new[uzone, SIZE[T]]
```

Zone implementors should take care in porting their code to use the right granularity for the units arguments, which must be expressed in addressing units, not words.

Uncounted zone implementors should note that a level of indirection that existed in PrincOps Cedar/Mesa is no longer present in Mimosa. Both counted and uncounted zones have identical positions for their procedures.

The implementors of both counted and uncounted ZONE allocators are warned that the compiler relies on objects returned from such allocators being aligned to the strongest alignment constraints of the target machine. Taking the Sun 4 as an example, the addresses returned by all ZONE allocators should be aligned on 8 byte boundaries. All frame based storage (both local and global) is also required to be based on addresses with the strongest alignment constraints (4 bytes for the Sun 3, 8 bytes for the Sun 4).

This requirement may be reduced to merely ensuring word alignment.

Current restrictions

The Cedar/Mesa language, plus the Mimosa extensions, define a number of features not yet fully

supported. This section describes those features that we intend to support, but do not yet support.

Importing PROGRAM values

This problem is due to some missing information in the object files, and some missing algorithms in the compiler to generate and use that information. The problem is that procedure entry points can no longer be fully specified by index; they must also supply the offset in the global frame, which is not available at a sufficiently early stage in the compilation.

There is no schedule set for this support.

It is recommended that the DIRECTORY clause only mention definitions modules.

Renamed IMPORTS

Renaming imports through the construct

```
IMPORTS X: Defs
```

is not yet supported. If renaming is desired, the renaming available through

```
OPEN X: Defs
```

is recommended.

Discontinued features

The following language features have been eliminated due to their dependence on the PrincOps architecture. These features have also been eliminated from the MesaPort compiler.

POINTER TO FRAME

POINTER TO FRAME cannot be supported for local frames on machines where variables are kept in registers, yet that is a crucial technique for efficient code on many non PrincOps machines.

An acceptable replacement for POINTER TO FRAME is to explicitly construct addressable records in either the local or global frames, and to use pointers to those records in place of POINTER TO FRAME.

PORT

Ports have been eliminated since they do not have efficient implementations for non PrincOps machines. The use of processes is recommended as a replacement.

There are only two instances of PORT in the PrincOps Cedar Nucleus, both of which are used in delicate cases of allocation when a frame cannot be afforded. Neither of these cases is likely to occur for non PrincOps machines, since the same cases of allocation will be handled by software instead of microcode.

STATE

The following features have been eliminated, since they are highly machine dependent, and have no meaning on non PrincOps machines.

var ← STATE
STATE ← var
RETURN WITH var
TRANSFER WITH var

MDSZone

This feature is not supported. The whole concept of MDS is in extreme disfavor.

NEW frame

The dynamic creation of global frames is not supported. The best substitute is to rewrite client code to support dynamically allocated objects. This should normally only involve adding an extra argument to a few procedures (something we have done for the Mimosa compiler).

RESTART, STOP

RESTART and STOP are not supported. If the initialization routine cannot complete its work immediately, it should fork a process to perform the completion.

RESIDENT

RESIDENT is not supported. Similar functionality can be had by either using a MakeBoot style of program or explicit runtime calls to pin data structures and code.

CONDITION timeout fields

The timeout field for condition variables is not supported. Both CONDITION and MONITORLOCK types are completely opaque, and require the use of routines in the Process interface for non standard initialization.

MesaPort compatibility

This section discusses compatibility with the MesaPort compiler. The current intention is for Mimosa to remain compatible with the MesaPort compiler except where it degrades future migrations or introduces significant inconsistencies. Unless otherwise stated, the Mimosa compiler will be compatible with the MesaPort compiler.

This approach may necessitate a source conversion program to be written when converting between the Mimosa and MesaPort variants of the language.

Cedar features

Mimosa supports the Cedar extensions to the Mesa language. There is no mechanism to support subsetting. Where restricted subsets of Cedar/Mesa are desired, separate programs to check for the subsets are recommended.

Arithmetic

Mimosa defines some types to have machine dependent widths, where the width is equal to the "natural" word width of the machine. The MesaPort compiler retains the PrincOps definitions of these types. These types are: INTEGER, CARDINAL (= WORD), and NAT (= NATURAL).

The rationale for using a 32 bit default width on a 32 bit machine is that it typically leads to better code efficiency. For example, if we used 16 bit arithmetic for INTEGER on the Sun machines to try to increase portability, then there would be substantial efficiency penalties for calculations such as:

```
a, b, c: INTEGER;
... a+b = c ...
```

The inefficiency would come from trying to reproduce the modulo 2**16 arithmetic provided by PrincOps on a machine where the arithmetic is naturally performed in 32 bits. Extra masking operations would be required to preserve the PrincOps semantics for arithmetic.

Mimosa barely supports LONG REAL, and instead prefers DREAL. The interpretation of LONG REAL is not consistent with the other uses of LONG, which otherwise promote from bitsPerWord to bitsPerLongWord precision.

Mimosa pays more attention to distinctions between signed and unsigned arithmetic, since there may be different operations for signed and unsigned arithmetic. This may lead differences in bounds checking from PrincOps Mesa. Explicit coercions to a particular type are recommended to resolve any problems in this area.

Record layout

As described above in the "Record alignment and layout" section.

The tilde operator (~)

The use of the tilde character (~) differs between the Cedar and MesaPort compilers. In particular, tilde cannot be used to freely negate the sense of relational operators. Only the following special binary operators using tilde are supported (with no space permitted after the tilde):

```
~=, ~<, ~>
```

The following binary operators, formerly valid, are no longer syntactically correct and are not allowed by the Mimosa compiler:

```
~ IN, ~#, ~<=, ~>=
```

Tilde is still a valid unary operator, where it has the same meaning as NOT.

This restriction is motivated by conflicts between the Cedar compiler syntax and the XDE Mesa syntax. The compromise we have made is believed to affect very few programs.

Miscellaneous changes

This section describes a few minor changes that are not otherwise discussed.

Xerox Character Code

For compatibility with the Xerox Character Code (XCC), the following characters are accepted by the Mimosa compiler:

Code	Printed	Equiv	Meaning
244c	$		dollar sign
253c	«	<<	double left angle
254c	□	:=	□□□□ □□□□□
255c	□		□□ □□□□□
264c	×	*	multiplication
270c	÷	/	division
273c	»	>>	double right angle

At least some of these characters are accepted by the XDE and Cedar 7.0 compilers with the same meaning.

For compatibility with the old character set used for Mesa programs, we have the following equivalences for character codes.

New	Old	Printed	Meaning
244c	044c	$	dollar sign
254c	137c	_	left arrow
255c	136c	^	up arrow

These translations do not occur inside of string constants.

Newline character

The newline character (\n) is defined to be target dependent. For example, the PrincOps newline character is 015C, while the Sun target newline character is 012C. It is recommended that portable programs accept either code as line terminator.

Syntax

The ":=" token is everywhere equivalent to the "□" token.

Opaque types

Opaque types with explicit lengths have their lengths specified in addressing units rather than words. This allows portable code to be written using the SIZE operator. For example, for an opaque type George that must be large enough to contain a LONG POINTER and a CARD, the declaration of the type would be:

```
George: TYPE [SIZE[LONG POINTER]+SIZE[CARD]];
```

It is strongly recommended that no type be exported that does not fit in an integral number of words.

String literals

To aid in calling C routines, all STRING (and ROPE and REF TEXT) literals generated by Mimosa are null terminated, and the maxlength is rounded up so that an integral number of words is occupied

(the length reflects only the number of characters explicitly in the literal). For the Sun target, the null termination will add one word (all zero bytes) to the string in one case out of four, otherwise the null character comes about from the roundup of the length. When the length is rounded up, there is no real space penalty, since Mesa string bodies need to start on word boundaries anyway, and there isn't anything useful that can be done with the leftover bytes.

For the literal "Hello", the length is 5, the maxlength is 8, and the contents (but not the length) are the same as if one had used the literal "Hello\000\000\000". In the case of "Four", the length is 4, the maxlength is 8, and the contents are the same as "Four\000\000\000\000".

Of course, the compiler only makes this guarantee for literals. If the string is dynamically constructed the compiler has no influence over the contents. Also, the compiler makes no guarantees about the character contents of string bodies that are created by:

```
foo: STRING □ [N];
```

The expression foo+SIZE[StringBody[N]] is usually ill advised, since it computes the address of the word containing the character indexed by N, not the address of the character itself.

Argument records

The semantics for field defaults in argument records has been changed slightly. In particular, omitting arguments that have no defaults is not permitted. For example, consider the procedure:

```
P: PROC [ptr: LONG POINTER TO CARD, c: CARD] = {
    IF ptr□ # c THEN ERROR;
    };
```

In the XDE Mesa and Cedar 7.0 compilers, a call of the form P[,X] is permitted. In Mimosa, such a call is not permitted (an error message is generated). This change is believed to improve error checking without affecting existing correct programs.

Potentially surprising optimizations

This section describes a few minor changes that are not otherwise discussed.

String literals

The compiler is permitted to treat certain expressions involving string literals as constants even when the runtime behavior is not strictly constant. For example, the compiler is permitted to treat ("abc").length as being equivalent to the constant 3. As another example, the compiler is permitted to treat ("abc")[1] as the constant 'b. The compiler is not permitted to treat local string literals as constant (as in "abc"L). Programs that assume the mutability of string literals are strongly discouraged.

Bounds checking

The compiler believes the programmers's assertions about bounds. This leads to a number of optimizations that may be surprising when combined with a program that has uninitialized variables. For example, in the procedure:

```
Check: PROC [x: [0..7]] = {
    IF x > 7 THEN ERROR;
```

```
};
```

The compiler is permitted to determine that the declared bounds of x allow the comparison to be determined at compilation time, so at execution time there is no check, and no possibility of an exception being raised.

A similar lack of bounds checking may occur in array access. If the index to an arry is determined to be within the index type of an array then there will be no bounds checking for that access, regardless of the state of the bounds checking option.

This behavior is good whenever x has a well formed value. However, if x is passed an uninitialized value, or assigned using LOOPHOLE, then this optimization might avoid a check that would otherwise produce the error. The programmer is hereby warned.

Addressability rationale and discussion

Mimosa has been required to support the current Cedar/Mesa language without significant loss of processing efficiency. In addition, the ability to write portable code has been judged important by both the designers and the projected customers. Relatively few compromises have been necessary, but the change from a 16 bit word size to a 32 bit word size has resulted in a semantics that at least a few people have found surprising.

In Mimosa, most variables are padded to a word boundary when necessary. These include simple variables, unpacked record components, unpacked array and sequence elements, and dereferenced pointers. Some variables are declared to be packed, and occupy a machine dependent amount of storage. These include packed record components and packed array and sequence elements. Machine dependent records are packed to even tighter boundaries, either implicitly or explicitly given by the programmer. The above statement is also true for product XDE Mesa and for Cedar 7.0. The main differences introduced by Mimosa are that the word size is now 32 bits rather than 16 bits, and the ability to specify that a record is packed (with meaning similar to that of packed arrays). Earlier in the development of Mimosa there was an attempt to relax the need for word padding, but that attempt failed due to two reasons: first, the difficulty of changing the way that the compiler allocated storage for variables, and second, the desire for efficiency.

In the following discussion, assume that we have the following definitions:

```
Index: TYPE = [0..16);
Rec16: TYPE = RECORD [a: CARD16, b: INT16];
Array16: TYPE = PACKED ARRAY Index OF INT16;
PackedRec16: TYPE = PACKED RECORD [a: CARD16, b: INT16];
PackedArray16: TYPE = PACKED ARRAY Index OF INT16;
Ptr16: TYPE = POINTER TO INT16;
uArray: Array16;
pArray: PackedArray16;
uRec: Rec16;
pRec: PackedRec16;
localVar: INT16;
```

The Mimosa compiler pads the components of Rec16 and Array16 variables to a full word (therefore, uArray occupies 16 32 bit words, and uRec occupies 2 words). Further, the components can be addressed, with the address having type Ptr16. Also, variables such as localVar

can be addressed. For all padded variables of type INT16 the padding extends the sign bit. In other cases the padding uses zero bits.

For packed records and arrays, the Mimosa compiler may pad the components as well, but usually to tighter boundaries. This packing is machine dependent, and is intended to be a compromise between access efficiency and space efficiency. It is normally true that the packing for arrays is to an element size that is a power of 2 bits for subword quantities (usually one of 1, 2, 4, 8, 16, 32 bits per element). The strategy used for record packing depends on the machine, but is typically as tight as feasible without having fields cross word boundaries (unless those fields are multiple words, in which case they are word aligned). For the Sun target, pArray occupies 8 32 bit words, and pRec occupies 1 word.

The requirement that addressable variables be padded to word boundaries and be aligned on word boundaries stems from the potential ambiguity of pointers. Is the referent of the pointer padded or not? Since access via pointers must generate code for either the packed case or the padded case the compiler must adopt a consistent semantics. Mimosa has chosen a padded semantics for addressable variables. This allows localVar to be addressable, with the semantics and storage layout as if it were not addressable.

Suppose that Mimosa chose instead a packed semantics for addressable variables. To preserve efficient access for most (but not all) machines, this choice leads to the requirement that padding is only performed up to a byte, halfword, or word boundary (the least boundary required to hold the variable) with a similar alignment constraint. This is potentially a consistent view, but it can lead to inefficient code, and also to code with reduced portability.

Machines such as the Sun 3 (68020 processor) and Sun 4 (Sparc processor) have the ability to access 16 bit and 8 bit quantities in memory (but not in registers) with efficiency equivalent to 32 bit word access. Machines such as the Am29000 must expend extra instructions to do so. Even on the 68020, which supports 32 bit access on 16 bit boundaries, unaligned access is less efficient, and unaligned access on the Sparc is not supported. Therefore, portable code that needs to be efficient should not depend on addressing of subword variables, nor on the ability to access unaligned variables.

The addressing constraints affect storage efficiency for records and arrays only when it is necessary to form a pointer to an internal component of the record or array. This is believed to be a very small percentage of all records and arrays. Since current Mesa does not support byte addressing, no current Mesa programs rely on addressing of bytes. Many Mesa programs rely on the addressability of words, and this is preserved. Unfortunately, a few Mesa programs are believed to rely on the addressability of 16 bit words, and this is not preserved. Therefore, there may be a conversion cost associated with the addressing constraints.

The addressing constraints may also make it awkward to write Mesa code that directly calls externally defined procedures written to use data formats that do not obey the Mesa constraints. In most cases the Mimosa compiler permits writing declarations that describe the external data formats. The machine dependent addressing that occasionally needs to be performed can be handled by special routines written in the external language (a preferred alternative), or by relying on the behavior of the Mimosa compiler for the specific target (dangerous, sometimes desirable, never strictly necessary).

There is an additional motivation to adopt padded rather than packed addressability, which is to maintain consistency with the current XDE Mesa and Cedar 7.0 compilers. The Mimosa compiler is descended from the Cedar 7.0 compiler, which has a common ancestor with the XDE Mesa compiler. An earlier attempt to relax the constraints of addressability failed because the necessary structure was not present in the compiler. For packed semantics the layout of simple variables and

access to simple variables must be made dependent on whether or not the variable is addressed. Also, the layout of records and arrays must always assume that the components are addressable. The effect of these changes to the compiler would have been a delayed compiler.

Our CedarPort experience suggests that the addressability constraints are not very costly. There are few instances of data that falls naturally into the realm of packed arrays of 16 bit values, and even fewer instances where forming the address of an element is useful.

In communicating with C and Unix programs it is also infrequent to have difficulties with addressing. One case that comes up several times, but has a simple solution, is when the Unix routine expects the address of a character (char *) and uses packed semantics to modify the referent, as opposed to Mesa's padded semantics. A recommended declaration to use is:

```
CharsIndex: TYPE = [0..BITS[WORD]/BITS[CHAR]);
WordAsChars: TYPE = PACKED ARRAY CharsIndex OF CHAR;
word: WordAsChars;
status: CHAR;
CallUnixRoutine[..., @word, ...];
status ← word[0];
```

The above code works for any supported size of word or character, and for either byte order (e.g. 68020 or 80386). The code is not as attractive as the more intuitive (but incorrect) use of

```
status: CHAR;
CallUnixRoutine[..., @status, ...];
```

Despite the inelegant syntax, the effort required of the programmer is small. The potential benefit from promoting portable code is large.

Curiouser and curiouser

This section tries to give some advice on features that people have found surprising.

INT16 and LOOPHOLE

The type INT16 is intended to be useful to those converting from PrincOps Mesa to portable Mesa. This type is considered to be a special unbiased subrange of INT. INT16 should be be used when a signed quantity is desired that is exactly 16 bits in length. This type is especially useful in packed records and arrays.

For efficiency, when an INT16 value is stored in a padded variable it is padded to the word width of the machine (typically, but not always, 32 bits). This avoids a certain amount of sign extension, particularly when an INT16 value is held in a register, or compared against an INT value.

The philosophy on LOOPHOLE is that it should take place without performing conversion operations. Its use is an implied statement that the programmer knows something about the equivalence between types. A special case in the compiler, however, recognizes LOOPHOLE[X, INT16] as requiring sign extension when the result is assigned to a variable larger than 16 bits, even if the expression X was already large enough. This is done to relieve the programmer of the task of performing sign extension explicitly.

Bounds checking and NIL checking (AR 17)

If an inline procedure is defined in an interface and "called" in a program module, NIL checking (or bounds checking) is generated for its object code if the nil check (bounds check) switch is turned on during compilation of the program module. The nil check (bounds check) switch has no direct impact on interface compiles.

String constants in inline procedures (AR 18)

Mimosa allows the use of string constants in inline procedures declared in interfaces (unlike PrincOps Mesa). Such string constants can be declared to be local (eg "foo"L;), but this declaration is ignored since there is no local frame to associate with the string constant. All such string literals are allocated in the global frame of the module which imports the inline procedure.

STARTing a module after it has been started (AR 20)

STARTing a module after it has already been started, either explicitly or by start trap has no effect.

Variant record constructors (AR 21 and 25)

A variant record containing a variant of the form string[s: STRING] when assigned to using the constant constructor "[string[NIL]]" results in writing more storage than "SIZE[string foo]". Programmers are strongly encouraged to use the NEW (or z.NEW) operator to allocate variant records to avoid these problems.

The constructor was assigned to a variable with type Foo and was padded to have a size of SIZE[foo]; however, only SIZE[string foo] had been allocated by the programmer.

If pt1 is a pointer to a bound variant that is shorter than the maximum length variant, aqpt is an unbound variant whose value is a variant that is longer than pt1□, and pt1□ □ aqpt is executed, data stored after pt1□ is smashed.

MLM Typos (AR 22 and 27)

On page 7-15 of the 12.0 MLM (Sept 85), "langauge" is misspelled in " The langauge requires only..."

On page 5-7 of the 12.0 MLM, on the line after "Younger", there's a line ending in "...]] END;" which appears to have one too many right brackets.

Implementation dependency in overlapping record copy (AR 23)

The result of ptr1□ □ ptr2□ is undefined if the two records overlap and ptr1 > ptr2 (ie the semantics of overlapping record copys are machine dependent and should not be relied upon by programmers).

Definition of ABS (AR 24)

The MLM should do a better job of defining what ABS[foo] returns. (See 12.0 MLM pg 2-13). Currently the MLM reads:

The built in function ABS computes the absolute value of its argument.

For Mimosa it should read:

The built in function ABS computes the absolute value of its argument; the type of the return value of ABS is equivalent to the argument type except as specified in the table below:

Argument	Return
INTEGER	CARDINAL
INT	CARD
DINT	DCARD
INT16	CARD16

Semantics of REJECT (AR 29)

Chapter 8 of the MLM should make it clear that if a catch phrase assigns new values to the arguments of a SIGNAL, then REJECTs the signal, future catch frames do not see the changes it made to the signal's arguments. Any such assignment affects only that one catch frame. That is, all catch phrases for a signal receive the same arguments, regardless of any actions taken by previous catch phrases which rejected the signal.

Automatic dereference of pointer to CONDITION (AR 32)

While WAIT will automatically dereference a pointer to CONDITION, NOTIFY and BROADCAST will not. This should be documented in the fine point of pg 9-11. (Currently it doesn't have anything to say on this except for the explicit dereference in the CondProc example on page 9-11 of the 12.0 MLM).

Record constructors for return clauses (AR 286)

Using record constructors for ReturnsClauses can be confusing. For example the following test case will not compile:

```
tmp1: PROGRAM = BEGIN

    LongDivMod: PROCEDURE [num: CARD32, den: CARD16]
        RETURNS [quotient, remainder: CARD16] = INLINE {
        quotient ← num / den;
        remainder ← num MOD den; };

    DIVMOD: PROCEDURE [num, den: CARD16] RETURNS [quotient, remainder: CARD16] =
        INLINE { RETURN[LongDivMod[num, den]] };

    END.
```

The compiler issues the error message:

```
1 too few elements in list, at DIVMOD[296]:
    INLINE { RETURN[LongDivMod[num, den]] };

LongDivMod[num, den]  has incorrect type (expected ...), at DIVMOD[296]:
```

```
INLINE { RETURN[LongDivMod[num, den]] };
```

But the following the test will compile:

```
tmp2: PROGRAM =
  BEGIN

  LongDivMod: PROCEDURE [num: CARD32, den: CARD16]
    RETURNS [quotient, remainder: CARD16] = INLINE {
    quotient ← num / den;
    remainder ← num MOD den; };

  DIVMOD: PROCEDURE [num, den: CARDINAL] RETURNS [quotient, remainder: CARDINAL]
    =
    { RETURN LongDivMod[num, den] }; -- brackets removed

  END.
```

The confusion is over the use of record constructors for ReturnsClauses - since LongDivMod[num, den] results in a record constructor, the extra set of brackets acts as a record constructor of a record constructor - which is not the desired effect. Section 5.1 of the MLM should include this example and explanation.

Indirection on PACKED ARRAYs of WORD16s (AR 298)

In the test case:

```
MesaImpl: PROGRAM  = BEGIN

  Unspec: TYPE = MACHINE DEPENDENT RECORD [
    u: UNSPEC16
    ];

  first: PACKED ARRAY [0..0] OF Unspec;

  firstPosition: DESCRIPTOR FOR PACKED ARRAY OF Unspec ← DESCRIPTOR [first];

  END.
```

The compiler generates the error message:

```
first  does not allow indirect reference, at MesaImpl[179]:
firstPosition: DESCRIPTOR FOR PACKED ARRAY OF Unspec ← DESCRIPTOR [first];
```

The reason that the compiler says that "first does not allow indirect reference" is that the array is packed and the array size is not an integral number of words. This is the same restriction imposed by the PrincOps Cedar and Mesa compilers, only the word size is different for Mimosa.

TRASH as a return value (AR 480)

In the test case:

```
Test: PROGRAM = {
    Proc: PROCEDURE[] RETURNS[a:INTEGER, b:CARDINAL] = {
        RETURN[a: 4, b:TRASH];
        };
    }.
```

The compiler generates the error message:

```
b  cannot be defaulted or voided, at Proc[75]:
    RETURN[a: 4, b:TRASH];
```

Disallowing TRASH as a return value (or argument value) is an attempt to discourage nasty argument or return values. This was motivated by the problem with the implicit form "P[a,]", where the second argument is voided. This is frequently a bug, rather than the intended behavior, so the compiler was changed to detect it. It also detects the explicit form using TRASH. Programmers are strongly encouraged to use explicit return values.

Synonym for ARRAY CARDINAL (AR 34)

The type constructor:

ARRAY OF *T*

is equivalent to:

ARRAY CARDINAL OF *T*

For the SPARC target, arrays declared using this construct exceed the normal addressing limits, which is OK until you try to have one as a variable or assign one, at which point the compiler will complain.

This construct is a relic from the old Mesa compiler, and its use is not recommended.

C/Mesa Language Changes

Two language constructs, "STATIC REQUESTS" and "DYNAMIC REQUESTS", have been added to C/Mesa to facilitate building multiple language configurations and loading on demand. Both constructs are added to the C/Mesa file after the EXPORTS clause, and each is followed by a list of filenames.

For static requests, Cinder puts the filenames in the "ld" command of the ".ld" file; this allows non Mesa derived object files to be linked into the configuration. Static Requests should be used to link a module written in C into a configuration, for instance.

Examples

```
Foo: CONFIG
    IMPORTS ...
```

```
EXPORTS ...
STATIC REQUESTS "foo.o", ...
DYNAMIC REQUESTS "/usr/local/lib/cedar/BarPackage", ...
CONTROL ...
```

When this configuration is sent through Cinder, "foo.o" will appear in the link command in Foo.ld. "XR_request(/usr/local/lib/cedar/BarPackage") will appear in Foo.c.

CONDITION variable access rules (AR 509)

The MLM claims that one ought to be able to access the timeout field of a CONDITION variable with simple assignment (4.1.2). Mimosa changes this to make these types opaque, requiring programers to use runtime routines (specifically routines exported by the Process interface) to update these fields.

The following test case will compile with the XDE and Cedar 7.0 compilers but not with Mimosa:

```
ConditionTest: PROGRAM = {
    DataRecord: TYPE = RECORD [
        blockCondition: CONDITION □ [timeout: 100]
        ];
    }...
```

giving the error message:

```
timeout  is not valid as a key, at ConditionTest[85]:
    blockCondition: CONDITION □ [timeout: 100]
```

Collected Cedar/Mesa Grammar (AR 16 and 37)

Cedar/Mesa is a living language which has undergone many changes since its initial implementations. Extensions and refinements continue to be made. Consequently, the BNF which is found in the Mesa language may not reflect the current state of Cedar/Mesa syntax. The BNF grammar given below is the grammar currently used by the parser of the Mimosa compiler. This technique has two advantages: the the parsing tables are smaller, and in a number of cases it is easier for the compiler to give informative messages about "semantic" restrictions than to recover correctly from a syntax error.

```
goal                ::= . module .
    | . module ..

module              ::= directory identlist cedar proghead trusted checked block
    | directory identlist cedar defhead defbody

includeitem         ::= id : FROM string using
    | id : TYPE using
    | id using
```

```
    | id : TYPE id using

cedar               ::= CEDAR
    |

proghead            ::= resident safe class arguments locks interface bindop public

resident            ::=

defhead             ::= definitions locks imports shares bindop public

definitions         ::= DEFINITIONS

defbody             ::= BEGIN open declist END
    | BEGIN open declist ; END
    | { open declist }
    | { open declist ; }

locks               ::= LOCKS primary lambda
    |

lambda              ::= USING ident typeexp
    |

moduleitem          ::= id
    | id : id

declaration         ::= identlist public entry readonly typeexp initialization
    | identlist public TYPE bindop public typeexp default
    | identlist public TYPE optsize

public              ::= PUBLIC
    | PRIVATE
    |

procaccess          ::=

entry               ::= ENTRY
    | INTERNAL
    |

idlist'             ::= id
    | id , idlist'

identlist'          ::= id :
    | id , identlist'
    | id position :
    | id position , identlist'

position            ::= ( exp optbits )
```

```
optbits          ::= : bounds
    |

interval         ::= [ bounds ]
    | [ bounds )
    | ( bounds ]
    | ( bounds )

typeexp          ::= id
    | typeid
    | typecons

range            ::= id
    | id interval
    | typeid interval
    | interval
    | typeid

typeid'          ::= id . id
    | typeid' . id

typeappl         ::= typeappl . id
    | id length
    | typeid length
    | typeappl length

typeid           ::= id id
    | id typeid
    | typeid'

typecons         ::= interval
    | id interval
    | typeid interval
    | dependent { elementlist }
    | dependent monitored RECORD reclist
    | ordered base pointertype
    | VAR typeexp
    | REF readonly typeexp
    | REF readonly ANY
    | REF
    | LIST OF readonly typeexp
    | PACKED typecons
    | MSBIT typecons
    | LSBIT typecons
    | NATIVE typecons
    | WORD8 typecons
    | WORD16 typecons
    | WORD8
    | WORD16
    | WORD32
    | WORD64
```

```
    | WORD32 typecons
    | WORD64 typecons
    | ARRAY indextype OF typeexp
    | DESCRIPTOR FOR readonly typeexp
    | safe transfermode arguments
    | id RELATIVE typeexp
    | typeid RELATIVE typeexp
    | heap ZONE
    | LONG typeexp
    | FRAME [ id ]
    | id PAINTED typeexp
    | typeid PAINTED typeexp
    | typeappl

ident           ::= id position :
    | id :

element         ::= id ( exp )
    | ( exp )
    | id

variantpart     ::= PACKED variantpart
    | SELECT vcasehead FROM variantlist ENDCASE
    | SELECT vcasehead FROM variantlist , ENDCASE
    | SEQUENCE vcasehead OF typeexp

safe            ::=
    | UNSAFE
    | SAFE

arglist         ::= ANY
    | fieldlist
    |

returnlist      ::= RETURNS ANY
    | RETURNS fieldlist
    |

monitored       ::= MONITORED
    |

dependent       ::=
    | MACHINE DEPENDENT

reclist         ::= [ ]
    | NULL
    | [ pairlist ]
    | [ typelist ]
    | [ pairlist , variantpair ]
    | [ variantpart default ]
    | [ variantpair ]
```

```
variantpair        ::= identlist public variantpart default

vcasehead          ::= ident public tagtype
    | COMPUTED tagtype
    | OVERLAID tagtype

pairitem           ::= identlist public typeexp default

tagtype            ::= *
    | typeexp

variantitem        ::= idlist => reclist

typelist           ::= id
    | typecons default
    | typeid default
    | id  defaultopt
    | typecons default , typelist
    | typeid default , typelist
    | id , typelist
    | id  defaultopt , typelist

pointertype        ::= pointerprefix
    | pointerprefix TO readonly typeexp

transfermode       ::= PROCEDURE
    | PROC
    | PORT
    | SIGNAL
    | ERROR
    | PROCESS
    | PROGRAM

initialization     ::=
    |  initvalue
    | bindop initvalue

initvalue          ::= procaccess trusted checked inline entry block
    | CODE
    | procaccess trusted checked MACHINE CODE BEGIN codelist END
    | procaccess trusted checked MACHINE CODE { codelist }
    | trash
    | exp

trusted            ::=

codelist           ::= orderlist
    | codelist ; orderlist

statement          ::= IF exp THEN statement
```

```
    | IF exp THEN balstmt ELSE statement
    | casehead casestmtlist ENDCASE => statement
    | basicstmt

balstmt          ::= IF exp THEN balstmt ELSE balstmt
    | casehead casestmtlist ENDCASE => balstmt
    | basicstmt

basicstmt        ::= lhs
    | lhs □ exp
    | [ explist ] □ exp
    | trusted checked block
    | casehead casestmtlist ENDCASE
    | forclause dotest DO scope doexit ENDLOOP
    | EXIT
    | LOOP
    | GOTO id
    | GO TO id
    | RETURN optargs
    | transfer lhs
    | free [ exp optcatch ]
    | WAIT lhs
    | ERROR
    | STOP
    | NULL
    | RESUME optargs
    | REJECT
    | CONTINUE
    | RETRY

block            ::= BEGIN scope exits END
    | { scope exits }

scope            ::= open enables statementlist
    | open enables declist ; statementlist

binditem         ::= exp
    | id ~~ exp
    | id : exp

exits            ::= EXITS exitlist
    |

casestmtitem     ::= caselabel => statement

caseexpitem      ::= caselabel => exp

exititem         ::= idlist => statement

casetest         ::= optrelation
    | ~ optrelation
```

```
    | exp

caselabel          ::= ident typeexp
    | caselabel'

controlid          ::= ident typeexp
    | id

forclause          ::= FOR controlid □ exp , exp
    | FOR controlid direction IN range
    | THROUGH range
    |

direction          ::=
    | DECREASING

dotest             ::= UNTIL exp
    | WHILE exp
    |

doexit             ::=
    | REPEAT exitlist
    | REPEAT exitlist FINISHED => statement
    | REPEAT exitlist FINISHED => statement ;

enables            ::= ENABLE catchcase ;
    | ENABLE catchany ;
    | ENABLE BEGIN catchlist END ;
    | ENABLE { catchlist } ;
    |

catchlist          ::= catchhead catchcase
    | catchhead
    | catchhead catchany
    | catchhead catchany ;

catchcase          ::= lhslist => statement

optargs            ::= [ explist ]
    |
    | lhs

transfer           ::= SIGNAL
    | ERROR
    | RETURN WITH ERROR
    | START
    | RESTART
    | JOIN
    | NOTIFY
    | BROADCAST
```

```
keyitem          ::= id ~ optexp
    | id : optexp

defaultopt       ::= trash
    | exp 'l trash
    |
    | exp

optexp           ::= trash
    | exp
    |

exp              ::= IF exp THEN exp ELSE exp
    | casehead caseexplist ENDCASE => exp
    | lhs □ exp
    | [ explist ] □ exp
    | ERROR
    | transferop lhs
    | disjunct

disjunct         ::= disjunct OR conjunct
    | conjunct

conjunct         ::= conjunct AND negation
    | negation

negation         ::= ~ relation
    | NOT relation
        | relation

relation         ::= sum optrelation
    | sum

sum              ::= sum addop product
    | product

product          ::= product multop factor
    | factor

optrelation      ::= NOT relationtail
    | relationtail

relationtail     ::= IN range
    | relop sum

relop            ::= =
    |
    | <
    | <=
    | >
    | >=
```

```
addop           ::= +
    | -

multop          ::= *
    | /
    | MOD

factor          ::= primary ** factor
    | addop primary
    | primary

primary         ::= [ explist ]
    | prefixop [ orderlist ]
    | VAL [ orderlist ]
    | ALL [ orderlist ]
    | new [ typeexp initialization optcatch ]
    | cons [ explist optcatch ]
    | listcons [ explist ]
    | NIL
    | typeop [ typeexp ]
    | BITS [ typeexp ]
    | BITS [ typeexp , exp ]
    | BYTES [ typeexp ]
    | BYTES [ typeexp , exp ]
    | UNITS [ typeexp ]
    | UNITS [ typeexp , exp ]
    | SIZE [ typeexp ]
    | SIZE [ typeexp , exp ]
    | WORDS [ typeexp ]
    | WORDS [ typeexp , exp ]
    | ISTYPE [ exp , typeexp ]
    | @ lhs
    | DESCRIPTOR [ desclist ]
    | lhs

qualifier       ::= . prefixop
    | . typeop
    | . BITS
    | . BYTES
    | . UNITS
    | . SIZE
    | . WORDS
    | [ explist optcatch ]
    | . id
    |

lhs             ::= id
    | num
    | string
    | lnum
```

```
    | flnum
    | char
    | lstring
    | atom
    | NARROW [ exp opttype optcatch ]
    | LOOPHOLE [ exp opttype ]
    | APPLY [ exp , exp optcatch ]
    | ( exp )
    | lhs qualifier

optcatch          ::= ! catchlist
    |

transferop        ::= SIGNAL
    | ERROR
    | START
    | JOIN
    | NEW
    | FORK

prefixop          ::= LONG
    | ABS
    | PRED
    | SUCC
    | ORD
    | MIN
    | MAX
    | BASE
    | LENGTH

typeop            ::= CODE
    | FIRST
    | LAST
    | NIL

desclist          ::= exp , exp opttype
    | exp

directory         ::= DIRECTORY ;
    | DIRECTORY includelist ;
    |

imports           ::= IMPORTS
    | IMPORTS modulelist
    |

pointerprefix     ::= POINTER
    | POINTER interval

free              ::= FREE
    | lhs . FREE
```

```
new                 ::= NEW
    | lhs . NEW

cons                ::= CONS
    | lhs . CONS

listcons            ::= LIST
    | lhs . LIST

exports             ::= EXPORTS
    | EXPORTS modulelist
    |

fieldlist           ::= [ ]
    | [ pairlist ]
    | [ typelist ]

using               ::= USING [ ]
    | USING [ idlist ]
    |

declist             ::= declaration
    | declist ; declaration

elementlist         ::=
    | elementlist'

pairlist            ::= pairitem
    | pairlist , pairitem

variantlist         ::= variantitem
    | variantlist , variantitem

orderlist           ::= optexp
    | orderlist , optexp

casestmtlist        ::=
    | casestmtlist'
    | casestmtlist' ;

statementlist       ::=
    | statementlist'
    | statementlist' ;

exitlist            ::=
    | exitlist'
    | exitlist' ;

catchhead           ::=
    | catchhead catchcase ;
```

lhslist ::= lhs
| lhslist , lhs

caseexplist ::=
| caseexplist'
| caseexplist' ,

includelist ::= includeitem
| includelist , includeitem

modulelist ::= moduleitem
| modulelist , moduleitem

elementlist' ::= element
| elementlist' , element

bindlist ::= binditem
| bindlist , binditem

statementlist' ::= statement
| statementlist' ; statement

casestmtlist' ::= casestmtitem
| casestmtlist' ; casestmtitem

caselabel' ::= casetest
| caselabel' , casetest

exitlist' ::= exititem
| exitlist' ; exititem

keylist ::= keyitem
| keylist , keyitem

caseexplist' ::= caseexpitem
| caseexplist' , caseexpitem

identlist ::= identlist'

open ::= OPEN bindlist ;
|

idlist ::= idlist'

explist ::= orderlist
| keylist

class ::= PROGRAM
| MONITOR

```
casehead          ::= SELECT exp FROM
    | WITH binditem SELECT optexp FROM

readonly          ::= READONLY
    |

ordered           ::= ORDERED
    |

base              ::= BASE
    |

heap              ::= UNCOUNTED
    |

inline            ::= INLINE
    |

arguments         ::= arglist returnlist

interface         ::= imports exports shares

bindop            ::= ~
    | =

shares            ::= SHARES idlist
    |

default           ::=  defaultopt
    |

optsize           ::= [ exp ]
    |

bounds            ::= exp .. exp

length            ::= [ exp ]

indextype         ::= typeexp
    |

catchany          ::= ANY => statement

trash             ::= TRASH
    | NULL

opttype           ::= , typeexp
    |

checked           ::=
    | CHECKED
```

| TRUSTED
| UNCHECKED