Chapter 1. Introduction

This manual describes the Cedar language. It is organized into three major parts:

Chapter 2: A description of a much simpler kernel language, in terms of which the Cedar language is explained. This description includes a precise definition (¶ 2.2 formal explanation of the ideas of the kernel and the restrictions imposed by curr (¶¶ 2.3-2.9). ¶ 2.1 contains an overview or glossary, in which the major technical in the kernel are briefly defined.

Chapter 3: The syntax and semantics of the current Cedar language. The semantics i precisely by a desugaring into the kernel. It is also given more informally by Eng This chapter also contains a number of examples to illustrate the syntax.

Chapter 4: The primitive types and procedures of Cedar. For each one, its type is well as an English definition of its meaning. This chapter is organized according hierarchy of the primitive types (¶ 4.1).

In addition, there is a one-page grammar, and a two-page language summary which includes grammar, the desugaring, and the examples from Chapter 3,

The document you are reading is a draft. Most of Chapter 2 is missing, an of other sections are incomplete.

Error reports and comments on the presentation are most welcome.

Table of Contents

Chapter 4. Primitive types and type constructors

Tables

Chapter 2. The kernel language

This document describes the Cedar language in terms of a much simpler kernel language. C
differs from the kernel in two ways:

- It has a more elaborate syntax. The meaning of each construct in Cedar is explaine
  giving an equivalent kernel program.

  Often the kernel program is longer or less readable; the Cedar construct can be thought of as an idiom
  conveniently expresses a common operation. Sometimes the Cedar construct has no real advantage, and the
  difference is the result of backward compatibility with the ten-year history of Mesa and Cedar.

- It has a large number of built-in types and procedures. In the kernel language all
  could in principle be programmed by the user, though in fact most are provided by
  code in the Cedar compiler. In general, you can view these built-in facilities muc
  library, selecting the ones most useful for your work and ignoring the others.

Unfortunately, the kernel language is not a subset of the current Cedar langauge. Many i
objects (notably types, declarations and bindings) which are ordinary values in the kern
that can be freely passed as arguments or bound to variables, are subject to various res
Cedar: they can only be written in literal form, cannot be arguments or results of proce
whatever. The long-term goal for evolution of the Cedar language is to make it a superse
kernel defined here. In the meantime, however, you should view the kernel as a concise a
hopefully clear way of describing the meaning of Cedar programs. To help in keeping the
and current Cedar separate, keywords and primitives of the kernel which are not availabl
current Cedar are written in SANS-SERIF SMALL CAPITALS, rather than the ROMAN SMALL CAPITALS
used for keywords of current Cedar. Operator symbols of the kernel which are not in curr
are not on the keyboard.

The kernel is a distillation of the essential properties of the Cedar language, not an e
invention. Most Cedar constructs have simple translations into the kernel. Those which d
many of the features of OPEN) are considered to be mistakes, and should be avoided in new
programs.

¶ 2.2 defines the syntax and semantics of the Cedar kernel language, the former with a g
and the latter by explaining how to take a program and deduce the function it computes a
state changes it causes. The remainder of the chapter explains the concepts behind the k
also gives the restrictions imposed by the current Cedar language on the full generality
here; for more on this subject, see Chapter 3. The meaning of the various built-in primi
given in Chapter 4. ¶ 2.9 describes the incompatibilities between the kernel language an
Cedar, i.e., the constructs in Cedar which would have a different meaning in a kernel pr
the most part, these are bits of syntax which do not have consistent meanings in current
future evolution of the language will replace them with their kernel equivalents.

Usually, terms are defined and explained before they are used, but some circularity seem
unavoidable. ¶ 2.1 gives a brief summary of each major idea which may be helpful as a re
Both this and the explanations in ¶¶ 2.3-2.7 are given under five major headings, as fol

Doing computations: Application Value Variable Group Binding Argument

The type system: Type Type-checking Mark Cluster Declaration

Programs: Name Expression Scope Constructors Recursion

Conveniences: Coercion Exception Finalization Safety Process

Miscellaneous: Allocation Static Pragma

The kernel definition in ¶ 2.2 follows the ordering of the kernel grammar.

2.1 Overview

This section gives a brief summary of the essential concepts on which the Cedar language
The explanations are concise and incomplete. For more precise definitions, see ¶ 2.2.

2.1.1 Doing computations

Application: The basic mechanism for computing in Cedar is applying a procedure (proc fo
arguments. When the proc is finished, it returns some results, which can be discarded or
arguments to other procs. The application may also change the values of some variables.
program an application is denoted by (the denotation of) the proc followed by square bra
enclosing (the denotation of) the arguments: f [x, y]. There are special ways of writing
of application: x+1, person.salary, IF x<3 THEN red ELSE green, x: INT_7.

Value: An entity which takes part in the computation (i.e., acts as a proc, argument or
called a value. Values are immutable: they are not changed by the computation. Examples:
"Hello", l x IN x+3; actually these are all expressions which denote values in an obvious

Variable: Certain values, called variables, can contain other values. The value containe
variable (usually called the value of the variable) can change when a new value is assig
variable. In addition to its results, a proc may have side-effects by changing the value
Every type has a NEW proc which creates a variable of the type. A variable is usually rep
a single block of storage; the bits in this block hold the representation of its value.

Group: A group is an ordered set of values, often denoted like this: [3, x+1, "Hello"]. A
itself a value.

Binding: A binding is an ordered set of [name, value] pairs, often denoted by a construc
[x: INT~3, y: BOOL~TRUE], or simply [x~3, y~TRUE]. Sometimes it is called an environment. If b is a
binding, b.n denotes the value of the name n in b.

Argument:

Incomplete

2.1.2 The type system

Type: A type defines a set of values by specifying certain properties of each value in t
integer between 0 and 10); these properties are so simple that the compiler can make sur
arguments have the desired properties. A value may have many types; i.e., it may be in
these sets. A type also collects together some procs for computing with the value (e.g.,
multiply).

A type is itself a value which is a pair:

- Its predicate, a function from values to the distinguished type BOOL. A value has t
  T's predicate returns TRUE when applied to the value.
- Its cluster, a binding in which each value is usually a proc taking one argument o
  The expression e.f denotes the result of looking up f in the cluster of e's syntact
  and applying the resulting proc to e.

A proc's type depends on the types of its domain and range; a proc with domain (argument
and range (result type) R has the type D_R. Every expression e has a syntactic type denot
e.g., the result type or range declared for its outermost proc; in general this may depe
arguments. The value of the expression always has this type (satisfies this predicate);
may have other types as well.

Mark: Every value carries a set of marks (e.g., INT or ARRAY; think of them as little flag
top of the value). The predicate HASMARK tests for a mark on a value; it is normally used
type predicates. The set of all possible marks is partially ordered.

The set of marks carried by a value must have a largest member m, and it must include every mark smaller than
Hence all the marks on a value can be represented by the single mark m; we can say that m is the mark on the v

Type-checking: The purpose of type-checking is to ensure that the arguments of a proc sa
predicate of the domain type; this is a special kind of pre-condition for executing the
proc body can then rely on the fact that the formal parameters satisfy their type predic
establish that the results satisfy the predicate of the range type; this is a special ki
condition which holds after executing the proc. Finally, the caller can rely on the fact
results satisfy their type predicate. In summary:

Caller	establish pre-condition: arguments have the domain type;
	rely on post-condition: results have the range type.
Body	rely on pre-condition: formals have the domain type;
	establish post-condition: returns have the range type.

Declaration: A declaration is an ordered set of [name, type] pairs, often denoted like t
BOOL]. A declaration can be instantiated (e.g., on block entry) to produce a binding in w
name is bound to a variable of the proper type; instantiating the previous example yield
[x: VAR INT~(VAR INT).NEW, y: VAR BOOL~(VAR BOOL).NEW].
If d is a declaration, a binding b has type d if it has the same set of names, and for ea
value b.n has the type d.n. A binding b matches d if the values of b can be coerced to yi
b(which has type d.

2.1.3 Programs

Name: A name appearing in the program denotes the value bound to the name in the scope t
name appears in (unless the name is before a colon (declaration) or tilde (binding), or
An atom is a value that can be used to refer to a name; a literal atom is written like t

Expression: In the program a value is denoted by an expression, which is:
a literal value (3 or "Hello"), or
a name (x or salary), or
an application of a proc to other values(Sin[90] or GetProperties[directory, ReadFileName[
a l-expression, which yields a proc value (l [x: INT] IN (IF x<0 THEN x ELSE x)), or
a constructor for a declaration or binding ([x: INT~3, y: REAL~3.14].
If a value is known for each name in the expression, then the expression can be evaluate
produce a value. Thus an expression is a rule for computing a value.

Scope: A scope is a region of the program in which the value bound to a name does not ch
(although the value might be a variable, whose contents can change). For each scope ther
binding which determines these values. A new scope is introduced (in the kernel) by IN or
constructor for a declaration or binding; e.g.,

```
LET x~3 IN x+5;
LET fact~l [n: INT] IN IF n=0 THEN 1 ELSE n*fact[n 1].
```

Constructors;

Incomplete

Recursion:

Incomplete

2.1.4 Conveniences

Coercion: Each type cluster contains To and From procedures for converting between values type and values of other types (e.g., Float: PROC[INT]_[REAL]). One of these procedures is automatically if necessary to convert or coerce an argument value to the domain type of application is a coercion. Each coercion has an associated atom called its tag (e.g., $w INT_REAL or $output for INT_ROPE); several coercions may be composed into a single one if th have the same tag.

Exception: There is a set of exception values. An expression e denotes a value which is type De or is an exception. Whenever an exception value turns up in evaluating an expres immediately becomes the value of the whole expression, unless (in the kernel) the expres the form e BUT {...}. The {...} tests for exception values and can supply an ordinary val another exception, as the value of the BUT expression. An exception value may contain an value, so that arbitrary information can be passed along with an exception.

Finalization: When a variable is no longer accessible, the storage it occupies is freed in the safe language). Before this is done, a finalization proc in the cluster of the va called to do any other appropriate resource deallocation. The local variables of a proc scope may also be finalized (using UNWIND).

Safe: The safety invariant says that all references are legal, i.e., each REF T value is a variable of type T. A proc is safe if it maintains the safety invariant whenever it is arguments of the proper types. If a proc body (l-expression) is

- checked, the compiler guarantees that it is safe, and the proc value is safe;
- trusted, the programmer asserts that it is safe (but the compiler makes no checks) proc value is safe;
- unchecked, the compiler makes no checks and the proc value is unsafe.

It is best to write checked code whenever possible. However, checked code cannot call un (since the compiler then cannot guarantee safety).

Process: Concurrency is obtained by creating a number of processes. Each process execute sequential computation, one step at time. They all share the same address space. Shared (touched by more than one process) can be protected by a monitor; only one process can e within any proc of the monitor at a time. Thus monitor procs can read and write shared d A process can wait on a condition variable within a monitor; other processes can then en monitor. The waiting process runs again when the condition is notified, or after a timeo

2.1.5 Miscellaneous

Allocation

Incomplete

Static

Incomplete

Pragma

Incomplete

The remainder of Chapter 2 is not released in this draft.

Chapter 3. Syntax and semantics

This chapter gives the concrete syntax for the current Cedar language, together with an explanation of the meaning of each construct, and a precise desugaring of each construct kernel language defined in Chapter 2. The desugaring, together with the definitions of t primitives used in it, are the authority for the meaning; the informal explanation is ju reading pleasure. The primitive procs and types of Cedar are specified in Chapter 4.

In addition to the grammar rules and desugaring, there are examples for each construct. intended to illustrate the constructs and do not form a meaningful program. ??? has long which do something interesting and also illustrate the use of the standard Cedar package

The chapter begins with a description of the notation (¶ 3.1) The remaining sections dea systematically with the rules of the grammar, explaining peculiarities of the syntax and semantics:

¶ 3.2, rules 56-61: The lexical structure of programs.

¶ 3.3, rules 1-3: Modules.

¶ 3.4, rules 4-10: Blocks, OPEN, ENABLE, EXITS.

¶ 3.5, rules 11-3: Declarations and bindings.

¶ 3.6, rules 14-18: Statements.

¶ 3.7, rules 19-27: Expressions.

¶ 3.8, rules 28-35: IF and SELECT.

¶ 3.9, rules 26-55: Types.

The order of the grammar rules is:

module,

expression,

type,

and top-down within these.

3.1 Notation

With the exception of the abbreviated non-terminals listed below, every non-terminal app rule is defined in a rule immediately below, or its defining rule is cross-referenced wi superscripted number[12]. If a non-terminal (other than e, t or n) is used in more than one all the rules that use it are listed in a comment after its definition.

The following non-terminals are so basic to the language and so frequently used, that th represented in the grammar by abbreviations:

b=binding[13]
d=declaration[11]
db=declaration or binding[10]
e=expression[19]
n=name[56] (identifier)
s=statement[14]
t=type[36]

3.1.1 Notation for the grammar

The grammar is written in a variant of BNF:
Bold parentheses are for grouping: (interface | implementation).
Item | item means choose one.
?item means zero or one occurrences of item.
item; ... means zero or more occurrences of item separated by ";". The separator may al
ELSE, IN, or OR, or it may be absent. If the separator is ";", a trailing ";" is optiona
item; !.. is just like item; ... but there is at least one occurrence.
A terminal is a punctuation character other than bold ()?| or ;, or a word in underlined
SMALL CAPS. Note that [] and {} are terminals, and do not denote optional occurrence and repetition as the
other variants of BNF.
The rules are numbered sequentially, and each use of a non-terminal not defined just be
referenced with a small superscript number.
, marks an unsafe feature, • an obsolete one; „ a feature needed only for machine-deper
an efficiency hack.

3.1.2 Notation for desugaring

The right-hand column is desugaring into the Cedar kernel language. This is a purely syn
transformation; i.e., it is done on the text of the program, not on the values. The rewr
one rule at a time; a single step of rewriting involves elements from exactly one rule.
desugaring is specified by slightly informal but straightforward rewriting rules, in whi

> An occurrence of a non-terminal (written in bold) denotes the text produced by tha
> terminal in the grammar rule.
>
> Alternation reflects a corresponding alternation in the grammar rule, ? reflects a
> corresponding optional item in the grammar rule, and bold parentheses are for grou
> in a grammar rule. As in grammar rules, literal parentheses are underlined.
>
> Everything else is taken literally.

An underlined non-terminal in the right column means that the desugaring specified for t
terminal must be done in order to obtain a legal program. Otherwise the transformations
done in any order, yielding a legal program at each step.

Every occurrence of expression and type in the desugaring should be enclosed in parenthe
that the desugared program parses as the rewriting rule indicates. These parentheses are
clarity.

For type options like PACKED, the desugaring of the construct in which they appear is a ca
built-in a type constructor which takes a corresponding BOOLEAN argument defaulting to FAL
the attribute is present, the argument is supplied with the value TRUE.

Examples: the following rule for subranges:

subrange ::= (typeName |) (	
([e_1 .. e_2] | [e_1 .. e_2)	(INT | typeName).MKSUBRANGE[e_1, (e_2 | PRED[e_2])]) |
((e_1 .. e_2] | (e_1 .. e_2))	(INT | typeName).MKSUBRANGE[SUCC[e_1], (e_2 | PRED[e_2])]

generates these desugarings

Index [10 .. 20]	Index.MKSUBRANGE[10, 20]
Index [10 .. 20)	Index.MKSUBRANGE[10, PRED[20]]
(1 .. 100)	INT.MKSUBRANGE[SUCC[1], PRED[100]]

Names introduced in the desugaring are written with one or more trailing dash ("(") char
Such names cannot be written in a Cedar program, and hence they are safe from name confl

The desugaring is constructed so that the ordinary scope rules prevent multiple uses of
from being confused.

3.2 The lexical structure of programs

```
56 name ::= letter (letter | digit)... -- But not one of the reserved words in Table ???.
57 literal ::= num | num (B | C | D | H) ?num |    -- WHOLENUMBER, decimal if radix omitted or D, octal i
  digit (digit | A | B | C | D | E | F) ... (B | H) ?num |  -- WHOLENUMBER in hex.. |
  ?num . num ?exponent |                           -- REAL as a scaled decimal fraction; note no traili
  num exponent |                                   -- With an exponent, the decimal point may be omitte
  ' extendedChar | • digit !.. C                   -- CHAR literal; the C form specifies the code in oc
  " extendedChar ... |" ?•L                        [ ('extendedChar), ...] -- Rope.ROPE, TEXT, or STRING
  $ n                                              -- ATOM literal
58 exponent ::= (E | ) ?(+ | - ) num               -- Optionally signed decimal exponent
59 num ::= digit !..
60 extendedChar ::= space | \ extension |
  printingCharExceptQuoteOrBackslash
61 extension ::= digit1 digit2 digit3 |-- The character with code digit1 digit2 digit3 B |
  (n | N | r | R | t | T | b | B |                 -- CR, '\015 | TAB, '\011 | BACKSPACE, '\010 |
  (f | F | l | L | \ | ' | "                        -- FORMFEED, '\014 | LINEFEED, '\012 | \ | ' | "
```

```
m, x1, x59y, longNameWithSeveralWords: INT;
n: INT~1+12D+2B9+2000000000B
          +1H+0FFH;
r1: REAL~0.1+.1+1.0E 1
           +1E 1;
a1: ARRAY [0..3] OF CHAR~['x, '\n, '\', '\141];
r2: ROPE~"Hello.\n...\nGoodbye\F";
a2: ATOM~$NameInAnAtomLiteral;
```

The main body of the grammar (rules 1-55) treats a program as a sequence of tokens. Rule
give the syntax of most tokens. A token is:

A literal[57]. More information about literals of type T can be found in Chapter 4, a
the treatment of type T.

A name[56].

A reserved word, which is a string of uppercase letters that appears in the list o
words in Table ???. A reserved word may not be used as a name, except in an ATOM li

One of the following two-character symbols (used in the grammar rules indicated):

~=	not equal[19 30]
<=	less than or equal[22]
~<	not less thanequal[19 30]
>=	greater than or equal[22]
~>	not greater than[19 30]
=>	chooses[8 17 30 31 33 35]
>	RETURNS[43]
..	subrange constructor[25 48]
~~	bind by name[6 34]

A punctuation symbol: any printing character not a letter or digit, and not part o
the two-character sequences above. The punctuation symbols are: !@#$~*-
+=|(){}[]_^;:'",.<>/. The following ASCII characters are not punctuation symbols (a
illegal in a program except in an extendedChar[60]): %&\?. Note that Cedar uses a var
ASCII which includes the characters _ (instead of the underline) and ^ (instead of the
circumflex).

A comment is not a token and may appear between any pair of tokens; it is a token delimi
hence cannot appear in the middle of a token.

The program is parsed into tokens by starting at the beginning and successively taking f
front the longest sequence of characters which forms a token according to the rules abov
discarding any amount of initial whitespace or comment.

> Whitespace is a space, tab, and carriage return.
>
> A comment is a sequence of characters beginning with --, not containing -- or a ca return, and ending either with -- or with a carriage return.

Whitespace and comments thus do not affect the meaning of the program except:

> When they delimit a token.
>
> Within a CHAR literal or a ROPE literal, where they are taken literally. Thus ' is '\040, and "I
> am --not--" is equal to "I\Nam --not--" and different from "I\Nam ".

Both reserved words (Table ???) and names with predefined meanings (Table 4 5) are made entirely of upper case letters. These names may not be rebound by the program.

Note: Semi-colons are used to separate declarations, bindings and statements in a body, separate choices in a statement. Commas are used to separate declarations in fields (i.e domain or range, a recordTC or a unionTC), bindings in an application, choices in an exp in a unionTC. In general these sequences may be empty, and an extra separator at the end harmless except when the sequence is bracketed with [].

The braces which delimit a block[4], interface body[2], choices in an enable[7], or MACHINE CODE may be replaced by BEGIN and END brackets. BEGIN replaced "{" and END replaces "}". If one brace is replaced, its matching partner must also be replaced. The braces delimiting an may not be replaced by BEGIN ... END.

3.3 Modules

```
1 module ::= DIRECTORY (n_d (: TYPE  (n_t |l)[|(n)_d( : ( (TYPE n_t | TYPE n_d) | TYPE n_d), ... ] IN
       ?(USING [ n_u, ... ] ) ),  ... ;     LET (n_d(~RESTRICT[n_d(, [$n_u, ... ] ] ), ...
 ( interface | implementation )     IN ( interface | implementation )
2 interface ::= n_m :  ?CEDAR DEFINITIONS LET (n_it((~n_it), ... IN l [((n_iv | n_it):n_it((), ...]=>
 ?(IMPORTS ( (n_iv : | ) n_it ), ...)                [n_m: TYPE n_m] IN
 ?(SHARES n_s !..)                  -- access to PRIVATE names from n_s allowed in the mod
 ~ ?access12 { ?open6 db10; ... } .   open [ db, ... ]
3 implementation ::= n_m, !.. : ?CEDAR LET (n_it((~n_it), ... IN l [( ( n_iv | n_it ):n_it((), ...]
 ?„RESIDENT ( PROGRAM drType43  |           [(n_e: n_e , ... , n_m: TYPE n_m , CONTROL: PROGRAM]
                   MONITOR drType43 LET LOCK~MONITORLOCK.NEW ,
                      ( | LOCKS e ( | USING n_u : t)) (l) IN LOCK | (l IN e | l [n_u : t] IN
                                          lock(~l aul IN)
 ?(IMPORTS ( (n_iv: | ) n_it), ...) ?(EXPORTS n_e, ...)  LET b(~NEWPROGINSTANCE[block].UNCONS IN
 ?•(SHARES n_s, !..)                    [ (n_e~BINDDFROM[n_e, b(] ), ... , n_m~b( ]
 ~ ?•access12 block .               where the body of the block is desugared to a decl 
                                        [ db, ... , n_m: PROGRAM drType={s; ...} ]
```

```
DIRECTORY
  Rope: TYPE USING [ROPE, Compare],     -- There should always be a USING clause
  CIFS: TYPE USING [OpenFile,Error,Open,read],  -- unless most of the interface is used
  IO: TYPE IOStream,
```

```
  Buffer: TYPE;                          -- or it is exported.

BufferImpl: MONITOR [f: CIFS.OpenFile]-- Implementations can have arguments.
  LOCKS Buffer.GetLock[h]^             -- LOCKS only in MONITOR, to specify
    USING h: Buffer.Handle             -- a non-standard lock.
  IMPORTS Files: Files: CIFS, IO, Rope -- Note the absence of semicolons.
  EXPORTS Buffer                       -- EXPORTS in PROGRAM or MONITOR.
~ { -- module body -- } .              -- Note the final dot.
```

Modules serve a number of functions (which might perhaps better be disentangled, but are

A file of text (BufferImpl.mesa, or its translation into object code, BufferImpl.bcd)

The unit handled by the editor, named in DF files and models, and accepted by the compiler, the binder, and the loader.

A set of related structures (types, procedures, variables) which are freely access other, hiding irrelevant information from other modules.

A procedure which can accept interface types and bindings as arguments, and return interface values as results.

The first two uses are not relevant to the language definition, and are not discussed fu see ???. The others are the subject of this section.

There are two kinds of modules: interface modules (written with DEFINITIONS) and implementa (written with PROGRAM or MONITOR). They have the same header (except that interfaces have n RESIDENT option or EXPORTS list); it defines the parameters and results of the module viewe proc (¶ 3.3.1) and specifies the name n_m of the module. The bodies (following the ~) are

The remainder of this section deals in turn with:

Modules as procedures, and the interface or instance values obtained by applying t 3.3.1).

How modules are applied (¶ 3.3.2).

Module parameters: the DIRECTORY and IMPORTS lists; USING clauses (¶ 3.3.3).

Interface module bodies and interfaces (¶ 3.3.4).

Implementation module bodies; the EXPORTS list (¶ 3.3.5).

SHARES and access[12] (¶ 3.3.6).

The meanings of the other parts of a module header are discussed elsewhere:

CEDAR in ¶ 3.4.4.

MONITOR is ¶ CONC.???.

RESIDENT in ¶ ???.

3.3.1 Modules and instances

A module is a proc which takes two kinds of arguments:

Interfaces, declared in the DIRECTORY list. These arguments are supplied by the mode on the command line for the compiler),

Interface instances, declared in the IMPORTS list. These arguments are also supplied model (or in a config file passed to the binder, or implicitly by the loader).

¶ 3.3.3 discusses the types of these arguments and how they are declared. In addition, a

implementation may take PROGRAM arguments declared in the drType following PROGRAM or MONITOR. These are ordinary values; they are discussed in ¶ 3.3.2.1.

When a module is applied to its arguments, the resulting value is

For an interface module, an interface, also called an interface type.

For an implementation module, a binding whose values are instances: one interface for each interface it exports, plus one for the program instance, also called a gl

This application cannot be written in the program, only in the model; it is described in

An interface or interface type is a type, as the latter name suggests. This type is a de (obtained from the declarations which constitute the module body), with an extended clus includes all the bindings in the module body that don't use declared names (¶ 3.3.4). A type is an interface is an interface instance; such values are the results of instantiat implementation modules.

A program instance or a global frame is a frame, as the latter name suggests, i.e., a bi from the bindings and declarations of the module body, just like any proc frame (¶ 3.3. Normally the part of the program outside the module does not deal with the instance dire only with the exported interface values.

In most cases, there is:

Exactly one application of each module, and hence exactly one interface or one ins

Only one module which exports an interface.

Only one interface exported by a module.

Only one argument of the proper type for each module parameter (¶ 3.3.3), so that redundant to write the arguments explicitly.

When these conditions hold, there is a close correspondence among the following four obj
an interface module;
the interface which it returns (since the arguments need not be written explicitly
the implementation module which exports the interface;
its instance (again, since the arguments need not be written explicitly).
The distinctions made earlier in this section then seem needless; it is sufficient to si the interface and implementation modules, and identify them with the files which hold th more complicated situations, however, it is necessary to know what is really going on.

Need an example

3.3.2 Applying modules

A module is not applied to all its arguments at once. Instead, the arguments are supplie stages:

A module is applied to its interface (DIRECTORY) arguments by compiling it; the resu BCD (represented by a .bcd file). The bcd is still a proc, with instance parameters proc, a module can be applied to different arguments (i.e., different versions of interface arguments) to yield different BCDS.

A BCD is applied to its instance (IMPORT) arguments by loading (or binding) it; the a program instance, together with any interface instances exported by the module. the BCD can be applied to different arguments (i.e., different interface instances) different instances. Indeed, because an instance may include variables, even two a to the same arguments yield different instances.

These two stages are separated for several reasons:

All the type-checking of a module can be (and is) done in the first stage, by the
The only type errors possible in the second stage are supplying an unsuitable argu

Compiling is much slower than loading, and a module needs to be recompiled only wh
its interface arguments change, not when the interface values change. The latter a
in the implementations of the interfaces, and are much more common.

Other reasons. History

3.3.2.1 Initializing a program instance

A program instance PI may be uninitialized, because no code in the module is executed wh
instance is made. It is the job of the PROGRAM proc PP to initialize PI, perhaps using the
arguments if there are any. Until PP has been called, PI is not in a good state. It would
supply the PROGRAM arguments along with the imported instances, and call PP as part of mak
PI, so that PI is never accessible in its uninitialized state. But it isn't done that way
programmer must ensure that PP is called (using the START construct, ¶ 4.4.1) before any
made of PI. Note that PP also contains the initialization code for any variables or non-s
in the instance; e.g., if x: INT_3, the value of x will not be 3 until after PP has been

There is some error detection associated with this kludge. If a proc in the instance is
the instance has been initialized by START, a start trap occurs. At this point, if PP take
arguments it is called automatically, and the original call then proceeds normally; if P
arguments, there is a Runtime.StartFault ERROR.

Caution: If the module is a monitor, PP runs without the monitor lock; if another process
the module while PP is running, it will not wait, but will run concurrently with PP. This
to be right. It is unwise to rely on a start trap to initialize a monitor module; call P

Caution: If a variable in the instance is referenced before the instance has been initia
is detected, and the uninitialized value will be obtained. PP can still be called to init
instance, and may still be called automatically by a start trap.

3.3.3 Parameters to modules: DIRECTORY and IMPORTS

The interface parameters of a module are declared in the DIRECTORY list. An interface I ha
TYPE n, where n is any one of the names given before DEFINITIONS in the header of the inter
module that produced I it. The use of these names provides a clumsy check that the prope
interface is supplied as an argument.

An interface is a type which can only be used:

Before a dot (¶ 4.14), to obtain a value from the type's cluster, which simply con
bindings in the interface module body (¶ 3.3.4).

In an IMPORTS list as the type of an instance parameter to a module.

The USING clause in the DIRECTORY, if present, restricts the cluster of the interface to co
items with the names n_u, ... Thus in the example, only ROPE and Compare are in the cluster
in the BufferImpl module. This means that Rope.ROPE and Rope.Compare are legal, but Rope.n f
other n will be an error. Note that USING affects only the cluster of the parameter; it d
the clusters of any types or the bodies of any INLINE procs obtained from the interface.
Rope, Compare might be bound by

```
Compare: PROC[r1, r2: ROPE]_[BOOL]~INLINE {
  IF Length[r1~=Length[r2] THEN ... }
```

A call of Rope.Compare in BufferImpl is perfectly all right, even though Rope.Length would
error.

In the example, CIFS, IO, and Rope are interfaces. They are the types of three IMPORTS para named Files, IO, and Rope (if the IMPORTS clause gives no name for the parameter, the name interface is recycled). An actual argument for an IMPORT parameter must be an interface i i.e., a value whose type is an interface type. Such a value is obtained from one or more which export the interface (¶ 3.3.5). An instance is a binding; in this binding the valu declared in the interface is provided by the exporter; the value of a name bound in the (such as Compare) is just the value that the interface binds to the name (in this case, t proc). This rule has two effects:

The client can ignore the distinction between names bound and declared in the inte since both appear in the instance binding and are referenced uniformly with dot no This means that the client is not affected, for example, when a proc is moved from INLINE in the interface to an ordinary definition in an implementation.

The client can often ignore the distinction between the interface and the instance the values in the interface are also in the instance, with the same names. This is motivation for the shorthand which allows the name of an IMPORT parameter to defaul the name of the interface; the interface is no longer accessible, but Rope.Compare same meaning whether Rope is the interface or the instance.

Restriction: An interface module may not import more than one instance of a given interface I. If an implement module P imports more than one instance of I, the principal instance of I is the one with no name in the IMPORTS (which is therefore named I by default). If P imports only one instance of type I, then that instance is the p instance.

Restriction: Often an interface module has no IMPORTS, because it only needs access to the static values (type, constants) bound in its interface parameters, and does not need values for any names declared there (ordinary interface variables). If an interface module does have IMPORTS, however, and there is more than one instance of imported interface around, then there is a restriction on the argument values. Suppose that Int1 imports Int2, program module P imports Int1. Then Int1 may only import one instance of Int2, and if P also imports Int2, the p instance of Int2 in P must be the same as the value of Int2 imported by the Int1 imported by P. For example, wi

```
DIRECTORY Int2; Int1: DEFINITIONS IMPORTS Int2V: Int2 ...
DIRECTORY Int1, Int2; P: PROGRAM IMPORTS Int1V: Int1, Int2V: Int2 ...
```

we must have in P that Int1V.Int2V=Int2.

3.3.4 Interface module bodies

The body of an interface module I is a collection of bindings (e.g., y: INT~7) and declar x: INT). There are restrictions on what may follow the ~ in one of the bindings[11]:

If it is an expression, it must be static (¶ ???).

If it is a block (providing the body of a proc), it must be INLINE.

It may not be CODE.

The values of the bindings can be accessed directly by dot notation (e.g., I.y, which is here). The declarations cannot be accessed directly (I.x is an error).

The result of applying an interface module is an interface (¶ 3.3.2), which is a type. T simply the declaration obtained by collecting the declarations in the body, with a clust extended to include all the bindings of the body. The bindings may not refer to names in in the declaration, except that:

Any declared name may be used

in the body of an INLINE, or

after a "_" in a defaultTC[40] in the fields[44] of a transferTC[41] which is the ty decl in the interface's db.

A declared type may be used anywhere.

The declarations in an interface module are not quite like ordinary declarations. They a kinds, depending on whether the type of a declaration is:

A transfer type; this is just like a declaration of a transfer parameter to an ord
except that it is readonly.

TYPE or TYPE[e]; this is an opaque type or exported type, discussed in ¶ 3.3.4.1 bel
expression e must be static. These types are not allowed in an ordinary declaratio

VAR T, or READONLY T for any other type T; this is an interface variable; discussed i
3.3.4.2 below. •T can be written for VAR T, which is not allowed in an ordinary dec

An interface instance II has the interface type I if for each item n: T in the interface,
item n~v in the instance, and v has type T. This is the same rule which determines that a
has the type of a declaration; e.g., that a proc argument has the domain type. In this r
nothing special about an interface.

Note that a name can be declared PRIVATE in an interface, even though it must be declared
in the exporter. This can be useful if the name is used in a type constructor or inline
interface, but its value should not be accessible to the client.

3.3.4.1 Opaque types

An opaque type declaration in an interface is the only way to declare a type parameter (
the interface parameters declared in the DIRECTORY). Not surprisingly, any type has type T
any type can be supplied as the argument for an opaque type declared T: TYPE. T is called
opaque. A type V has type TYPE[n] if:

SIZE[T]=n.

V has standard NEW, INIT, ASSIGN, EQUAL and ISTYPE procs. All the assignable primitive
types do except the RC types (¶ 4.5.1), bound variant types (¶ 4.6.2), and types p
a defaultTC[40].

Representation: The standard NEW proc allocates n words. INIT does nothing. The standard ASSIGN
copies n words. The standard EQUAL compares n words bitwise. The standard ISTYPE compares the mark of the
value with a single mark associated with the type.

Only such a type V can be supplied as the argument for an opaque type declared U: TYPE[n]
called n-opaque.

The cluster of a fully opaque type T is empty: it provides no operations. A T value canno
passed as a parameter, and there are no VAR T variables. Thus you cannot use T as the typ
declaration. The only thing to do with T is use it as the range of a reference type such

The cluster of an n-opaque type U has VAR, NEW, INIT, ASSIGN, EQUAL and ISTYPE procs (the last
yet implemented). Thus these operations can be done on a U value. As a consequence, a U v
can be passed as a parameter and declared.

Restriction: All instances of any interface produced by applying an interface module which declares an opaque
must supply the same type value for T if they supply any value at all; this value is called the standard imple
T. Because of this restriction, clients can safely interassign values of type T, no matter how obtained. In ad
safe for any exporter of T to convert a value of type T to a value of the argument type, and conversely. The r
arises from the fact that the current implementation cannot properly distinguish among the different instances
different values for T can get mixed up. If there is only one value, a mixup cannot compromise type safety.

Two type values are the same in this sense only if they come from the same type constructor (presumably in som
shared interface module, usually one which is private to the two implementors of T).

It is not necessary to import an interface to refer to an opaque type declared in that i
(because of the above restriction).

Within an implementation P which exports an opaque type T declared in interface I, D.T ar
(simply T within P) imply each other. However, they have different clusters, and are not
You can convert from one to the other using NARROW (¶ 4.3.1).

Performance: This conversion costs nothing at runtime.

3.3.4.2 Interface variables

An interface variable gives clients of an interface direct access to the variable in a p
which is exported to provide its value This is the only kind of variable parameter in cu

•If you use the obsolete shorthand of T for VAR T in an interface variable declaration, y
declare a transfer type variable as an interface variable, since that already means pass
value.

Caution: the variable which is exported to provide the value for an interface variable i
initialized until its module is initialized (¶ 3.3.2.1). However, there is nothing to st
accessed.

Performance: An interface variable can be read and (if not READONLY) set directly, which i
significantly faster that Get and Set procs. Of course, the implementor gives up some cor
not quite as fast as access to an ordinary variable, since there is an extra level of in
costs one or two extra instructions each time. There is also one pointer per interface v
module which refers to it.

•You can get direct access to all the variables of a module by using a POINTER TO FRAME typ
4.5.3), but this is not recommended.

3.3.5 Implementation module bodies

The body of an implementation module Imp is simply a block. This block plays two roles. O
one hand, it is an ordinary block, the body of an almost ordinary proc PP called the PROG
proc, which has parameters and results like any other. PP is special in one way: it has a
type rather than a PROC type. When PP is applied (using the special construct START; see ¶
its declarations and bindings are evaluated, its statements are executed, and its result
as with any proc. The only difference is that the values bound to the names introduced i
(i.e., PP's frame) are retained after the proc returns; in fact, forever (unless Runtime.U
free the frame). Procs local to the block can access these values in the usual way, and
exported names can also be accessed through interfaces, as explained below.

As with any proc (¶ 3.5.1), PP's frame includes the parameters and results from Imp's drT
as the names introduced in the block's db. It also includes an additional name

    Imp: PROGRAM T~PP

where Imp is the name of the module, T is its drType, and PP is the proc described above.

The body of Imp has a second role: to supply values for the names declared in the interfa
exported by Imp. For each interface Ex which Imp exports, an interface value ExI of type
constructed. Each name n in ExI acquires a value as follows:

> If n: T is in Ex and n~x in the body of Imp, then n~x in ExI. This is a slightly pec
> kind of binding, and like ordinary binding, x must be coerceable to T (¶ 4.13). A
> have PUBLIC access (¶ 3.3.6) in the body.
>
> If n is declared in Ex and not bound in the body of Imp, then n~UNBOUND in ExI.
> UNBOUND is a special value with the following properties:
>
> > For a proc P, it causes a Runtime.UnboundProcedure signal on any aplication of
> >
> > For a variable v, it causes a Runtime.PointerFault error on any reference to v
> >
> > For a type T, it causes an error on any application of T.ISTYPE (including NAR
> > and WITH ... SELECT). Other uses of T are perfectly all right.

If n~x in Ex, then n~x in ExI. Thus any names bound in the interface are bound the way in any interface value.

Caution: A name can be exported to several interfaces without any warning, if it has a s This is unlikely to be what is wanted.

The result of instantiating Imp is a binding B with:

One item for each exported interface Ex, namely Ex: Ex~ExI, where ExI is the interfa value constructed above. Here Ex is the name n_d given to the interface in the DIRECT

One item for Imp itself, namely Imp: POINTER TO Imp~programInstance, where programInstance is the program instance, i.e., the frame of the module's body.

This binding is accessible in a model, where it can be used to get access to the interfa instances.

What is the current story on executable NEW? prog from DIR: gets file name Or, copy from imported prog or PTF.

Where do we put impl in DIR?

3.3.6 PUBLIC, PRIVATE and SHARES

Cedar has a rather complicated mechanism for controlling access to names. Most uses of i considered to be obsolete, with the following exceptions:

Names to be exported must be declared PUBLIC.

Names included in an interface for use in inline procs etc., but not intended for clients, should be declared PRIVATE.

Acess to a name is declared by writing PUBLIC or PRIVATE right after the colon in a declara name:

```
x: PUBLIC T
```

In the Cedar syntax these colons occur in the declarations[11] and bindings[13] in bodies[9], fi and interface modules[2], and in the tag[50] of a unionTC. You can set a default access for a names in a module[2, 3] or record[46] by writing PUBLIC or PRIVATE just before the { or RECORD; th overridden by accesses inside. By default, an interface is PUBLIC and an implementation is

A PRIVATE name defined in module M can only be referenced:

from within M;

from a module which SHARES M; avoid this feature unless you export M.

This does not mean that the name is invisible if, e.g., M is OPENed, but that it is an er Thus in

```
x: INT; {OPEN M; f [x]}
```

if x is bound in M (and not suppressed by a USING clause), the call of f is equivalent to regardless of whether x is PUBLIC or PRIVATE. It is illegal if x is PRIVATE, but it never re x declared by the x: INT.

Furthermore, if a record has any PRIVATE components, a constructor or extractor for the re legal only in a module where use of the PRIVATE names is legal.

3.4 Blocks, OPEN and ENABLE

```
4 block ::= attributes { ?open ?enable body
  ?(EXITS (n, !..=>s); ...) }
  In 3, 13, 15.
5 attributes ::= ( CHECKED | UNCHECKED |
  TRUSTED ) ...
6 open ::= OPEN ( n ~~ e | • e ), ...;
  In 2, 4, 17. •The ~~ may be written as :.
7 enable ::= ENABLE (eChoice | {eChoice, ...})
  In 4, 17.
8 eChoice ::=(e | ANY), !.. => s
  In 7, 26.
9 body  ::= ?( db; ... ;) s; ...
  In 4, 17.
10 db ::= d | b
  In 2, 9.
```

```
OPEN n((, ... : EXCEPTION~NEWLABEL[] , ...
  IN body ) BUT { (n((, ... => s ); ... }
-- n(( is not visible in s.

LET n~l_open IN e.DEREF |    --The final IN is a separa
  LET BINDN[D(e.DEREF).NAMES,
    OPENPROCS[D(e.DEREF).NAMES, l IN e.DEREF] ] ) IN ...

BUT (.. {eChoice| { eChoice, ... } )

(e | ANY), ... => { s; REJECT }

LET NEWFRAME[ [db, ...] ].UNCONS IN { s; ...}
```

```
CHECKED {                              -- Unnamed OPEN OK for exported
  OPEN Buffer, Rope;                   -- interface or one with a USING clause.
  ENABLE  Buffer.Overflow=>GOTO HandleOvfl; -- A single choice needn't be in {}.
  stream: IO.Stream~IO.CreateFileStream[UBE]; -- Use a binding if a name's value is fixed.
  x: INT_7;                            -- Better to initialize declared names.
    {OPEN b~~GetBuffer[stream];        -- A statement may be a nested block.
      ENABLE  {                        -- Multiple enable choices must be in {}.
      CIFS.Error[--error, file--]=>{   -- ERRORs can have parameters.
        stream.Put[IO.rope[error]];
        ERROR Buffer.Error["Help"] };  -- Choices are separated by semicolons.
      ANY=>{ x_12; GOTO AfterQuit } }; -- ANY must be last. ENABLE ends with ;.
    y: INT_9; ... };                   -- Other bindings, decls and statements.
  x_stream.GetInt; ...                 -- Other statements in the outer block.
  EXITS                                -- Multiple EXIT choices are not in {}.
    AfterQuit=>{...};                  -- AfterQuit, HandleOvfl declared here,
    HandleOvfl=>{...} };               -- legal only in a GOTO in the block.
```

The main function of a block is to establish a new scope (¶ 2.3.4) and to allow for the variables declared in the block, as in Algol or Pascal. A Cedar block has four other fea

attributes[5]: CHECKED, UNCHECKED and TRUSTED are treated in ¶ 3.4.4 on safety.

open[6]: a combination of sugar for LET and call by name; see ¶ 3.4.2.

enable[7]: catches signal and error exceptions in the body; see ¶ 3.4.3.1.

EXITS: catches GOTO exceptions in the body or enable; see ¶ 3.4.3.2.

Note that the braces around a block may be replaced by BEGIN and END (¶ 3.2).

3.4.1 Scope of names and initialization

The names introduced in the block body's db (i.e., appearing before a : or ~) are known body with the values supplied by the db, except in inner scopes where they are reintrodu are not known elsewhere in the block. The frame of the block is a binding with a value f such name.

Actually, the frame is a value of an opaque type which has a coercion (called UNCONS) to this binding. As the de for body indicates, the frame is constructed (by NEWFRAME), and then a LET makes the names in the binding known i the statements of the body.

Anomaly: A name introduced by a binding, n: T~e, has the value of e throughout the body i
static. If e is not static, it is evaluated after all preceding db's, but before any fol
means that n is trash in all the db's before its binding. Symmetrically, if e refers to a
introduced in a following decl or non-static binding, it will get a trash value. Compili
switch will cause a warning in this case. Note that only attempts to obtain the value of
n may appear anywhere in a l-expression, and all will be well as long as the l-expressio
applied before n's binding is evaluated.

A name introduced by a declaration, n: T, is bound to a new VAR T. The variable bound to
allocated, and its INIT proc executed (to set a REF or transfer value to NIL) before anyth
block is executed (this is done by the NEWFRAME proc in the desugaring).

Anomaly: However, any initialization specified by a defaultTC[40] in T is done at the same
a non-static binding would be evaluated. As with a binding, n is trash before this time.
Furthermore, any (unwise) assignment to n before this time will be overriden by the defa

The expression in a binding or defaultTC should be functional, or at least it should hav
benign side-effects. There is no enforcement of this recommendation, unfortunately. In c
Cedar such an expression is evaluated exactly once, at the time described above. This ma
in the future, however.

The variables created by a declaration are deallocated when execution of the block is co
unless the block's frame is retained. Currently only an implementation's block[3] has its
retained. There are two ways to hang on to a variable v after execution of the block is

> Obtain a pointer to v with @; this pointer value can survive the block.
>
> Obtain a proc value for a local procedure which refers to v; this proc value can s
> block.

In the checked language both these dangling references are impossible: the @ operator, b
unsafe, is forbidding, and ASSIGN for proc values gives an error unless the proc is local
program instance (which has a retained frame). An unchecked program can get into trouble
however.

Performance: There is no overhead associated with block entry or exit, even if the block
open, enable or EXITS. The only cost is for initializing its names. It is good style to u
to limit the scope of names.

3.4.2 OPEN

There are two forms of open. The first, n~~e, binds the name n to l_{open} IN e.DEREF. This is
l IN e.DEREF, except that there is a coercion from n to n[]. In other words, every time n
value is obtained by evaluating e.DEREF. The effect is exactly like call by name in Algol
to remind you that this is not ordinary value binding. The value of e.DEREF is e if the c
does not have a DEREFERENCE proc, or e^.DEREF if it does. In other words, a reference valu
dereferenced, repeatedly if necessary, to obtain a non-reference value. In an open, e.DER
a record, interface or instance.

The scope of an open is all the rest of the block, including any enable and any EXITS. A
open may have several bindings. These are applied sequentially, so that the names bound
ones are known to the later ones as well as to the rest of the block.

The second, nameless, form of open gives an expression without binding it to a name: { OP
...}; e must evaluate to a binding b:

> A record value has a corresponding binding (returned by UNCONS in the desugaring) wh
> has the names of the record fields are bound to the field values (or variables, fo
> record).

An interface or instance value is a binding (¶ 3.4.2).

The nameless open converts b into another binding bp in which each value is a l_{open} proc, introduces bp's names in the block with a LET. Thus in the program

```
R: RECORD [a: INT_3, b: REAL_3.4]; r: R;
{ OPEN r; ...}
```

the names a and b are known in the body of the block, and have exactly the same meaning a and r.b.

Style: Good style demands that a nameless open be used with discretion, with the smalles practicable scope, and only if the value being opened is very familiar, or heavily used, Nameless open can cause great confusion, since it is not obvious from the text of the pr where to find the bindings for the names it makes known.

3.4.3 ENABLE and EXITS

The ENABLE and EXITS constructs are two forms of sugar for exception handling (¶ 2.2.4). EN catches signals and errors raised in the body (but not the open, enable, or exits; EXITS GOTOS in the body or enable (but not the open or exits). Both are in the scope of the ope Neither is in the scope of any names introduced in the body.

3.4.3.1 ENABLE

An enable has a chance to catch any signal or error raised in the block (and not caught level). A nearly identical construct can appear in an application[26]; the following explan both cases.

Each enable choice (eChoice[8]) has a list of expressions with exception values, •or ANY, b => . If ANY appears, it must be in the last eChoice. If the exception is equal to one of t or if ANY appears, the statement after the => is executed. Control leaves this statement the following ways:

A REJECT statement causes the exception to be the value of the block; it will then propagated within the enclosing block, or if the block is a proc body it will be p to the application.

A GOTO statement sends control to the matching choice in the EXITS. There are three special cases:

A RETURN is not allowed in an eChoice.

A CONTINUE statement ends execution of the current statement (in this case the block); execution continues with the next statement following. If the block body, the effect is the same as RETURN. You cannot write CONTINUE in a body's db.

•A RETRY statement begins execution of the current statement (in this case th block) over again at the beginning. You cannot write RETRY in a body's db.

The semantics of CONTINUE and RETRY follow from the desugaring of statement[14].

A RESUME statement (signals only) is discussed below.

•If the statement finishes normally, a REJECT statement is then executed.

If a single expression with value v appears before the =>, then within the eChoice state names in v.DOMAIN are declared and initialized to the arguments of the exception. With mu expressions, or ANY, the arguments are inaccessible. •The use of ANY is not recommended.

Finalization

You are supposed to think of an ERROR as an unusual value ev which can be returned from a
application; this value immediately stops the evaluation of the containing application,
likewise returns ev as its value. This propagation is stopped only by an enable choice wh
the ERROR. As each application is stopped, it is finalized. Aside from invisible housekee
finalization confusingly consists of executing the statement in an eChoice which catches
UNWIND. The programmer can write any cleanup actions he likes in this statement. If the
finalization raises another ERROR which it does not catch, it will itself be stopped, wit
confusing consequences. It isn't very useful to know exactly what happens then: avoid th

Caution: In fact, things are a bit more complicated. When a signal or error is propagate
eChoice statement is called as a proc from the SIGNAL or ERROR which raises the exception.
control leaves the statement by a GOTO (or CONTINUE, RETRY or LOOP), the finalization is do
This means that the eChoice statement is executed before any finalization. This is usefu
which often resume. In some cases, however, notably if finalization would release monito
can cause trouble. Avoid the problem by exiting from the enable immediately with a GOTO.

Caution: An eChoice can raise a second exception ex2 and fail to catch it. This will pro
in confusion, and should be avoided. If it happens, ex2 is propagated just like the firs
ex1; all the eChoices which saw ex1 will see ex2. This is because the eChoice statement
called as a proc. Unless ex2 is a signal which is resumed, the eChoice which caught ex1 w
finalized and abandoned.

Caution: ANY unfortunately catches UNWIND, and hence its statement will be taken as the
finalization. It is better not to use ANY. Also, it is possible to raise UNWIND explicitly

Signals

Incomplete

3.4.3.2 EXITS

An EXITS construct (confusingly called REPEAT in a loop) declares one or more exceptions wh
local to its block, and also catches them. The syntax is just like an enable. However, n
labels appear before the => rather than expressions, and the EXITS introduces these names
scope which includes the block body and any enable, but not an open and not the statemen
EXITS itself. A label may only be used in a GOTO statement.

Anomaly: Actually labels have their own name space, disjoint from the other names known
block. Hence it is possible to declare a label n and still to refer to another n in the
feature.

Like the raising of any exception, a GOTO n stops execution of the current statement. The
associated with n is executed. If it finishes normally, execution continues after the bl
was declared. If it raises an exception, that exception becomes the value of the block.

3.4.4 Safety

A SAFE proc has the property that if the safety invariants hold before it is called, they
afterwards. Roughly, these invariants ensure that the value of every expression has the
of the expression, and that addresses refer only to storage of the proper type (¶ 4.5.1)
proc may lack this property. Hence a safe proc type implies the corresponding unsafe one

We want to have confidence that the safety invariants hold. To this end, we want to have

as few unsafe procs as possible;

a mechanical guarantee that a proc is safe, if possible.

Clearly, a proc whose body calls only safe procs will be safe.

Applying this observation, Cedar provides three attributes which can be applied to a bl

CHECKED: the compiler allows only safe procs to be applied; hence the block is automatically safe, and any proc with the block as its body is safe.

UNCHECKED: there are no restrictions on the block, and it is unsafe.

TRUSTED: there are no restrictions on the block, but the programmer guarantees that preserves the safety invariants; the compiler assumes that the block is safe. This restricted form of LOOPHOLE.

These attributes are defaulted as follows.

A block is checked if its enclosing block is checked; otherwise it is unchecked.

If CEDAR appears in the module header, the outermost block is checked, and a transf constructor anywhere in the module defaults the SAFE option to TRUE. Hence the resul type will be safe, and its initialization must be safe or there is a type error.

Otherwise, the outermost block is unchecked, and a transfer type constructor anywh the module defaults the SAFE option to FALSE. Hence the resulting type will be unsaf there is no safety restriction on its initialization.

Of course you can override these defaults by writing CHECKED, UNCHECKED or TRUSTED on any block, and SAFE or UNSAFE on any transferTC. The defaults are provided to make it convenie

write new programs in the safe language;

continue to use old, unsafe programs without massive editing.

An unsafe proc value cannot be bound to a name declared with a safe type. This applies t choices and signals as well as to procs. In both cases, the body must be checked or trus type is safe. ERRORs (including UNWIND) are treated differently, however, because of the v an ERROR is a value returned from an application, unlike a signal which calls the eChoice expression. Hence the eChoice for an ERROR is treated just like any statement in its encl and is not considered to be bound to a proc when the ERROR is raised.

The following primitive procs are unsafe:

@, DESCRIPTOR and BASE.

^ or FREE applied to a pointer, and all pointer arithmetic.

withSelect[34].

APPLY for process and port types (JOIN and port calls).

LOOPHOLE which produces a RC value (¶ 4.5.1).

APPLY of a sequence or sequence-containing record.

The fields of an OVERLAID union.

ASSIGN of:

An unspecified type to anything other than the same unspecified type (¶ 4.9)

A union or variant record.

A proc, if the value being assigned is local to another proc, rather than to implementation. Such a proc value also cannot be passed as an argument to FOR

3.5 Declaration and binding

11 declaration::= n, ... : ?access varTC[89] | n: varTC), ...
In 10, 44. READONLY only for interface variable.

12 access ::= PUBLIC | PRIVATE
In 2, 3, 11, 13, 46, 47, 50.

13 binding ::= n_1 ?(, n_2, ...) : ?access -- The desugaring for n_2 is at the end.

t ~ e	n_1: t~e
TYPE ~ t_2	n_1 : TYPE ~ t_2 -- Same as e except for conflicting synt
t ~ CODE	n_1 : t ~ NEWEXCEPTIONCODE[] --tgSIGNAL or ERROR
t ~ (ENTRY | INTERNAL | INLINE)... block[4]	n_1 :|t~l [d(: t.DOMAIN] IN LET rb(~NEWFRAME[t.RANGE] IN LET rb(IN {t.DOMAIN~d(; block; RETURN} BUT {RETURN(=>rb
„,t ~ MACHINE CODE { (e, ...); ... })	n_1 : t~MACHINECODE[(BYTESTOINSTRUCTION[e, ...]), ...]) ?(, (n_2: t~n_1), ...) -- e is evaluated only once

Block or MACHINE CODE only for proc types.
In 10. •The ~ may be written as =.
•ENTRY and INTERNAL may be written before t.

```
HistValue: TYPE;                        -- An exported type in an interface.
Histogram: TYPE~REF HistValue;          -- A type binding.
baseHist: READONLY Histogram;           -- An exported variable in an interface.
AddHists: PROC[x, y: Histogram]         -- An exported proc in an interface.
  RETURNS [Histogram];
LabelValue: PRIVATE TYPE~RECORD[        -- PRIVATE only for private stuff in an int.
   first,last:INT,s:ROPE,x:REAL,f,g:INT,r:REF INT];
Label: TYPE~REF LabelValue;
Duration: PROC[l:Label] RETURNS[INT]~   -- An inline proc binding in an interface.
  INLINE { RETURN [l.last l.first] };

-- Decls in an implementation of this interface.
H: TYPE~Histogram[11]; Size: INT~10;    -- A TYPE and an INT binding.
HistValue: PUBLIC TYPE~ARRAY [0..Size]OF INT;PUBLIC only for exported names.
baseHist: PUBLIC H_NEW[HistValue_ALL[17]];An exported variable with initialization.
x, y: HistValue_[ 20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0];
Setup: PROC[bh: Handle[4], a: INT, b: LIST OF H]
  ~ENTRY {...};                         -- An entry proc.
FatalError: ERROR[reason: ROPE]~CODE;   -- Binds an error.
i,j,k: INT_0; p,q: BOOL; lb: Label; main: Handle;
```

Declarations are explained in ¶ 2.3.7. Their peculiarities in the different contexts whe
appear are explained elsewhere:

interfaces in ¶ 3.3.4;

blocks in ¶ 3.4.1;

fields in:

domains and ranges in ¶ 4.4;

records in ¶ 4.6.1;

unions in ¶ 4.6.3.

Bindings are explained in ¶ 2.2.5. There are several special forms of binding given in r
however, which are defined here. See also ¶ 3.7 on argument bindings.

A TYPE binding is the only way in which a type value can be bound to a name, since
cannot be passed as parameters. Unlike other bindings, this one expects a type[36] ra
an expression[19] after the ~.

A name with a signal or error type can be bound to CODE; this use of CODE is not all
anywhere else. The effect is to construct a unique exception value, not equal to a
An enable choice which catches this value will only catch an exception raised with
value; it cannot catch some other expression with the same name. Unfortunately, C
does not yield a unique value at each execution. The value is only unique to the t
occurrence of CODE and the module instance; if CODE appears inside a proc, the same
is produced each time the proc is applied. Thus care may be needed if the proc is

,,A MACHINE CODE construct can be bound to a name with a proc type. This construct
allows machine instructions to be assembled into a proc value. The instructions ar
separated by semi-colons. Each instruction is assembled from a list of expressions
by commas. An expression in the list is evaluated to yield a [0..256) value which
byte of the instruction; successive expressions form successive bytes.

A l-expression derived from a block can be bound to a name with a proc type. The
complicated semantics of this construction are explained in the following subsecti

3.5.1 PROC bindings

A binding of the form n: T~{...} is the only way to construct a proc value and bind it t
since you cannot write a l-expression in current Cedar.

There are other ways to construct proc values:

The expression in a defaultTC[40] is turned into a parameterless proc which is bound to Default in the typ
cluster.

The expression following ~~ in an open or WITH ... SELECT is turned into a parameterless proc with a
deproceduring coercion (¶ 3.4.2).

The statement in an enable choice for a SIGNAL is turned into a proc with domain and range given by the
exception type (¶ 3.4.3.1).

The expression following LOCKS in a MONITOR heading is turned into a parameterless proc according to a
peculiar rule (¶ CONC.???).

The l-expression is constructed from the block in the following way. Its domain and rang
domain and range of the proc type T. Its body declares a variable for each item of the d
range; these variables have the names of the domain and range items, and their scope is
block, not just the block body. The domain variables are initialized to the parameters,
variables in the usual way according to their types. Then the block, with a RETURN tacked
end, is evaluated. A RETURN out of the block is caught, and the current values of the ran
variables are the result of the l-expression. The only other way out of the block is to
ERROR.

A RETURN in the block is sugar for GOTO RETURN, which is caught as described. RETURN e assign
to the range variables and then does a GOTO RETURN.

It is an error to introduce the same name twice in the domain, range or block.

Performance: A proc call and return is about 30% faster if the proc is denoted by a name
bound to a proc body in the same module as the call. A proc which is local to another pr
than bound in the body of an implementation, is about 20% slower to call. It also introd
overhead when its parent proc is called, and its access to non-static names introduced i
proc is slower than access to other names.

The attributes ENTRY and INTERNAL can be used only in a MONITOR; they are discussed in ¶
CONC.???.

The attribute INLINE has no effect on the meaning of the program, but it causes the proc
be expanded inline whenever it is applied. This saves the cost of a proc call and return
of argument passing, and it may allow constant arguments to participate in static evalua
the proc. There are certain restrictions on the use of an INLINE proc:

It may not be recursive.

It may not be used as a proc value except in an application. Thus you cannot, for assign it to a proc variable.

It may not be the argument of FORK.

It may not be exported.

It may not be accessed from the cluster of a POINTER TO FRAME type.

Performance: Excessive use of inline procs will result in much larger compiled code and larger data structures in the compiler. The following cases are efficient:

An inline proc in an implementation which is called exactly once.

An inline proc which has a simple body, no locals, no named results, and no side e

3.6 Statements

```
14 Statement ::= sS                          {SIMPLELOOP {sS; CONTINUE; EXITS  RETRY((=>NULL};
  In 4, 9, 17, 19.                               EXITS CONTINUE((=>NULL }
15 sS ::= e1_e2 | e | block4 | control [e1_e2] | e --must yield VOID-- | --all yield v
16 control ::= GOTO n | GO TO n |            EXORVAL[exception[code~ n((, args~ NIL]] |
  EXIT | CONTINUE | •LOOP | •RETRY |         GOTO (EXIT | CONTINUE | LOOP | RETRY ) |
  REJECT | (RETURN | RESUME) e |             THISEXCEPTION[] |{VALUEOF[rb(]_e;(RETURN|RESUME)}|
  (RETURN | RESUME) |                       EXORVAL[exception[code~(RETURN|RESUME),args~NIL]] |
  „e _ STATE                                 DUMPSTATE[e]

17 loop ::= (iterator | )                    { ( iterator | done(~FALSE, Next(: PROC~{}; )
  (WHILE e | UNTIL e | )                        Test(~l IN (NOT e | e  | FALSE);
  DO  ?•open6 ?•enable7 body9                   { open SIMPLELOOP {
                                                          IF Test([] OR done( THEN GOTO FINISHED;
                                                          { enable body EXITS LOOP=>NULL }; Next([
  ?•(REPEAT (n, !..=>s); !..) ENDLOOP          EXITS EXIT=>NULL; (n, !..=>s);!..; FINISHED=>NULL}}

18 iterator ::= THROUGH  e |                 FOR x(: e IN | e
  FOR (n : t | •n)                           ( n: t; | )
     ( ( | DECREASING) IN e |                done(: BOOL; Range(: TYPE~e;
                                             Next(: PROC~{ IF n ( >LAST | <FIRST )[Range(]
                                               THEN done(_TRUE ELSE n_(PRED | SUCC)[n] };
                                             n_(FIRST | LAST)[Range(]; done(_n NOT IN Range( |
       _ e1 , e2)                            done(: BOOL~FALSE; Next(: PROC~{n_e2}; n_e1 ) ;
  e is a subrange. For n: t, n is readonly except in the assignment in the iterator's desugaring.

x_AddHists[baseHist, baseHist]^;          -- A stat can be an assignment,
Setup[bh~main, a~3, b~CONS[...] ];        -- or a proc without results,
{ENABLE FatalError=>RETURN[0]; []_f[3];   -- or a block,
IF i>3 THEN RETURN[25] ELSE GOTO NotPresent;  -- or an IF or a control statement,

FOR t:INT DECREASING IN [0..5) UNTIL f[t]>3 DO  -- or a loop. Try to declare t  in the FOR as
  u: INT_0; ... ; u_t+4; ...              -- shown. Avoid OPEN, ENABLE after DO
  REPEAT Out=>{...}; FINISHED=>{...} ENDLOOP;  (use a block). FINISHED must be last.

THROUGH [0..5) DO ... ENDLOOP;
```

Cedar makes a distinction between expression and statements. This distinction is most ea understood in terms of a special type called VOID. This is the range type of a PROC [...]_ also the type of a block, control, loop or NULL statement. An expression whose value is a

be used as a statement, and cannot be used as an ordinary value in a binding (since it w
have the right type). If you want to call a proc which returns values as a statement, yo
the results to an empty group:

```
[]_f [...]
```

Assignment is a special case; an assignment can be used as a statement even though its v
value of the right operand. This is explained in the desugaring[15] using a special proc TO
cluster of every assignable type; it takes a value of the type and returns a VOID.

Anomaly: In a select[29] which is a statement (i.e., returns VOID), the choices are separate
semicolons; in an ordinary select expression they are separated by commas.

Anomaly: •If you write an expression whose value is a proc taking no arguments as a stat
proc gets applied. Thus

```
P;
```

is the same as

```
P[];
```

This is the only situation in which an ordinary proc gets applied by coercion (but see ¶
open procs).

A statement[14] is actually a more complicated construct than you might think, as the desug
shows. This is because of the CONTINUE and RETRY statements, which respectively terminate a
repeat the current statement. The desugaring shows exactly what this means in various ob
cases. CONTINUE and RETRY are legal only in an enable choice (¶ 3.4.2), and they may not ap
a declaration at all. •RETRY should be avoided everywhere, since it introduces a loop int
program is a distinctly non-obvious way.

Control[16] consists mainly of the various flavors of GOTO (including EXIT, CONTINUE, LOOP, RETR
RETURN and RESUME) which raise a local exception bound in an EXITS; this is explained in ¶ 3
REJECT is explained in ¶ 3.4.3.1.

Anomaly: Note that you cannot use a GOTO to escape from a proc body, even though the body
within the scope of the label. Only normal completion, or a RETURN or ERROR exception (or a
SIGNAL which is not resumed) can terminate a proc body.

A loop[17] is repeated indefinitely until stopped by an exception, or by the iterator[18] or t
UNTIL test. It has a body, bracketted by DO and ENDLOOP, which is almost like a block, but
some confusing differences:

> You catch GOTO exceptions with REPEAT, which is exactly like EXITS in a block immedia
> around the loop,except for the different delimiting reserved word. Note that the s
> the labels does not include the iterator or the test, even though these are evalua
> repeatedly during execution of the loop. This feature is best avoided, but unfortu
> necessary if you want to catch the FINISHED exception explained below.
>
> •You can write an open or enable. This is also best avoided, since the scope is co
> is better to write a block explicitly inside the DO if you need these facilities.

There are three special exceptions associated with loops:

> EXIT is equivalent to GOTO EXIT, where EXIT is a label automatically declared in the r
> of every loop. Its enable choice does nothing. Thus EXIT simply terminates the smal
> that encloses it.
>
> FINISHED is raised when the iterator or the WHILE/UNTIL test terminates the loop. It c
> declared in the REPEAT like any label, but it must come last. If it is not declared
> enable choice is supplied for it. Anomaly: You cannot write GOTO FINISHED.
>
> •LOOP causes the next repetition of the loop to start immediately.

An iterator[18] declares a control variable v which is initialized by the iterator and upda
execution of the loop; the scope of v is the entire loop, and it is read-only in the loo
loop is terminated by the iterator, the value of v is undefined. •If you omit the declar
simply name an already declared variable, it will be used as the control variable, and w
read-only.

There are three flavors of iterator:

THROUGH, which has no explicit control variable; THROUGH [0..k) is convenient when yo
just want to loop k times.

FOR v: T IN [first, last] ...; v is initialized to first, and set to SUCC[v] after eac
iterator stops the loop after a repetition which leaves v>last. DECREASING reverses t
in which the elements of the subrange are used. The subrange need not be static. N
the subrange is evaluated only once, before execution of the loop begins.

FOR v: T_first, next ...; v is initialized to first, and set to next after each repet
iterator never stops the loop. Note that the expression next is reevaluated each ti
the loop. The usual application is something like

```
FOR v: List_header, v.next UNTIL v=NIL.
```

Note that the WHILE test is made with v equal to its value during the next repetition, an
tests are made before the first repetition, so that zero repetitions are possible.

3.7 Expressions

19 expression ::= n | literal | application[26] |
e . n | LOOKUP Z [De, $n] [e] |
builtIn [e1 ?(, e2, !..)] | funnyAppl | e1.builtIn ?([e2, ...]) | e . funnyAppl(|
prefixOp e | e1 infixOp e2 | e1 NOT relOp e2 | e1.infixOp[e2] | e1.(NOT relOp)[e2] |
e1 AND e2 | e1 OR e2 | IF e1 THEN e2 ELSE FALSE | IF e1 THEN TRUE ELSE e2 |
e ^ | •STOP | ERROR | e . DEREFERENCE | STOP[] | ERROR NAMELESSERROR |
[argBinding[27]] | s | --Binding must coerce to a record, array, or •local
subrange | if | select | safeSelect | µwithSelect

Precedence is noted in bold in the operator rules. All operators associate to the left except _, which asso
to the right. ^, . and application have higher precedence than any Op. AND has precedence (2) and OR has
precedence (1). Subrange only after IN. s only in IF[28] and in SELECT[29] choices.

20 prefixOp ::= @ (8) | (7) | (~ | NOT) (3) | VARPOINTER | UMINUS | NOT
21 infixOp ::=
* | / | MOD (6) | + | (5) | relOp (4) | TIMES | DIVIDE | REM | PLUS | MINUS | ASSIGN
22 relOp ::=
= | # | < | <= | > | >= | IN -- In 19, 21, 30 | EQUAL | NOT = | LESS | NOT > | GREATER | NOT < | IN
23 builtIn ::= -- These are enumerated in Table 4 5.
24 funnyAppl ::= FORK | JOIN | SIGNAL | ERROR | NEW | •START | •RESTART | WAIT | NOTIFY |
BROADCAST | RETURN WITH ERROR | „,TRANSFER WITH | „,RETURN WITH

25 subrange ::= (typeName[37] |) (
([e1 .. e2] | (e1 .. e2) |
(e1 .. e2] | (e2))
In 19, 39, 55

```
LET t(~(typeName | INT) IN (
  t(.MKSUBRANGE[e1, (e2 | PRED[e2] )] BUT
      {BoundsFault=>t(.MKEMPTYSUBRANGE[e1]} |
  t(.MKSUBRANGE[SUCC[e1], (e2 | PRED[e2] )] BUT
      {BoundsFault=> t(.MKEMPTYSUBRANGE[SUCC[e1]] } )
```

26 application ::= e [argBinding
?(! eChoice)]
27 argBinding ::= (n ~ (e | | µTRASH))
(e | µTRASH |), ...

```
LET map(~e, args(~[ argBinding
  map(.APPLY Z args(.BUT { eChoice } ) )
(n ~ (e | OMITTED | TRASH) ), ... |
(e | OMITTED | TRASH ), ...
```

In 19, 26. •The ~ may be written as :.
• TRASH may be written as NULL.

```
lv: LabelValue13_[ i, 3, "Hello", 31.4E-1, g[x]+lb.f+PRED[j] ];
                                        -- A constructor with some sample
                                        -- expressions.
p1: PROCESS RETURNS [INT]_FORK f[i, j];  -- FunnyAppls take one unbracketted arg;
ERROR NoSpace; WAIT bufferFilled;       -- many return no result, so must be stats.
RT: RTBasic.Type_CODE[LabelValue13];
h[ 3, NOT(i>j), i*j, i_3, i NOT >j, p OR q, lv19_[first~0, last~10, x~3.14, g~2, f~5]];
                                        -- An application with sample expressions.
                                        -- Short for lv_LabelValue13[...].
[first~i, last~j]_lv19;                 -- Assignment to a VAR binding (extractor).

b: BOOL_i IN [1..10]; FOR x: INT IN (0..11) DO ...;
                                        -- Subrange only in types or with IN.
b_( c IN Color45(red..green] OR x IN INT[0..10));
                                        -- The 0 is redundant.

fh_Files.Open[name~lb.s, mode~Files.read
  ! AccessDenied=>{...}; FatalError=>{...}];
                                        -- Keywords are best for multiple args.
                                        -- Semicolons separate choices.
(GetProcs[j].ReadProc)[k];              -- The proc can be computed.
file.Read[buffer~b, count~k];           -- WFile.Read[file, b, k] (object notation).
f[i~3, j~ , k~TRASH]; f[i~3, k~TRASH]; -- j and k may be trash (see defaultTC40).
f[3, , TRASH];                          -- Likewise, if i, j, and k are in that order.
```

Most of the forms of expression are straightforward sugar for application: prefix, infix operators, explicit application of a primitive function, or the funnyAppl24 in which the argument follows the proc name without any brackets. All of these constructs desugar int notation (¶ 2.4.4, ¶ 4.14); this means that the procs come from the cluster of the first exceptions to this rule are ALL, CONS for variant records and lists, LIST, and the single forms of LOOPHOLE and NARROW; all of these get the proc from the target type of the express 4.2.4). All the primitive procs are described in Chapter 4.

Note that AND and OR are not simply sugar for application. Rather, they are sugar for an expression, since the second operand is evaluated only if the first one is TRUE or FALSE r

Rules 19-21 give the precedence for operators: @ is highest (binds most tightly) and _ i All are left-associative except _, which is right-associative. The ^, . and [] (applicat have still higher precedence. These rules are sufficiently complex that it is wise to pa expressions which depend on subtle differences in precedence.

The first operand of assign can be an argBinding27 whose value is a variable group or bir one whose elements are variables; this is sometimes called an extractor. The second argu typecheck if it coerces to a group or binding with corresponding elements which can be a the variables. Usually the second argument is either an application which returns more t result or a record constructor.

Anomaly: In the second case, the fact that the order of evaluation in an expression is n over-exploited: some of the variables may be changed before all the elements of the cons evaluated.

A funnyAppl which takes more than one argument has the extra arguments written inside br in the usual way; e.g., START P [3, "Help"].

Anomaly: Enable choices are legal only for the following: FORK JOIN RESTART START STOP WAIT. You can write empty brackets if neccessary to get a place for the eChoices.

A subrange25 denotes a subrange type. Standard mathematical notation for open and closed intervals is used to indicate whether the endpoints are included in the subrange. A subr also be used after IN in an expression or iterator; in these contexts it need not be stat

You can write enable choices[8] after a ! inside the brackets of an application[26]. See ¶ 3.
semantics of this. Note that only an exception returned by the application is caught by
choices, not one resulting from evaluating the proc or arguments.

An argBinding[27] denotes a binding for the arguments of an application. You can omit a [na
value] pair n~e in the binding if the corresponding type has a default, or you can write
without the value expression (e.g., n~) with the same meaning. You can also write TRASH
NULL) for the value; this supplies a trash value for the argument (¶ 4.11).

3.8 IF and SELECT

[28] if ::= IF e_1 THEN e_2 (ELSE e_3 |)	IF e_1 THEN e_2 ELSE (e_3 | NULL)
[29] select ::= SELECT e FROM choice; ... endChoice	LET selector(~e IN choice; ... endChoice
The separator written ";" is "," in an expression.	ELSE is a separator for repetitions of the choice.
[30] choice ::= ((|relOp[22]|NOT relOp[22]) e_1), (...) => e_2	IF selector((= | relOp | NOT relOp) e_1) OR ...) THEN e_2
[31] endChoice ::= ENDCASE (| => e)	ELSE (NULL | e)
In 29, 32, 34.	
[32] safeSelect ::= WITH e SELECT FROM safeChoice; ... endChoice[30]	LET v(~e IN safeChoice; ... endChoice
[33] safeChoice ::= n : t => e	IF ISTYPENOTNIL[v(, t] THEN LET n : t_NARROW[v(, t] IN e
[34] •withSelect ::= WITH (n_1 ~~ e_1 | • e[1]), SELECT (| ,e_2) FROM withChoice; ... endChoice[30]	OPEN v(~~e_1 IN LET n(~(n_1 | NIL), type(~De$_1$, selector(~(e_1.TAG | e_2) IN withChoice; ... endChoice -- e_2 must be defaulted except for a COMPUTED variant
•The ~~ may be written as :.	
[35] •withChoice ::= n_2 => e | n_2, n_2, !.. => e	IF selector(=n_2 THEN OPEN (BINDN[n(,LOOPHOLE[v(,type([n_2]]] | BINDN[n(, v(]]
i_(IF j<3 THEN 6 ELSE 8);	-- An IF with results must have an ELSE.
IF k NOT IN Range THEN RETURN[7]; SELECT f[j] FROM	-- SELECT expressions are also possible.
<7=>{...};	-- Wt:INT~f [j]; IF t<7 THEN {...} ELSE ...
IN [7..8]=>{...};	-- 7, 8=> or =7, =8=>{...} is the same.
NOT<=8=>{...};	-- ENDCASE=>{...} is the same here.
ENDCASE=>ERROR;	-- Redundant here: choices are exhaustive.
WITH r SELECT FROM	-- Assume r: REF ANY in this example.
rInt: REF INT=>RETURN[Gcd[rInt^, 17]];	-- rInt is declared in this choice only.
rReal: REF REAL=>RETURN[Floor[Sin[rReal^]]];	
ENDCASE=>RETURN[IF r=NIL THEN 0 ELSE 1]	-- Only the REF ANY r is known here.
nr: REF Node[49]~...; WITH dn~~nr SELECT FROM	See rule 49 for the variant record Node.
binary=>{nr_dn.b};	-- dn is a Node[binary] in this choice only.
unary=>{nr_dn.a};	-- dn is a Node[unary] in this choice only.
ENDCASE=>{nr_NIL};	-- dn is just a Node here.

The kernel construct if[28] evaluates the expression e_1 to a BOOL value test, and then evalua
test=TRUE, or e_3 if test=FALSE. In the expression

IF $test_1$ THEN IF $test_2$ THEN $ifTrue_2$ ELSE $ifFalse_2$

the grammar is ambiguous about which IF the ELSE belongs to. It belongs to the second one

A select[29] is a sugared form of if which is convenient when one of several cases is chose
a single value. The selector expression e is evaluated once, and then each of the choice
turn. Within each choice, each expression e_1 preceding the => is compared in turn with th

selector; if any comparison succeeds, the expression e_2 following the => is evaluated to
value of the select. If no comparison succeeds, the next choice is tried. If no choice s
expression e following the ENDCASE is evaluated to yield the value of the select; e defaul
and hence must be present when the select is not a statement to prevent a type error.

The comparison is selector relop e_1 if e_1 is preceded by a relop; otherwise it is selector=e

Style: It is good practice to arrange the tests so that they are disjoint and exhaust th
values of the selector. ENDCASE should be used to mean "in all other cases"; often the app
e_2 raises an error. Don't use ENDCASE when you mean another specific selector value which
don't bother to mention.

Performance: If the e_2 are static and select disjoint subsets of the selector values, and
size of these subsets is not too large, a select compiles into an indexed jump, which ex
time independent of the number of choices. This also happens if a contiguous subset of t
has this property.

A safeSelect[32] is a special form for discriminating cases of unions or ANY. The selector
value for which ISTYPE can be evaluated dynamically (¶ 4.3.1): REF ANY, PROC ANY_T, PROC
T_ANY, V or REF V, where V is a variant record. Each choice specifies one possible type th
selector might have, and declares a name which is bound to the selector value if it has
Thus, the example tests for r having the types REF INT and REF REAL. If it has REF INT, the
choice's e is evaluated; within e, rInt is bound to the selector, and has type REF INT. Li
REF REAL and the second choice. As with an ordinary select, the ENDCASE expression is evalu
(with no new names known) if none of the other choices succeeds. Note that safeSelect do
ordinary binding by value, not the binding by name done in open and withSelect.

•,A withSelect[34] is an unsafe and rather tricky construction for discriminating cases of
use should be avoided unless a safeSelect can't do the job; this is the case for a COMPUT
if the call by name feature of withSelect is required.

It incorporates an open (¶ 3.4.2) of the e_1 being discriminated. This means that e_1
dereferenced to yield a variant record value. It also means that this value is not
hence it can change its type during execution of a choice, either by assignment to
variant part of a variant record (itself an unsafe operation), or by a change in t
e_1.

If the union has a COMPUTED tag, the selector value to be used for the discriminatio
be given as e_2 in the withSelect. It is entirely up to the programmer to supply a
value. If the tag is not COMPUTED, e_2 must be omitted and the selector value is e_1.TA

The n_2 preceding => in a choice are literals of the (enumerated) type (¶ 4.7.1.1) w
the tag type of the union (¶ 4.6.3). They are compared with the selector, and if o
the e following => is evaluated as with an ordinary select. If exactly one is give
e following => is in the scope of

```
OPEN n1~~LOOPHOLE[e1.DEREF, V[n2]];
```

or simply

```
OPEN LOOPHOLE[e1.DEREF, V[n2]]
```

if no n_1~~ followed the WITH. If several n_2 are given, then there is no discriminati
the e following => is in the scope of

```
OPEN n1~~e1.DEREF     or     OPEN e1.DEREF
```

3.9 Types

This section gives the syntax for type constructors, together with a number of examples. information about the use of the constructors and the primitive procs available for each found in Chapter 4.

36 type ::= typeName | builtInType | typeCons
37 typeName ::= n_1 ?(. n_2) | •n_3 ... typeName(n_1 ?(. n_2) | typeName ([n_3]) ... --n_3 names a variar
In 25, 36.

38 builtInType ::= -- See table TYP 2.
?LONG (INTEGER | CARDINAL) | INT | NAT | REAL | „WORD | TYPE |
ATOM | MONITORLOCK | CONDITION | µ ?,UNCOUNTED ZONE | •, ?LONG UNSPECIFIED
TYPE only in a body's binding or an interface's decl. BOOL and CHAR are predefined enumerated types.

39 typeCons ::= subrange[25] | typeName[37]([e])...
varTC[39] | defaultTC[40] | transferTC[41] |
enumTC[45] | recordTC[46] | unionTC[49] | arrayTC[51] | seqTC[51a] |
•descriptorTC[52] | refTC[53] | listTC[54] | ,pointerTC[55] | relativeTC[55a]

```
P: PROC[
  b: Buffer[1].Handle,                  -- A type from an interface.
  i: INT_SIZE[TEXT[20]] ];              -- A bound sequence; only in SIZE, NEW.

TypeIndex: TYPE~[0..256);               -- A subrange type.
BinaryNode: TYPE~Node[49][binary];      -- A bound variant type.
```

39.1 varTC ::= (| READONLY | VAR) (t | (ANY))
In 11, 47, 52, 53, 54, 55. ANY only in refTC. VAR only in interface decl.

```
40 defaultTC ::=                        CHANGEDEFAULT[type~t, (
  t _ |                                             proc~NIL,trashOK~FALSE] |
  t _ e |                                           proc~INLINE l IN e,trashOK~FALSE] |
  µt _ e | TRASH |                                   proc~INLINE l IN e,trashOK~TRUE] |
  µt _ TRASH                                        proc~NIL,trashOK~TRUE] )
```
defaultTC legal only as the type in a decl in a body[9]
or field[44] (n: t _ e), or in NEW. Note the terminal |.
•TRASH may be written as NULL.

```
-- Except as noted, a constructor or application must mention each name and give it a va
Q: RECORD[                              -- Otherwise there's a compile-time error.
  i: INT,                               -- Q[] or Q[i~ ] leaves i trash (not for proc).
  j: INT_,                              -- No defaulting or trash for j.
  k: INT_3,                             -- Q[] or Q[k~ ] leaves k=3.
  l: INT_3 | TRASH,                     -- As k, but Q[l~TRASH,...] leaves l trash.
  m: INT_TRASH ];                       -- Q[] or Q[m~ ] leaves m trash.
```

41 transferTC ::=?SAFE transferFlavor drType MKTYPE[drType, flavor~transferFlavor]
42 transferFlavor ::= (PROCEDURE|PROC|PORT|
PROCESS | SIGNAL | ERROR | PROGRAM)
43 drType ::= ?fields$_1$?(RETURNS fields$_2$) domain~fields$_1$, range~fields$_2$
No domain for PROCESS. In 3, 41.
44 fields ::= [d, ...] | [t, ...] | ANY
ANY only in drType. In 43, 46, 49.

```
Enumerate: PROC[
   l: RL,
```

```
   p: PROC[x: REF ANY] RETURNS [stop: BOOL]]
  RETURNS [stopped: BOOL];
p2:PROCESS RETURNS[i:INT]_FORK stream.Get;
failed: ERROR [reason: ROPE]~CODE;

45 enumTC ::= { n,  !.. } |                    MKENUMERATION[ [$n, ...] ] |
  MACHINE DEPENDENT {( (n | )e2)(), !.. }      MKMDENUMERATION[ [ [($n | NIL), e], ... ] ]

Op: TYPE~{plus, minus, times, divide };
Color: TYPE~MACHINE DEPENDENT{                 -- A Color value takes 4 bits; greenW1.
  red(0), green, blue(4), (15)}; c: Color;

46 recordTC ::=
  ?access12 ?MONITORED  RECORD fields44 |      MKRECORD[ fields]s|
  „ ?access12 MACHINE DEPENDENT
       RECORD ( mdFields | •fields44 )         MKMDRECORD[( mdFields| fields)]s
  Any unionTC in fields must come last.
47 mdFields ::=
  [( (n pos), ... : ?•access12 varTC39),...]   MDFIELDS[ ( [ ([$n, (pos | NIL)] ), ... ] , t ), ... ]
  In 46, 49.
48 pos ::=e1( ?(: e2 .. e3) )                  MKPOSITION[firstWord~e1, firstBit~e2, lastBit~e3]
  In 47, 50.

Cell: TYPE~RECORD[next: REF Cell, val: ATOM];
Status: TYPE~MACHINE DEPENDENT RECORD [        -- Don't omit the field positions.
  channel (0: 8..10): [0..nChannels),          -- nChannels < 8.
  device (0: 0..3): DeviceNumber,              -- DeviceNumber held in < 4 bits.
  stopCode (0: 11..15): Color, fill (0: No gaps allowed, but any ordering OK.
  command (1: 0..31): ChannelCommand ];        -- Bit numbers >16 OK; fields can cross
                                               -- word boundaries only if word-aligned.

49 unionTC ::=  SELECT tag FROM                MKUNION[selector~tag, variants~[ [ labels~[ $n, ...],
  ( n, ... => ( fields44 | mdFields47          | •NULL) ), ...        value~fields ], ...] ]
     ?, ENDCASE
    Legal only as last decl in a recordTC or unionTC.
50 tag ::= (n ?„pos48 : ?•access12 |            [ [ ( $n, pos | $COMPUTED( | $OVERLAID( ) ] ,
  µ,COMPUTED | µ,OVERLAID ) (t | *)                ( t | TYPEFROMLABELS ) ]
  In 49, 51a.

Node: TYPE~MACHINE DEPENDENT RECORD [          -- rands is a variant part or union.
  type (0: 0..15): TypeIndex,                  -- This is the common part.
   rator (1: 0..13): Op45,
  rands (1: 14..79): SELECT n (1: 14..15) FROM -- Both union and tag have pos.
    binary=>[a (1:16..47), b (1:48..79)],      -- At least one variant must fill 1: 14..79.
    unary=>[a (1: 16..47): REF Node],          -- Can use same name in several variants.
    nonary=>[] ENDCASE ];                      -- Type of n is {binary, unary, nonary}.

51 arrayTC ::= ?µPACKED ARRAY ?t1 OF t2        MKARRAY[domain~t1, range~t2]
51.1seqTC ::= ?µPACKED SEQUENCE tag50 OF t     MKSEQUENCE[domain~tag,  range~t]
    Legal only as last decl in a recordTC or unionTC.
52 •,descriptorTC ::=
  ?LONG DESCRIPTOR FOR varTC39                 MKARRAYDESCR[arrayType~varTC] |
  The varTC must be an array type.

Vec: TYPE=ARRAY [0..maxVecLen) OF REF TEXT;

v: Vec~ALL[NIL];
```

```
Chars: TYPE~RECORD [text: PACKED SEQUENCE -- A record with just a sequence in it.
   len: [0..LAST[INTEGER]] OF CHAR]; ch: Chars;  -- chars.text[i] or ch[i] refers to an element.
dV: DESCRIPTOR FOR ARRAY OF REF TEXT~

53 refTC ::= REF (varTC39 | • )          MKREF[range~(varTC | ANY )]
54 listTC ::= LIST OF varTC39            MKLIST[value~varTC]

ROText: TYPE~REF READONLY TEXT;          -- NARROW[rl.first, ROText]^ is a
RL: TYPE~LIST OF REF READONLY ANY; rl:RL; -- READONLY TEXT (or error).

55 ,pointerTC ::= ?LONG ?•ORDERED ?BASE  MKPOINTER[range~varTC] |
     POINTER ?subrange25 ?(TO varTC39) |
  POINTER TO FRAME [ n ]                 xxx |
  Subrange only in a relativeTC; no typeName on it.
55.1,relativeTC ::= t2 RELATIVE t1       MKRELATIVE[range~t1, baseType~t2]
  t1 must be a pointer or descriptor TC, t2 a typeName for a base pointer.

UnsafeHandle: TYPE~LONG POINTER TO Vec51;
```

Chapter 4. Primitive types and type constructors

This chapter gives detailed information about the primitive types and type-returning pro constructors). It should be read after ¶ 2.3, which defines a Cedar type and explains th underlying the type system. The implies relations on primitive types are summarized in ¶ the coercions in ¶ 4.13.

¶ 4.1 gives the partial ordering called the class hierarchy that is used to classify the 4.2 lists all the primitives of Cedar. ¶¶ 4.3-4.10 give the declarations and semantics f primitive classes and types. These descriptions are ordered according to the class hiera 4 1. Each one specifies:

The constructor for types in the class.

Any literals or basic constructors for values of types in the class

The declarations in the class that are not in any bigger class.

Anomalies and facts about performance.

4.1 The class hierarchy

A useful way of organizing a set of types is in terms of the properties of their cluster cluster is a binding, its type is a declaration; we call such a declaration a class. For class NUMERIC is

```
[T: TYPE;
 PLUS: PROC[T, T]_[T];
 MINUS: PROC[T, T]_[T];
. . . -- Declarations for other arithmetic procs.
 LESS: PROC[T, T]_[BOOL];
. . . -- Declarations for many other procs. ]
```

By convention, the name T in a cluster denotes the type to which the cluster belongs.

A type T is in a class C if T.CLUSTER has the type C; we also say that T is a C type, e.g. class NUMERIC, or is a numeric type. To make this explicit, we give the type CLASS a cluste called TYPE, such that every type T in class C has type C.TYPE. For example, INT has type NUMERIC.TYPE. Thus,

```
T is a C typeWT in C WT has type C.TYPEW(C.TYPE).PREDICATE[T]=TRUE
```

A value satisfies the predicate for C.TYPE if it is a type, and its cluster satisfies the defines C. E.g., INT satisfies the predicate for NUMERIC.TYPE because it is a type, and its contains procs for PLUS, MINUS, LESS etc. with the right types. Precisely, (C.TYPE).PREDICATE

```
l [T: ANY] IN TYPE.PREDICATE[T] & C.PREDICATE[T.CLUSTER]
```

A class C is a subclass of another class D if CgD. Recall the implies relation for declar that

```
Each name n in C is also in D.
n's type in D implies n's type in C.
```

Precisely,

```
(AnBC.names) nBD.names & (D.ToBinding.ngC.ToBinding.n)
```

For example, the class ORDERED includes

```
LESS: PROC[T, T]_BOOL
```

Every subclass of ORDERED must also declare a LESS proc which takes two T's to a BOOL. If w a richer assertion language, there would also be axioms defining LESS to be an ordering r Similarly, every ORDERED type (e.g., INT) must have such a LESS proc in its cluster.

The subclass relation defines a class hierarchy, i.e., it gives a partial ordering on cl gives the class hierarchy for the primitive classes of Cedar. It is presented as a tree: sons N_1, N_2, ..., N_k is written

N N_1 | N_2, |... | N_k

and if any of the N_i are not leaves, they are defined on following indented lines:

N_i N_{i1} | N_{i2}, |...

In fact, however, the class hierarchy is not a tree but a partially ordered set; hence s
appear more than once in the table, with appropriate cross-references. Classes produced
type constructors are named by the constructors; other, more general classes are given s
names, sometimes lower-case versions of the constructor names. Each primitive type also
the table, under its class in the tree.

Class	Subclasses or types
all	general* | TYPE, ¶ 4.8 | fully opaque ¶ 4.3.4 | SEQUENCE
general ¶ 4.3.1	assignable* | variable ¶ 4.3.3 | PORT_transfer | MONITORLOCK ¶ 4.9 | CONDITION ¶ 4.9 | ARRAY_row, RECORD or union with a non-assignable component
assignable ¶ 4.3.2	everything not mentioned separately under all or general, i.e.:-- n-opaque ¶ 4.3.4 | transfer_map | descriptor_map | address | RELATIVE | ordered | unspecified | ARRAY_row, RECORD or union without a non-assignable component.
has NIL ¶ 4.3.3	variable_general | address | transfer_map
map ¶ 4.4	transfer* | row* | descriptor*/address | BASE POINTER/pointer | TYPE,_all
transfer ¶ 4.4.1	PROC | PORT | PROGRAM | PROCESS | SIGNAL | ERROR
row ¶ 4.4.2	ARRAY ¶ 4.2.1 | SEQUENCE/union--second class-- ¶ 4.2.2J(TEXT, | StringBody,)
descriptor ¶ 4.4.3 /address	LONG DESCRIPTOR | DESCRIPTOR
address ¶ 4.5	reference* | descriptor_map | ZONE ¶ 4.5.2 | POINTER TO FRAME ¶ 4.5.3
reference ¶ 4.5.1	REFJ(LIST | ATOM,) ¶ 4.5.1.1 | pointer*
pointer /ordered ¶ 4.5.1.2	LONG POINTERJLONG STRING, | POINTERJSTRING, | BASE POINTER_map
RELATIVE ¶ 4.5.4	RELATIVE POINTER | RELATIVE DESCRIPTOR
record --painted-- ¶ 4.6	RECORD ¶ 4.6.1 | variant ¶ 4.6.2
union --second class-- ¶ 4.6.2	
ordered ¶ 4.7	discrete* | numeric* | pointer_address | subrange ¶ 4.7.3
discrete ¶ 4.7.1	whole number_numeric | enumeration --painted-- ¶ 4.7.1.1J(BOOLWBOOLEAN, | CHARWCHARACTER,)
numeric ¶ 4.7.2	whole number*/discrete | REAL, ¶ 4.7.2.2
whole number ¶ 4.7.2.1	long number=(INTW•LONG INTEGER, | LONG CARDINAL,) | short number*
short number	INTEGER,JNAT, | CARDINAL,JNAT,
CONDITION, | MONITORLOCK, | unspecified ¶ 4.9	J(UNSPECIFIED, | LONG UNSPECIFIED,)
--kernel only-- exception | DECL | BINDING ¶ 4.10	

Notation:

n*	n is further specified in one of the indented lines below.
n,	n is a type, rather than a class.
n_m	n has its main definition under (and implies) class m.
n/m	n also appears under (implies) class m.
n=e | ...	n includes (is implied by) the e classes, which together exhaust n.
nJe | ...	n includes (is implied by) the e classes, which are special cases.

Table 4 1: The class hierarchy

4.2 Type-related primitives

The tables in this section summarize the primitive and predeclared types, type construct procs of Cedar.

4.2.1 Primitive types and constructors

Table 4 2 lists the primitive or predeclared types of Cedar, giving the name for each in
language, and either a definition or, for the primitive types, a comment suggesting the
the type. Later sections describe additional procs in the clusters of these types, and g
representations.

Name	Meaning
INT, LONG INTEGER ¶ 4.7.2.1	=[2^{31}..2^{31})
REAL ¶ 4.7.2.2	-- 32-bit IEEE floating point
BOOL, BOOLEAN ¶ 4.7.1.1	={FALSE, TRUE}
CHAR, CHARACTER ¶ 4.7.1.1	={'\000, ..., '\377}
TYPE ¶ 4.8	
ATOM ¶ 4.5.1.1	-- for unique strings, global property lists
CONDITION ¶ 4.9.1	-- for process synchronization
-- The following are appropriate when performance tuning is needed.	
µINTEGER ¶ 4.7.2.1	=[2^{15}..2^{15}); SIZE[INTEGER]=1
µNAT ¶ 4.7.2.1	=INTEGER[0..2^{15}); SIZE[NAT]=1
µTEXT ¶ 4.4.2.2	=MACHINE DEPENDENT RECORD [length (0): [0..LAST[INTEGER]] _ 0, text (1): PACKED SEQUENCE maxLength (1): [0..LAST[INTEGER]] OF CHAR]
µZONE ¶ 4.5.2	-- controls safe storage allocation
-- The following are not recommended for general use.	
µMONITORLOCK ¶ 4.9.1	-- use MONITOR or MONITORED RECORD
,UNCOUNTED ZONE ¶ 4.5.2	-- controls unsafe storage allocation
LONG CARDINAL ¶ 4.7.2.1	=[0..2^{32}), mixes poorly with INT.
CARDINAL ¶ 4.7.2.1	=[0..2^{16}); SIZE[CARDINAL]=1
-- The following are obsolescent.	
•,MDSZone	-- controls unsafe storage allocation in the MDS.
•,?LONG STRING ¶ 4.4.2.2	=?LONG POINTER TO StringBody
•StringBody ¶ 4.4.2.2	MACHINE DEPENDENT RECORD [--see text for anomalies-- length (0): CARDINAL _ 0, maxLength (1): --READONLY-- CARDINAL, text (2): PACKED ARRAY [0..0) OF CHAR]
•,UNSPECIFIED ¶ 4.9.2	-- unsafe, matches any one-word type
•,LONG UNSPECIFIED ¶ 4.9.2	-- unsafe, matches LONG INTEGER, LONG CARDINAL, LONG POINTER, or REF.

Table 4 2: Primitive and predeclared types

4.2.2 Type constructors

Table 4 3 gives the declarations of all the primitive Cedar type constructors. Since typ
procs cannot be written in the current language, these are in fact all the Cedar type co
The concrete syntax for invoking these constructors is given in the grammar (rules 40-55
4.2.2.1 on options.

All the arguments of a type constructor must be static (¶ ???). The only exception is MKS
which can have non-static arguments when it appears in an expression or iterator as the
operand of IN.

Name	Domain	Class of result	Rule	¶
MKSUBRANGE	[FIRST: T, LAST: T]	subrange	25	¶ 4.7.3
-- T is the DISCRETE base type, which has a MKSUBRANGE type constructor in its cluster.				
CHANGEDEFAULT	[type: TYPE, proc: (PROC[]_type), allowTrash: general]	general	40	¶ 4.3.1
MKXFERTYPE	[flavor:{PROC,PORT,PROCESS,SIGNAL,ERROR,PROGRAM}, domain, range: DECL_NIL, safe: BOOL_ISCEDAR]	transfer	41	¶ 4..4.1
MKPROC	[domain, range: DECL_NIL, safe: BOOL_ISCEDAR]	PROC	41	¶ 4..4.1
~MKXFERTYPE[domain, range, PROC, safe]				
MKENUMERATION	[LIST OF ATOM]	enumeration	45	¶ 4.7.1.1
MKMDENUMERATION	[LIST OF RECORD[ATOM, NAT]]	enumeration	45	¶ 4.7.1.1
MKRECORD or MKMDRECORD	[fields: DECL, access: {PUBLIC, PRIVATE}_CURRENTACCESS, monitored: BOOL_FALSE]	RECORD	46	¶ 4.6.1
MKPOSITION	[firstWord: NAT, firstBit: NAT_0, lastBit: nat_15]		48	¶ 4.6.1
MKUNION	[selector: TAG, variants: ?????	union	49	¶ 4.6.3
MKSEQUENCE	[domain: TAG, range: TYPE, packed: BOOL_FALSE]	SEQUENCE	51	¶ 4.4.2.2
MKARRAY	[domain: DISCRETE.TYPE_CARDINAL, range: TYPE, packed: BOOL_FALSE]	ARRAY	51	¶ 4.4.2.1
•MKARRAYDESCR	[arrayType: ARRAY.TYPE, long: BOOL_FALSE, readOnly: BOOL_FALSE]	DESCRIPTOR	52	¶ 4.4.3
MKREF	[range: TYPE, base: BASE_WORLD, readOnly, uncounted: BOOL_FALSE]	REF	53	¶ 4.5.1.1
MKLIST	[componentType: TYPE, readOnly: BOOL_FALSE]	LIST	54	¶ 4.5.1.1
MKPOINTER	[range: TYPE_UNSPECIFIED, long, readOnly, ordered, base: BOOL_FALSE]	pointer	55	¶ 4.5.1.2
~MKREF[range~range, readOnly~readOnly, uncounted~TRUE, base~(IF long THEN WORLD ELSE MDS)].				
µMKRELATIVE	[range: TYPE, baseType: BASE.TYPE]	RELATIVE	55	¶ 4.5.4

Table 4 3: Primitive type constructors

4.2.2.1 Options

The built-in type constructors take an assortment of optional BOOLEAN arguments, as indica
their declarations. In the current syntax these are specified by writing options in the
constructor. When an option appears in a type constructor, the argument of the same name
value TRUE; if it is missing, the argument has the value FALSE. The effect of these argume
type produced by the constructor is given as part of the description of its result class
the options and the constructors for which each is appropriate.

Option	Constructors
µBASE	MKPOINTER
LONG	MKPOINTER, MKARRAYDESCR, INTEGER, CARDINAL
MONITORED	MKRECORD
•ORDERED	MKPOINTER
µPACKED	MKARRAY, MKSEQUENCE, MKARRAYDESCRIPTOR
PUBLIC, PRIVATE	MKDECL, MKRECORD
READONLY	MKVAR, MKREF, MKLIST, MKPOINTER, MKARRAYDESCR, MKDECL (interface vars only)
SAFE	MKXFERTYPE
UNSAFE	MKXFERTYPE

Table 4 4: Type options and their constructors

4.2.3 Primitive procs

The primitive procs and other values of Cedar are listed in Table 4 5. All of the procs language except the type constructors (see Table 4 3) appear here.

The Name column gives the name of the value in the cluster. The Classes column gives the in which the [name, value] pair appears; the primitive classes of Cedar are summarized i 4 1. The Type column gives the type with which it is declared in those classes. The type to other names of the class; see the detailed class descriptions in ¶ 4.3-4.10 for more

The Notes column gives information about how a proc is applied or a non-proc value is de current Cedar. In the kernel a proc named P from the cluster of type T is applied to a va type T by the expression x.P (if there is only one argument) or x.P[y, ...] if there are s current Cedar, however, very few primitives can be applied or denoted by dot notation. I there are three ways of applying a primitive proc:

> It may be an operator with a symbol listed in the Notes column. If it takes two ar the operator is infix. Thus for a proc named P with operator symbol 4, you write x4 instead of x.P[y]. If it takes one argument the operator is usually prefix; you wri instead of x.P. The ^ operator is postfix; you write x^ instead of x.DEREFERENCE.
>
> It may be a built-in proc named P, in which case you usually write P[x] or P[x, y, of x.P or x.P[y, ...]. Other ways of applying a built-in are indicated in the Notes
>
> It may be a funny application proc named P, in which case you write P x or P x[y, . instead of x.P or x.P[y, ...].

The three kinds of primitive proc are listed in that order, and alphabetically within ea Primitive values which are not procs (ABORTED, FALSE, FIRST, LAST, NIL, SIZE, TRUE) are listed the built-in procs, and the syntax for them is given explicitly.

A few primitive procs cannot be desugared so simply into dot notation. These cases are i the Notes column, and are described here:

> ABORTED, FALSE, NIL and TRUE are globally known names.
>
> Some PROC [T]_[U] are coercions: CONS, FROMGROUND, LONG, TOGROUND, VALUEOF. This means that they may be invoked automatically when typechecking demands a U and an expression has syntactic type T; see ¶ 4.13 for details.
>
> Some involve target typing: ALL, CONS, LIST, LOOPHOLE, NARROW; they are marked TT. For these the proc does not come from the cluster of the type of the first argumen Instead, it comes from the cluster of the so-called target type. An application of

these procs must appear as an argument in another application (e.g., f [y, NARROW[x]
Z_NARROW[x]), and not before a dot. In this context the target type is known from th
declaration of the outer proc being applied (f or Z.ASSIGN in the example; if its t
PROC [U, T]_[V], the target type for the NARROW application is T). Target typing is a
used for enumeration literals (¶ 4.7.1.1).

One is ambiguous: MINUS for CHAR and pointer. The choice of proc depends on the type
the second arguement.

Name	Notes	Classes	Type
Operators (infix except as noted)			
VARTO-POINTER	@(prefix)	general	UNSAFE PROC[T]_[MKPOINTER[range~T.RANGE, long~LONG]
EQUAL	=	general	PROC[x: T, y: T]_[BOOL]
ASSIGN	_	assignable	PROC[x: VAR T, y: T]_[T]
PLUS	+	numeric	PROC[T, T]_[T]
	+	•CHAR, •pointer	PROC[T, INTEGER]_[T]
MINUS		numeric	PROC[T, T]_[T]
		•CHAR, •pointer	PROC[T, T]_[INTEGER]
	ambiguous		PROC[T, INTEGER]_[T]
UMINUS	(prefix)	numeric	PROC[T]_[T]
TIMES	*	numeric	PROC[T, T]_[T]
DIVIDE	/	numeric	PROC[T, T]_[T]
LESS	<	ordered	PROC[T, T]_[BOOL]
GREATER	>	ordered	PROC[T, T]_[BOOL]
	~	same as NOT	
DEREF-ERENCE	^(postfix)	reference	PROC[r: T]_[RANGE]
IN	IN	ordered	PROC[T, SUBRANGE]_[BOOL]
REM	MOD	whole number	PROC[T, T]_[T]
NOT	NOT(prefix)	BOOL	PROC[BOOL]_[BOOL]
Built-in procs (applied with P[x] instead of x.P, except as noted)			
ABORTED	ABORTED	SIGNAL	SIGNAL
ABS		numeric	
ALL	TT	ARRAY	PROC[x: RANGE]_[T]
BASE		row	PROC[a: VAR T]_[LONG POINTER TO UNSPECIFIED]
		descriptor	PROC[a: T]_[LONG POINTER TO UNSPECIFIED]
CODE		TYPE	PROC [T: TYPE]_[RTT.Type]
CONS	T[...]; coercion	ARRAY	PROC[g: RANGE X ...]_[T]
	T[...]; coercion	RECORD	PROC[b: FIELDS]_[T]
	a[...]; TT for a	variant	PROC[b: FIELDS]_[T[a]]
	z.CONS[...]; TT	LIST	PROC[z: ZONE_SafeStorage.GetSystemZone, x: VALUE, y: T]_[
DESCRIPTOR		row, variant record	PROC[r: VAR T]_ [LONG DESCRIPTOR FOR ARRAY T.DOMAIN of T.RANGE]
	xxx	descriptor	PROC[base: LONG POINTER TO UNSPECIFIED, length: CARDINAL, t: TYPE]_[LONG DESCRIPTOR FOR ARRAY CARDINAL OF t]
FALSE	FALSE	BOOL	T
FIRST	FIRST[T]	discrete	T
first	l.first	LIST	PROC[l: T]_[VAR VALUE]
FREE	z.FREE[..]	ZONE	UNSAFE PROC[z: T, p: NEWTYPE[NEWTYPE[u]]]_[]
FROMGROUND	T[...]; coercion	subrange	PROC[x: GROUNDTYPE]_[T]
ISTYPE		general	PROC[x: T, U: TYPE]_[BOOL]
LAST	LAST[T]	discrete	T
LENGTH		ARRAY, descriptor	PROC[a: T]_[CARDINAL]
LIST	z.LIST[...]; TT	LIST	PROC[z: ZONE g: VALUE X ...]_[T]
LONG	coercion	short number	PROC[p:T]_[LONG T]
	coercion	POINTER	PROC[p:T]_[LONG POINTER TO T.RANGE]
	coercion	DESCRIPTOR	PROC[p:T]_[LONG DESCRIPTOR FOR ARRAY OF T.RANGE]
LOOPHOLE	TT for U	general	UNSAFE PROC[tx T, U: TYPE]_[U]
MAX		ordered	PROC[T, ...]_[T]
MIN		ordered	PROC[T, ...]_[T]
NARROW	TT for U	general	PROC[x: T, U: TYPE]_[U]
NEW	z.NEW[..]	ZONE	PROC[z: T, U: TYPE]_[r: NEWTYPE[U]]
NIL	NIL[T] or NIL	variable, address, transfer	T or NILTYPE

PRED		discrete	PROC[x: T]_[T]
rest	l.rest	LIST	PROC[T]_[T]
SIZE	SIZE[T]	general	CARDINAL
SUCC		discrete	PROC[x: T]_[T]
TOGROUND	coercion	subrange	PROC[x: T]_[GROUNDTYPE]
Funny applications			
BROADCAST	no args	CONDITION	PROC[T]_[]
ERROR		SIGNAL, ERROR	like APPLY
FORK	FORK P[args]	PROC	PROC[PROC[DOMAIN]_[RANGE]]_ [PROC[DOMAIN]_[PROCESS []_[RANGE]]]
JOIN	no args	PROCESS	PROC[T]_[RANGE]
NEW	no args	PROGRAM	PROC[p: T]_[T]
NOTIFY	no args	CONDITION	PROC[T]_[]
RESTART	no args	PROGRAM	PROC[T]_[]
RETURN WITH		PrincOps.StateVector	
RETURN WITH ERROR		ERROR	???
SIGNAL		SIGNAL, ERROR	like APPLY
START		PROGRAM	like APPLY
STOP	no args	PROGRAM	PROC[]_[]
TRANSFER WITH		PrincOps.StateVector	
WAIT	no args	CONDITION	PROC[T]_[]
Not in current Cedar			
APPLY		map	PROC[map: T, arg: DOMAIN]_[RANGE]
BINDING			
DEREF			
DOMAIN		map	TYPE
HIDEEXCEPTION			
LOCALSTRING		STRING	PROC[[CARDINAL]]_[STRING]
NAMES		binding, decl	
NEW		general	PROC[z: ZONE_SafeStorage.GetSystemZone]_[r: REF T]
NEWEXCEPTIONCODE		exception	
OPENPROCS			
RANGE		map	TYPE
TOVOID		assignable	PROC [T]_[VOID]
UNCONS	coercion	record	PROC[T]_[FIELDS]

Table 4 5: Primitive procs

4.3 General and assignable types

The classes described in this section include nearly all (general) or most (assignable) types. In other words, nearly all types have equality, and most have assignment.

4.3.1 General types

The general class is

```
T: TYPE                                  -- The type itself.
SIZE: CARDINAL                           -- The number of words to represent a T
                                            value; denoted SIZE[T].
ISTYPE: PROC[x: T, U: TYPE]_[BOOL]       -- Roughly, TRUE if x has type U.
NARROW: PROC[x: T, U: TYPE]_[U]          -- Converts x into a U if possible, or raises
                                            the error Runtime.NarrowFault.
```

```
,LOOPHOLE: UNSAFE PROC[x: T, U: TYPE]_[U]  -- Returns the bits representing x as a U.
INIT: PROC[STORAGEBLOCK[SIZE]]_[VAR T]     -- Can't be called directly.
NEW: PROC[z: ZONE_SafeStorage.GetSystemZone  -- Denoted NEW[T] or z.NEW[T].
                    T: TYPE]_[r: REF T]
PREDICATE: PROC[x: T]_[PROCANY_BOOL]       -- The predicate of the type.
CLUSTER: PROC[x: T]_[BINDING]              -- The cluster of the type.
```

All types are in this class except TYPE and fully opaque types.

Anomaly: For NARROW and LOOPHOLE the second argument may be defaulted to the target type.

In current Cedar the value of ISTYPE[x, T] is determined as follows:

1) It is TRUE statically if:

   Dx and T have the same predicate, or

   one of Dx and T is an opaque type, and the other is the corresponding concre type (only in an implementation that exports the opaque type).

2) It is tested dynamically if (with V any variant record type without a COMPUTED tag, the name of a particular variant):

   Dx is REF ANY and T is REF U for any U except ANY, or

   Dx and V have the same predicate, and T is equal to V[a], or

   Dx and REF V have the same predicate, and T is equal to REF V[a], or

   Dx is equal to (LONG) POINTER TO V and T is equal to (LONG) POINTER TO V[a].

   Note that the result is TRUE if x=NIL.

3) It causes a static error in all other cases, even if it is statically false.

In current Cedar, NARROW[x, T] is

```
IF ISTYPE[x, T] AND (x~=NIL OR ISTYPE[x, REF ANY]) THEN x ELSE ERROR e
```

where e is

```
RTTypesBasic.NarrowRefFault[x, CODE[T.RANGE]] if ISTYPE[x, REF ANY];
RTTypesBasic.NarrowFault[] otherwise.
```

Note that NARROW[x, T] gives a static error if ISTYPE[x, T] does (case (3) above), and that x=NIL unless Dx is REF ANY.

Two types may be unequal and yet have the same predicate if they have different clusters Currently, the cluster can only be changed by CHANGEDEFAULT.

The INIT proc converts a block of storage into a legal variable of type T. It is currentl except for

RC types (¶ 4.5); these are set to NIL.

Bound variants; the tag field is set appropriately.

INIT cannot be supplied by the user and can only be called indirectly from NEW.

The NEW proc calls on the zone z to obtain a block of storage of size T.SIZE (¶ 4.5.2), ar T.INIT to convert the block into a VAR T; call it x. Then if T.DEFAULT exists, NEW calls it the result to x.

CHANGEDEFAULT can take any type and produce a new one which is identical except for the DEFAULT which determines how default values are supplied when a binding is coerced to a declaration; see ¶ 4.11 for details.

```
DEFAULT: PROC[]_[T]
```

Every general type has an EQUAL proc except a variant record or union type. A variant rec
has EQUAL only if its variant part is a union in which all the cases are the same size. N
bound variant does have EQUAL, unless it is itself a variant record. EQUAL is denoted by t
operator =.

```
EQUAL: PROC[x: T, y: T]_[BOOL]          -- TRUE iff the bits representing x and y
                                           are identical.
```

Anomaly: If v is a variant record and bvis one of its bound variants, the expression v=bv
EQUAL proc of the bound variant.

Representation: Addresses in the representation of a value are compared, not dereference
the comparison. Thus types like STRING and ZONE which are represented by addresses are
compared by comparing the addresses.

4.3.2 Assignable types

Most types (see Table 4 1 for exceptions) have a proc (denoted by the right-associative
operator _):

```
ASSIGN: PROC[x: VAR T, y: T]_[T]        -- Returns y after storing it in x.
```

As explained in ¶ 3.7, groups and bindings with variable elements are assignable. Since
write these types in declarations, you have to write the constructors explicitly on the
they are called extractors.

Representation: Since it involves a VAR, such a proc cannot be written in current Cedar.
primitive ASSIGN procs simply copy the bits of y's representation into the variable x, un
them represent REFS. In this case the assignment involves reference-counting if x is in c
storage; see ¶ ??? for details.

4.3.3 Variable types

For every non-variable type T there are corresponding variable types:

```
VAR T

READONLY T

SHORT VAR T

SHORT READONLY T
```

You cannot denote these types in current Cedar, but they are fundamental to an understan
how it works nonetheless. The basic facts about variables in Cedar are given in ¶ 2.2.3.

The var class has names:

```
RANGE: TYPE;                            -- (VAR T).RANGE=T
LONG: BOOLEAN;                          -- FALSE for short vars
READONLY: BOOLEAN;                      -- TRUE for readonly vars
VARTOPOINTER: UNSAFE PROC[T]_           -- Apply by prefix @
            [MKPOINTER[range~T.RANGE, long~LONG]];
VALUEOF: PROC[T]_[T.RANGE];             -- A coercion.
```

Furthermore, T inherits the cluster of T.RANGE. The procs are not modified, since the VALU
coercion provides them with T.RANGE arguments where needed.

The VAR constructor should be in the cluster.

4.3.4 Opaque types

Incomplete. Currently treated in ¶ 3.3.4.1

4.4 Map types

The map class is

```
  DOMAIN: TYPE;                              -- Domain type for the mapping.
  RANGE: TYPE;                               -- Range type for the mapping.
  APPLY: PROC[map: T, arg: DOMAIN]_[RANGE]-- map[arg] is sugar for map.APPLYZarg.
```

Usually DOMAIN and RANGE are declarations, so that bindings can be used for the arguments a
results. Application is denoted by brackets: map[arg]; for transfer types the syntactic f
¶ ???) can also be used.

There are several subclasses of map in Cedar, each with its own APPLY proc. These are sum
here, and treated in detail in the sections on the various subclasses.

Primitives (since you can't get hold of the value of the primitive, these can be a
with the various special syntactic forms summarized in Table 4 5).

Transfer types: procs, and their close friends signals, errors, ports and programs
transfer type executes the body of some l-expression (¶ 4.4.1). ¶ 2.2.1 and ¶ 2.6
applying procs.

Row and descriptor types: applying an array, sequence (or sequence-containing reco
array descriptor to an index value yields a VAR of the component type (¶ 4.4.2).

BASE POINTER types: applying a base pointer to a value which is relative to that bas
a (non-relative) pointer; this is unsafe (¶ 4.4.3).

Reference types: if the base type T has APPLY, then the reference type inherits it
with DEREFERENCE, so that a[arg] is the same as a^[arg] (¶ 4.5.1).

TYPE: Many subclasses of TYPE have APPLY procs with assorted meanings (¶ 4.8).

4.4.1 Transfer types

The subclasses of transfer are PROC, PORT, PROGRAM, PROCESS, SIGNAL, and ERROR. These types
are constructed by transfer type constructors which begin with those words, or in the ke
MKTRANSFERTYPE constructor. What they have in common is that application executes the body
some l-expression, but the transfer class adds no names to the map class.

One transfer type T implies another U if

The subclass is the same (or if T is a SIGNAL and U is an ERROR).

T.RANGE implies U.RANGE.

U.DOMAIN implies T.DOMAIN.

See ¶ 2.3.2 and ¶ 4.12. One declaration D implies another E if:

They have the same names, or each has only one name, and

The corresponding types imply each other. I.e.

If n: T is in D and n: U is in E, then TgU.

If D=[m: T] and E=[n: U], then TgU.

D implies a cross type T if D.STRIPNAMES implies T; in this case T also implies D.

Representations for transfer types are given in the PrincOps interface.

PROC types

The PROC class has no additional names. In the kernel, a new proc value is made by evalua
expression. In current Cedar, it is made by a binding of the form

```
P: T~{. . .}
```

in a block, where T is a proc type; see ¶ 3.5.1 for details.

PORT types

Incomplete.

PROGRAM types

The syntax for applying a program P is

```
START P[args]
```

The START may be omitted, so that it looks like an ordinary application. This expression'
DP.RANGE. The program class also has procs

```
STOP: PROC[]_[]                    -- Apply by STOP. Legal only if RANGE=[].
RESTART: PROC[T]_[]                -- Apply by RESTART P. Legal only if RANGE=[].
NEW: PROC[p: T]_[T]                -- Apply by NEW P; makes a copy of p's
                                      implementation.
```

Their use is not recommended; for details, consult a wizard. For more on implementations
???.

PROCESS types

A process always has DOMAIN=[]. The syntax for applying a process P is

```
JOIN P
```

The JOIN may be omitted, so that it looks like an ordinary application. This expression's
DP.RANGE.

The only way to make a new process is with

```
FORK: PROC[PROC[DOMAIN]_[RANGE]]_[PROC[DOMAIN]_[PROCESS []_[RANGE]]]
```

The syntax for using this is

```
FORK P[args].
```

The FORK P returns a proc when when applied to args creates a new process, starts it runni
returns it. You cannot write FORK P alone to get hold of the process-creating proc.

Anomaly: Note the peculiar parsing (FORK P)[args]. Of course, you can't get your hands on
(FORK P).

There are no other names in the process class, but Process.Abort[P] raises the ERROR ABORTED
CONC describes Cedar's facilities for concurrent programming.

SIGNAL and ERROR types

The syntax for applying a error (signal) E is ERROR (SIGNAL) E[args], or ERROR (SIGNAL) E if
are no arguments. For a signal, this expression's type is DE.RANGE; for an error, its typ
(since control can never return). •If there are arguments, the ERROR or SIGNAL is optional
this feature.

In the kernel, a new signal or error value is made by applying NEWEXCEPTIONVALUE. In current Cedar, it is made by a binding of the form

```
E: T~CODE
```

in a block, where T is a signal or error type; see ¶ 3.5.2 for more about this. ¶ 2.5.2 explain errors in detail. A signal is exactly like a proc, except that the closure that obtained from the statement of an enable choice; see ¶ 3.4.2 for details.

You can write an expression consisting simply of ERROR; this is short for ERROR NAMELESSERROR Here NAMELESSERROR is an error you cannot denote in the program. Hence it cannot be caught (except by ANY).

4.4.2 Row and descriptor types

A row value provides an indexed set of values of an arbitrary type, called the component row; application maps an index into the corresponding value. Usually the values are vari that assignment to a component is possible. Descriptors are unsafe pointers to arrays wh a subrange of the domain of index type in the descriptor value; thus the same descriptor point to arrays of different size. Because all the row types use the same representation values, it is possible to make a descriptor from any row value.

The domain or index type of a row must be a discrete type with no more than 2^{16} distinct note that this rules out large subranges of INT. There is one element in the range set fo of the domain type.

The PACKED argument of the row type constructors governs the representation of a row whos range type is represented in <8 bits. See the discussion of representation below. It als the use of @ on an element of the row.

The row class has the proc:

```
DESCRIPTOR: PROC[r: VAR T]_[            -- Returns a descriptor for r.
  LONG DESCRIPTOR FOR ARRAY DOMAIN of RANGE]
```

Since DESCRIPTOR returns an address, it must take a VAR; i.e., it can’t be given a row valu a constructor, but demands a row which has been declared or allocated.

Anomaly: DESCRIPTOR returns a LONG DESCRIPTOR unless r is a short VAR, i.e., a declared variab or the result of dereferencing a short pointer.

Representation: A VAR row value is represented by a contiguous block of words. If PACKED=F each element VAR occupies SIZE[T.RANGE] words, and the successive elements occupy consecuti blocks of storage, beginning with the one indexed by FIRST[T.DOMAIN]. If PACKED=TRUE and a T.RANGE value is represented in n<8 bits, each element occupies $2^{\text{CEILING}[\log_2[n]]}$ bits, i.e. 1, bits depending on its size; PACKED has no effect on the representation for ranges with bi values. Note that the entire representation of a packed array may be smaller than a word not be word-aligned in another packed array or in a record. This is the entire represent array value; a sequence value also has a tag field, which is represented like a componen containing record.

It is not possible to obtain a REF to a row element; this is because the implementation c reference counting and REF ANY discrimination requires more information about each VAR tha available for an array element. If the row is PACKED, it is not possible to obtain a poin element either (using @.

Performance: Passing a row as an argument entails copying the representation. Unless the quite small, this is expensive. It is usually better to pass a REF. Very large rows (say, words) should not be declared, since this results in large frames which consume the 64k frame space. Instead, they should be allocated with NEW.

4.4.2.1 ARRAY types

An array is a row with an element for each value in the domain; its APPLY proc is a total
The advantages of this are that no space is needed to store the length of an array, and
checking on a subscript is done against constant values (as part of narrowing the subscr
domain type, which is usually a subrange). The disadvantages are that a given proc, writ
with a given array type, cannot be used on other arrays of different lengths, since ther
current Cedar to parameterize the proc with a type. In this case it is better to use a s

The array class has the procs:

```
CONS: PROC[g: RANGE X ...]_[T]          -- A coercion from the group.
ALL: PROC[x: RANGE]_[T]                 -- Returns an array with each element =x
LENGTH: PROC[a: T]_[CARDINAL]           -- Returns the cardinality of DOMAIN.
BASE: PROC[a: VAR T]_[LONG POINTER TO UNSPECIFIED] -- Returns the address of a's first element
```

CONS takes a group of values, one for each element of the array, into an a array value. T
argument of CONS may have omitted values, which are filled in if possible by the defaulti
coercion for g. If the index type is enumerated, CONS takes a binding, with one element n
type T.RANGE for each index value n. In current Cedar you can't write T.CONS. Instead you
itself; i.e., T[...] for T.CONS[...]. Because CONS is a coercion from group to array, you
whenever the group appears as an argument or in a binding; see ¶ ???. Examples:

```
I: TYPE~INT_0; B: TYPE~BOOLEAN_TRUE
A: TYPE~ARRAY [0..5) OF I;
a1: A~[0, 1, 2, 3, 4];              -- OK to omit A here.
a2: A~[  , 1, 2, 3, 4];             -- Same as a1, by defaulting.
i: INT~A[4, 3, 2, 1, 0][1];         -- i=3. The A is required here.
E: TYPE~ARRAY {red, blue, green} OF B;
e1: E~[TRUE, FALSE, TRUE];
e2: E~[blue~FALSE];                 -- Same as e1.
```

Anomaly: ALL replicates its argument in all the elements of an array. In current Cedar yo
write T.ALL. Instead you just write ALL; it must be in an argument or binding. Unlike mos
ins, ALL is not sugar for dot notation. If the range type permits it, you can write ALL[NU
all the elements.

```
a3: A~ALL[3];                       -- Same as [3, 3, 3, 3, 3]
```

BASE returns the address of its VAR array argument. It is mostly useful for writing storag
The resulting LONG POINTER TO UNSPECIFIED can also be passed to DESCRIPTOR to yield a
descriptor for a different type of array. If the VAR is short, the result can be narrowed

Anomaly: An array may be declared with a domain type which is an empty subrange. The eff
to suppress the bounds checking in APPLY. If a pointer p to such an array is constructed
LOOPHOLE), then p^[i] (you can also write p[i], because p inherits APPLY) will never give a
BoundsFault. This kludge is sometimes useful for obtaining arrays whose size is not stati
beware that operations on the array other than subscripting (e.g., equality tests, assig
parameter passing) will believe the type declaration and do the wrong thing. It is gener
use a sequence or a descriptor.

4.4.2.2 SEQUENCE types

A sequence is like an array, but each sequence value includes a tag value which specifie
number of elements in that sequence, i.e. the values of the domain type for which APPLY i
If the domain type is T and the tag value is v, then APPLY is defined for [FIRST[T]..v). Us
NAT, so that v is the number of elements in the sequence, and the elements are indexed by
v 1.

In current Cedar there are many restrictions on the use of sequences. A sequence type is a sequenceTC[51]; it is not a first-class type, and can only appear as the type of the last variant record or union (¶ 4.6.2). Furthermore, the only items in the cluster of a seque the row APPLY and DESCRIPTOR procs; these are inherited by the containing variant record, w the type a program normally deals with.

A record type T containing a sequence field is a variant record. T is a first-class type bound to a name, but unlike a union-containing record it cannot be used where type[36] appe the grammar, except in a refTC[53] or pointerTC[55]. The only operations in the cluster of T a ones of the variant record class (¶ 4.6.2), and some inherited from the row class of the sequence:

```
DOMAIN: TYPE                              -- =TAGTYPE.
RANGE: TYPE                               -- The RANGE of the sequence.
APPLY: PROC[map: T, arg: DOMAIN]_[RANGE]-- Indexes the sequence.
 ~{RETURN[t.UNIONPART[i]]}.
DESCRIPTOR: PROC[r: VAR T]_[LONG DESCRIPTOR FOR ARRAY DOMAIN OF RANGE]
 ~{RETURN[DESCRIPTOR[t.s]]}.              -- Yields a descriptor for the sequence.
```

The tag of a sequence is readonly.

Hence the only uses of T are:

As the range type of a reference type, e.g., REF T.

In the form T[n] to yield a specialization of T.

The specialization T[n] has TAG=SUCC^n[FIRST[T.TAGTYPE]], and n elements in the sequence; n ne not be static. This application causes a Runtime.BoundsFault if n NOT IN T.TAGTYPE. T[n] is a first-class type; you cannot write it where type[36] appears in the grammar, and it has onl following cluster (¶ 4.3.1):

```
NEW                                       -- Denoted NEW[T[n]] or z.NEW[T[n]] .
SIZE                                      -- Denoted SIZE[T[n]].
```

Note that since you cannot use T or T[n] in a declaration, there are no declared variable fields, or arguments to non-primitive procs of these types; you must use REF T (or a poin Furthermore, these types have no ASSIGN or EQUAL procs; you must do these operations on the components. Finally, there are no constructors for sequence types; you must explicitly t sequence field in a record constructor. A sequence does get initialized when allocated, current Cedar this just means that basic RC variables are set to NIL.

Thus the normal way to use a sequence is to embed it in a record (which need not have an components), and to allocate one of the desired size using NEW (as in the examples below) record value can then be applied to index the sequence. Usually it is convenient to have DOMAIN=NAT. If, however, some maximum length N is important to you, consider DOMAIN=[0..N]; then the value of the tag field is a sequence of length n<N is just n, and the valid indi [0..n).

Examples:

```
StackRep: TYPE~RECORD[
  top: INT_ 1,
  item: SEQUENCE size: NAT OF T];
Number: TYPE~RECORD[
  sign: {plus, minus},
  magnitude: SELECT kind: * FROM
    short=>[val: [0..1000)],
    long=>[val: LONG CARDINAL],
    extended=>[val: SEQUENCE length: NAT OF CARDINAL]
    ENDCASE];
rs1: REF StackRep_NEW[StackRep[100]];            -- rs1.top= 1, rs1[i] is trash.
rs2: REF StackRep_NEW[StackRep[100]]_[top~3, item~nums];  -- rs2.top=3, rs2[i] is trash.
```

```
rn1: REF Number[extended]_NEW[Number[extended][2*k]];
-- ns2[2]=ns2^[2]=ns2.item[2]=ns2^.item[2], but all start out trashed.
```

A sequence may have a COMPUTED tag, with the same meaning as for unions: no tag field exis
bounds checking is possible so that application is unsafe, and the cluster has no DESCRIPT
You can still compute the address of the sequence with @ and use the unsafe three-argume
of DESCRIPTOR (¶ 4.4.2.3). A sequence may not have an OVERLAID tag, and * cannot be used for
tag type. Example:

```
-- Here is the recommended method for imposing an indexable structure of raw storage.
WordSeq: TYPE~RECORD[SEQUENCE COMPUTED CARDINAL OF Word];
```

A sequence may appear in a MACHINE DEPENDENT record. It must come last, both in the record
constructor and in the layout. The total length of a record with a zero-length sequence
a multiple of the word length. The size of the sequence field (if specified) must descri
length sequence; i.e., it must account for just the space occupied by the tag field (if

There is a predefined sequence TEXT; see Table 4 2 for its declaration. There are literal
REF TEXT, denoted as in rule 57 by the characters of the literal enclosed in doublequotes.
literal is shorthand for a constructor (which you couldn't actually write in current Ced
lacks constructors for sequences). TEXT can be used where efficiency is critical; for gen
use ROPE (¶ ???).

•There are also unsafe predefined types LONG STRING and STRING; see Table 4 2 for their
declaration. They are described here for completeness, but should not be used. These typ
pointers to a StringBody type also given in Table 4 2. In spite of the declaration, String
like a sequence with tag maxlength and sequence text. Thus z.NEW[StringBody[n]] returns a ST
LONG STRING with maxlength=n; if s is a STRING or LONG STRING, s[i] indexes its text, etc. You
also use s.text, as with sequences, but this is not recommended: because of the definitio
never bounds-checked (use s[i]), and DESCRIPTOR[s.text] describes an array of length 0 (use
DESCRIPTOR[s^].

•There is a special kludge for allocating a string in the local frame of a proc:

```
LOCALSTRING: PROC[ [CARDINAL] ]_[STRING] -- A coercion.
```

Because this is a coercion, you can write

```
s: STRING~[20]
```

to obtain a local string of length 20. Of course, the storage will be freed when the pro
freed, and a dangling reference may remain.

•There are literals of type STRING, denoted just like REF TEXT literals as in rule 57. Sinc
string literals, they are allocated in the MDS, where they consume precious space. By su
the literal, you can get it allocated in the proc frame, where the space is recovered wh
is freed, at the risk of a dangling reference.

4.4.2.3 Descriptor types

A descriptor is a pointer to a row value which includes a subrange of the row's domain a
the descriptor value. A proc which takes descriptors rather than rows or REFs to rows can
rows of different sizes. Because a descriptor is like a pointer, there are short, long a
descriptors which are exactly analogous to short, long and relative pointers; see ¶ 4.5.
for details.

Like a row, a descriptor can be applied to yield a VAR of the range type. If it is READONL
will be READONLY too.

Like array, descriptor has the procs:

```
LENGTH: PROC[a: T]_[CARDINAL]          -- Returns the cardinality of the subrange in a.
```

```
BASE: PROC[a: T]_[POINTER TO UNSPECIFIED] -- Returns the address of a's first element.
```

There is also an unsafe proc for making a descriptor with RANGE=CARDINAL from a LONG POINTER:

```
DESCRIPTOR: PROC[base: LONG POINTER TO UNSPECIFIED, length: CARDINAL, type: TYPE]
                                _[d: LONG DESCRIPTOR FOR ARRAY CARDINAL OF type]
```

LENGTH[d]=length and BASE[d]=base. There is a similar proc with both LONGs dropped.

Anomaly: The type argument of DESCRIPTOR may be omitted, in which case it is the range type
the target type (which must be a descriptor type). Similarly if the target type is packe

•4.4.3 BASE POINTER types

A base pointer bp is like an ordinary pointer, except that it has an APPLY operation which
relative pointer rp (see ¶ 4.5.1) into an ordinary pointer p:

```
APPLY: UNSAFE PROC[bp: T, rp: DOMAIN]_[p: rp.RANGE]
DOMAIN: (T RELATIVE POINTER).TYPE;
```

Note that the type of p is determined by the type of rp, and has nothing to do with the t
There can be many relative pointer types for a single base pointer type. The scheme is m
safe than ordinary pointers, since a particular relative pointer in general makes sense
a particular base value, but the type system allows it to be used with any base value of
base type.

In other respects, a base pointer is like an ordinary pointer; indeed, it is a subclass
it has a range type of its own, and can be dereferenced to yield a value of that type. T
point to a record or other variable at the start of the region. Confusingly, the base po
has nothing to do with the range of its APPLY.

A base pointer type implies the corresponding non-base type, and vice versa.

Representation: The APPLY proc is

```
    l [bp: T, rp: DOMAIN] IN
        LOOPHOLE[LOOPHOLE[bp, LONG CARDINAL]+LOOPHOLE[rp, LONG CARDINAL],
                LONG POINTER TO rp.RANGE]
```

if T.LONG=TRUE or DOMAIN.LONG=TRUE, or the same thing without the LONGs if neither is long.

4.5 Address types

An address value is the address of a variable, i.e., of a block of storage. These values
many different ways. They have only one thing in common:

```
NIL: T                                  -- A distinguished value pointing to no storage.
```

In current Cedar you cannot write T.NIL; instead, you write NIL[T]. There is a universal v
which can be coerced into any particular NIL.

Storage is a precious resource which must be reclaimed when it is no longer needed, i.e.
variable it represents will no longer be touched by the program. A conservative definiti
variable is no longer reachable. If new address values are generated only by a NEW proc w
never generates the address of a reachable variable, then a variable is reachable if

> an address value for it is stored in some other reachable variable, or
>
> it is the process array or, in current Cedar, the global frame of a module.

REF and transfer values are intended to be such addresses; collectively they are called c
values:

- Within the safe language, a new counted value can be made only by such a NEW proc,
  by evaluating a l-expression.

NEW returns the address of a block of at least SIZE[T] words, none of which is of an already reachable variable.

Evaluating a l-expression L returns a closure (¶ 2.2.1), which includes the of the frame in which L was evaluated. This frame provides the proper environment for binding free variables in L. A frame in turn is made only by NEW proc for that frame type.

- A counted value always addresses counted storage, which is made only by such a NEW It is called counted because of the implementation of the test for reachability; s
- The storage representing a counted variable is reclaimed (by the garbage collector when it is no longer reachable.

These three invariants ensure that within the safe language a REF T always addresses a VA does not intersect any other variable. The second invariant is guaranteed entirely by th (given the other two invariants).

Incomplete

Furthermore, a VAR REF T v must never be equal to a VAR U for any U=REF T. Otherwise a U could be assigned to the VAR, incorrectly making a new REF T value which can be retrieved v.VALUEOF.

The third invariant requires an implementation which can compute reachability. Cedar has

One does reference counting, and hence must know about every assignment to a REF.

The other scans all the reachable variables, starting with the process array and f

Both need to be able to find all the REFs in a variable.

Thus to maintain the safety invariants, it is necessary to ensure that no other value is a value containing a REF. Such a value is called REF-containing, or RC for short. The fol classes have RC values:

REF (which includes LIST and ATOM)
record, union or array with a RC component.
transfer (not counted deficiency).

To define the safety invariants for non-reference addresses, it is necessary to define a which can contain such addresses. Such values are called pointer-containing, or PC for s following classes have PC values:

pointer (which includes POINTER TO FRAME and string);
descriptor;
record, union or array with a PC component.

Inconplete. Notes:

AC types (6T5/12):
pointers to RC: dangerous but allowed.
type encoding: none, prefix, quantum is in zone.

4.5.1 Reference types

This class has the following names:

```
RANGE: VAR.TYPE
DEREFERENCE: PROC[r: T]_[T.RANGE]        -- Denoted by r^
APPLY: PROC[r: T, arg: T.RANGE.DOMAIN]_ -- Inherited from the range type.
                        [T.RANGE.RANGE]
f: PROC[r: T, arg: T.RANGE.f.DOMAIN]_   -- Inherited from the range type.
                        [T.RANGE.f.RANGE]
```

The range of a reference type T may be any VAR type VAR U. If T is READONLY, then T.RANGE is READONLY also; this means that assignment to the dereferenced address is impossible. Dereferencing a T yields a VAR U (which can then be coerced to a U value if appropriate). Dereferencing NIL causes the error Runtime.PointerFault.

If the range has an APPLY, WAIT, NOTIFY or BROADCAST proc, or any record field procs in its c these are inherited by the reference type (except that APPLY is not inherited by a BASE PO which has its own APPLY; see ¶ 4.4.3). The value of an inherited f is

```
l [r: T, arg: T.RANGE.f.DOMAIN] IN r^.f [arg]
```

In other words, the address is dereferenced, and then the range's f is applied. The effe reference to an array or proc can be applied without explicit dereferencing, a reference condition can be used to do a WAIT or whatever, and a reference to a record can be used t field. This does not work for procs which get into a cluster by being in an interface i

4.5.1.1 REF types

A REF value can be created only by a NEW proc. Every general type except union has one of (¶ 4.3.1). There are no additional names in the REF class.

The type VAR ANY may be the range of a REF; it cannot appear anywhere else except as the d or range of a proc type. This REF type is denoted REF ANY. It is implied by every REF type can be used to test the particular REF type of a REF ANY value, and NARROW can be used to convert a REF ANY value into a REF T value (¶ 4.3.1). These two operations are combined in convenient way by the WITH ... SELECT construction (¶ ???). REF ANY does not have a DEREFEREN proc, and of course there are no procs for it to inherit from the range.

LIST types

The LIST class has names:

```
VALUE: TYPE;
first: PROC[l: T]_[VAR VALUE];          -- Denoted l.first, not first[l]
rest: PROC[l: T]_[T];                   -- Denoted l.rest, not rest[l]
CONS: PROC[z: ZONE_SafeStorage.GetSystemZone,   Denoted z.CONS[x, y] or CONS[x, y].
           x: VALUE, y: T]_[T]
LIST: PROC[z: ZONE_SafeStorage.GetSystemZone,   Denoted z.LIST g or LIST g.
           g: VALUE X ...]_[T]
```

The RANGE type R of a list type T is opaque, but it may be thought of as an unpainted rec VALUE, rest: T]; thus a list value is a REF to an R. The first and rest procs return the fi

CONS is NEW[R_[x, y]]; the optional zone tells where to do the NEW. LIST does a series of CO yielding a list such that

$$\text{LIST}[x_0, \ldots, x_n].\text{rest}^i.\text{first} = x_i$$

The g argument of LIST may have omitted values, which are filled in if possible by the de coercion for g. Examples:

```
L: TYPE~LIST OF INT_0;
l: L~LIST[0, 1, 2, 3, 4];
m: L~LIST[  , 1, 2, 3, 4];          -- Same as l, by defaulting.
```

The type ATOM

An ATOM is a REF to an opaque type which is exported from AtomsPrivate as

```
AtomRec: TYPE~RECORD[
  printName: Rope.ROPE,
  propertyList: REF ANY_NIL,
```

```
    link: ATOM_NIL]
```

There are no additional names in ATOM's cluster; the useful operations on ATOMS are provid
the ListsAndAtoms interface. However, the language does provide ATOM literals for atoms wh
Cedar names as their printnames, with the syntax $ n. Examples:

```
    $red
    $VeryLongAtomMadeUpOfSeveralWords
```

Note that you cannot put any spaces in an ATOM literal.

4.5.1.2 ,Pointer types

There are two flavors of pointer: short and long. Short pointers occupy one word, and po
within the 64k word main data space where frames are allocated. Long pointers occupy two
and point anywhere.

Pointer dereferencing is unsafe; hence all the inherited procs are also unsafe. Derefere
pointer may cause an address fault if it points to storage which is not mapped by the op
system; this is about the least disastrous thing that can happen if an unsuitable value
pointer.

Long pointer types have the following dubious names:

```
  •PLUS: PROC[T, LONG INTEGER]_[T]        -- Denoted by infix +.
  •MINUS: PROC[T, LONG INTEGER]_[T]       -- Denoted by infix  .
  •DIFF: PROC[T, T]_[LONG INTEGER]        -- Also denoted by infix  .
```

Anomaly: The infix " " cannot be desugared into dot notation, since there are two procs
by an infix " " whose first argument is a pointer. The choice between MINUS and DIFF is ba
the type of the second argument.

Short pointer types have the same procs without the LONG. They also have the following co
called lengthening:

```
  LONG: PROC[p:T]_[LONG POINTER TO RANGE]  -- Apply by LONG[p]
```

Note that VAR types have a VARTOPOINTER proc (denoted by prefix @) ; this turns a long VAR T
into a LONG POINTER TO T or a short VAR T into a POINTER TO T (¶ 4.5.4).

4.5.2 Zone types

The zone class has the names:

```
  NEWTYPE: PROC[U: TYPE]_[A: REFERENCE.TYPE];
  NEW: PROC[z: T, U: TYPE]_[r: NEWTYPE[U]];
  FREE: PROC[z: T, p: VAR NEWTYPE[u]]_[];-- For a ZONE.
  FREE: UNSAFE PROC[z: T,                -- For an uncounted zone.
                                    p: NEWTYPE[ NEWTYPE[u]]]_[];
```

Currently there are exactly three zone types:

ZONE, with NEWTYPE=l [U: TYPE] IN MKREF[range~U];

UNCOUNTED ZONE, with NEWTYPE=l [U: TYPE] IN MKPOINTER[range~U, long~TRUE];

MDSZone, with NEWTYPE=l [U: TYPE] IN MKPOINTER[range~U, long~FALSE].

In other words, a ZONE deals in REFS, an UNCOUNTED ZONE in LONG POINTERS, and an MDSZone
in POINTERS. The latter two are called uncounted zone types.

NEW is explained in ¶ 4.3.1. FREE takes a variable (or pointer to a variable, for an uncou
containing a reference r to a variable fv. The reference r must be supplied by the NEW pr
same zone; this is checked for a ZONE. FREE sets v (or v^) to NIL. In addition:

For a ZONE, FREE sets all the REF variables of fv to NIL; this helps to break circula structures, but only the collector actually reclaims storage. Hence FREE for a ZONE

For an uncounted zone, FREE reclaims the storage for fv by calling the Dealloc proc zone (see below); hence FREE is unsafe for an uncounted zone; the safety invariant demands that FREE not be called with a pointer unless the variable will not be used more. It is best if no other pointers to fv exist.

New zones can be obtained, and other aspects of storage allocation monitored and control the procs in SafeStorage (for ZONES) or UnsafeStorage (for uncounted zones). It is also pos though not recommended, to make up your own UNCOUNTED ZONE using a type like this:

```
UncountedZoneRep: TYPE~LONG POINTER TO MACHINE DEPENDENT RECORD [
  procs (0: 0..31): LONG POINTER TO MACHINE DEPENDENT RECORD [
    Alloc (0): PROC[zone: UncountedZoneRep, size: CARDINAL]_[LONG POINTER],
    Dealloc (1): PROC[zone: UncountedZoneRep, object: LONG POINTER]
    -- possibly followed by other fields-- ],
  data (2: 0..31): LONG POINTER       -- Optional; see below
  -- possibly followed by other fields-- ];
```

The same structure serves for a MDSZone, with all the LONGS dropped and the field position adjusted accordingly. You must use a LOOPHOLE to turn one of these Rep values into an unco zone value.

If z is an uncounted zone, the code generated for z.NEW[T] is

```
z^.procs^.Alloc[z, SIZE[T]]
```

and the code generated for z.FREE[p] is

```
{temp: LONG POINTER~p^; p^_NIL; z^.procs^.Dealloc[z, temp] }
```

Usually p is @q, for some variable q which holds the pointer being freed.

Within this framework, you may design a representation of zone objects appropriate for y storage manager. In general, you should greate an instance of a UncountedZoneRep for each instance. The procs record can be shared by all zones with the same implementation; the d pointer normally references the state information for a particular zone.

4.5.3 POINTER TO FRAME types

Incomplete. Notes:

POINTER TO FRAME: Construct with implementation (and NEW?)
can put impl p in DIRECTORY as well as interf
importing pi: p gives pi the same type as PTF[p]
there is a coercion from pi to the PROGRAM type for the impl
PTR TO FRAME by NEW on PROG or PTR TO FRAME, or by import.

4.5.4 RELATIVE types

Sometimes it is convenient to have addresses which are relative to the base of some regi pointers can be shorter than ordinary pointers. Also, the entire collection of variables can be moved in storage simply by changing the base; in fact, it can be written out and in to a possibly different place, and any relative pointers stored in it will still be v provides some (unsafe) support for this facility, in the form of RELATIVE types. A RELATIVE has a range type which is an ordinary pointer or descriptor. The RELATIVE type has no DEREFERENCE or APPLY proc. The only useful thing to do with a RELATIVE value is to apply a suitable BASE POINTER to it (¶ 4.4.4).

To indicate the desired size of a RELATIVE POINTER value, the type constructor can specify subrange of CARDINAL. There are coercions between RELATIVE POINTER types which differ only i their subranges; these are just like the coercions between subranges of CARDINAL (¶ 4.7.3)

This class has names:

```
BASE: TYPE;                          -- The type of the base pointer.
SUBRANGE: SUBRANGE.TYPE              -- The subrange type; only for pointers.
RANGE: TYPE;                         -- If b: BASE and rp: T then b[rp] has type RANGE.
```

4.6 Record and union types

Record types are Cedar's facility for grouping values of different types (since group an types cannot be denoted). Unions are closely related to records because they must be emb within records in current Cedar.

4.6.1 Record types

The MKRECORD type constructor takes one argument called fields: a declaration or cross typ is a cross type, it is rebound to a decl with secret names. If fields=[n_1: T_1, n_2: T_2, ... MKRECORD produces a type with the cluster

```
n_i: PROC[T]_T_i                     -- One for each name in the decl.
FIELDS: DECL
CONS: PROC[b: FIELDS]_[T]            -- Apply by T[b]; a coercion from the binding.
UNCONS: PROC[T]_[fields]             -- No denotation; a coercion to the binding.
```

Cross type fields are not very useful, since there is no way to name the field procs. The n_i procs are not accessible; they can only be applied with dot notation. Thus if r is a r r.n_i denotes its ith field.

A record type T with a single component or type U inherits all of U's cluster. There is a coercion from T to U. The effect is that a T value behaves just like a U value, but not v

A sequence-containing record also inherits some procs from the sequence type (¶ 4.4.2.2)

If v is a VAR U returned by a field proc, you can only apply @ to it if SIZE[U]>1, or U's representation occupied an entire word, or by accident v happens to occupy a whole word record representation.

Record types in interfaces are painted: each type produced by RECORD[...] (i.e., by MKRECOR MKMDRECORD) in an interface has a unique mark. Thus two occurrences of a record type const in an interface always produce two different types. In this respect, recordTCs are like enumerationTCs, and differ from all other type constructors. In a program module, howeve types are not painted; this is so that old values will still be useful after module repl the painting of record and enumeration types is the only way to generate unique marks, i only way that an implementation can guarantee that its types cannot be forged. In practi however, the protection afforded by opaque types (¶ 4.3.4) is usually adequate.

Representation: A record variable is represented by a contiguous block of words, in whic representing each field are contiguous and do not cross a word boundary unless they fill words, but are otherwise arranged at the discretion of the compiler. It is not possible REF to a row element; this is because the implementation of both reference counting and r discrimination requires more information about each VAR than is available for a record fi a field fills one or more words, it is not possible to obtain a pointer to the field eit this is because addresses point to words.

A MACHINE DEPENDENT RECORD type constructor can specify the exact arrangement of the fields i a record, using the syntax of rules 46-48. Examples are given with the rules. Fields mus arranged according to the following rules.

> A pos[48] (w) means that the field occupies word w, or bits 16w through 16w+15, of th record variable; (w: f..l) means that it occupies bits 16w+f through 16w+l (0<f<l i required; there is no upper bound on l). All of w, f and l must be static.

The pos must be large enough to hold a variable of the field type U: if SIZE[U]>1, completely fill at least SIZE[U] words; if SIZE[U]=1 and U is a discrete, row, or re represented in less than 16 bits, it need only be as large as the representation, cross a word boundary. Union fields are treated specially (¶ 4.6.2).

Fields may not overlap, and together they must completely fill an integral number words. The order of fields is not important.

If any field has a pos, each must have one. A machine dependent record may have no This form is not recommended. If it is used, the fields are arranged consecutively constructor must be such that that the rules about word alignment and boundary cro are not violated by this arrangement.

Note that a pos is really an explicit specification of the field proc, written in a rath special language.

4.6.2 Variant record types

There are two classes, unions (¶ 4.6.3) and sequences (¶ 4.4.2.2), whose types are not f values, but can only appear as the type of the last field of a record or union. A record field is one of these types is a variant record, and its last field is a variant field. shared by a union and a sequence type is that each is a generalization of a number of sp there is a single value called the tag which identifies the special case.

For a union, the special cases are unrelated, and the tag is a value from an enume

For a sequence, the special cases are rows of different length, and the tag is a v the row's domain.

The tag[50] is treated as a field of the containing variant record. This field is readonly. can be changed only by assigning to the entire variant part or the entire variant record or new variant is RC this is unsafe. There is no way to change the tag field of a sequen COMPUTED or OVERLAID means that there is no tag field; instead, the tag value must be suppl an expression in a withSelect[34] when it is needed for specialization. Tags of * and OVERLA only for unions, and are explained in ¶ 4.6.3.

The cluster of a variant record has the following items:
The usual procs for the record fields (including the variant field itself, and the tag inherited by the record type.

```
TAGTYPE: TYPE                          -- The type of the tag.
TAG: TAGTYPE;                          -- Another proc for the tag field.
VARIANTTYPE: TYPE                      -- The (union or sequence) type of the variant field
VARIANTPART: PROC[T]_[VARIANTTYPE]     -- Another proc for the variant field.
SPECIALIZE: PROC[n: CARDINAL]_[BT: TYPE] -- A bound variant of T; denoted T[n].
```

Specialization yields a record type called a bound variant in which the type of the vari one of the special cases of the union or sequence. The bound variant lacks SPECIALIZE, its is readonly, and its VARIANTTYPE is the special case. Note that if the special case is its sequence, the bound variant is still a variant record; otherwise it is an ordinary recor

Anomaly: A variant record type has EQUAL only if it does not have a SEQUENCE field, and for two tag values a and b, SIZE[T[a]]=SIZE[T[b]]. Even if not all sizes are equal, however, EQ allowed if one of the operands is a bound variant.

The special properties of the subclasses of variant records are given in the sections on and sequences (4.4.2.2).

4.6.3 Union types

Together with REF ANY, union types provide Cedar's facilities for associating a type T wit
which contains subtypes T_1, ..., T_n, and dynamically narrowing a value of type T into a v
proper type T_i. REF ANY is more convenient:

Any REF T is a subtype of REF ANY; no pre-planning of the subtypes is required.

REF T implies REF ANY; hence procs taking REF ANY accept any REF T without further ado

Union types, on the other hand, have performance advantages:

A union type is just a value, not constrained to be a REF. These values or their VA
embedded in records or arrays without paying for extra storage allocation or an ex
of indirection.

The subtype of a union type can be discriminated several times faster than a REF AN

Union types can therefore be recommended when performance tuning is required.

A union type is defined by a unionTC[49]; it is not a first-class type in Cedar, and can or
the type of the last field of a variant record (¶ 4.6.2) or another union. The only item
of a union type are the field procs for its fields; these are inherited by the containin
is the type a program normally deals with.

The cases of the union are given by the arms of the SELECT. The type of the tag must be a
enumeration, and each case is named by one or more literals of the enumeration. Thus Node
example has cases binary, unary and nonary, and the type of the tag could have been writte
unary, nonary}. The * which actually appears for the tag type is short for an enumTC[45] whi
all the names preceding the => symbols of the SELECT in turn. If the tag type is given ex
any enumeration values which don't appear preceding a => symbol have empty fields.

A record type T containing a union field is a variant record. T is a first-class type wh
used like any other Cedar type. The only operations in the cluster of T are the ones of
record class. The fields of the union cases are not in the cluster of the variant. Howev
of the selected case in a bound variant are in the cluster (e.g., Node[binary] has procs
the example of rule 49). The names declared in a field must not be the same as any name
in the containing record. However, the same name may be declared in more than one case o
union. NULL following => is an obsolete synonym for [].

Anomaly: A constructor for a union value has the form a[...], where a is one of the enum
literals of the tag type, and [...] is an ordinary binding for the fields of case a. The
be omitted. Thus

```
n: Node_[rator~plus, rands~binary[a~NIL, b~NIL]]
```

Anomaly: If n is the name of the variant field, and r: T, r.n is legal only as the first
In all other cases, only a constructor can denote a union.

The primitive ISTYPE can be used to distinguish the case of a variant record value x, and
can be used to obtain a value bx of the bound variant type from x; see ¶ 4.3.1. The safeS
construct is a useful and efficient combination of ISTYPE and NARROW which deals systematic
with any number of cases. The withSelect[34] construct is an unsafe version of safeSelect w
be used with any union type, and is the only alternative when the tag is COMPUTED or OVERLA
See ¶ 3.8 for discussion of these forms.

If the tag is OVERLAID, any field name that appears in exactly one case of the union has a
the cluster of the variant record. When such a proc is applied to a value x, there is no
x is the proper case of the union. Obviously this is not typesafe, and if the field is R

A union has machine-dependent fields if and only if its containing record is machine-dep
The union field must be last both in the fields and in the representation. Its pos inclu
need not be word-aligned, though its tag and each field in each case must obey the align
for record fields (¶ 4.6.1). If the union field is <16 bits in size, all cases must be t
Otherwise, all cases must be a multiple of 16 bits in size, and at least one case must e
space for the union field.

4.7 Ordered types

Ordered types can be compared, and they have subranges. The subclasses are discrete, num
pointer, and subrange. The class has names

```
LESS: PROC[T, T]_[BOOL];              -- Apply by infix <. See rules 19, 22.
GREATER: PROC[T, T]_[BOOL];           -- Apply by infix >. See rules 19, 22.
IN: PROC[T, SUBRANGE]_[BOOL];         -- Apply by infix IN. See rules 19, 22.
MAX: PROC[T_FIRST[T], ...]_[T];       -- Apply by MAX[x, y, ...].
MIN: PROC[T_LAST[T], ...]_[T];        -- Apply by MIN[x, y, ...].
```

All these procs do just what you expect. MAX and MIN accept more arguments than you can w
Pointers have these procs only if ORDERED=TRUE.

The cluster also has names:

```
SUBRANGE: CLASS;                      -- The class of subrange types of T.
MKSUBRANGE: PROC[first, last: T]_[SUBRANGE];  -- See rule 25 for denotations.
MKEMPTYSUBRANGE: PROC[first: T]_[SUBRANGE];   -- See rule 25 for denotations.
```

These are discussed in ¶ 4.7.3

4.7.1 Discrete types

The discrete types are those which have a useful bijection into an interval of the natur
whole numbers and enumerations. These are the types that can be used as domains for row
4.4.2). The class has names:

```
FIRST: T                              -- Denoted FIRST[T]
LAST: T                               -- Denoted LAST[T]
PRED: PROC[x: T]_[T]                  -- Predecessor, apply by PRED[x].
                                         May cause a bounds fault.
SUCC: PROC[x: T]_[T]                  -- Successor, apply by SUCC[x].
                                         May cause a bounds fault.
```

Whole numbers are discussed in ¶ 4.7.2 as a subclass of numeric.

4.7.1.1 Enumeration types

An enumeration type is isomorphic to a [0..k] subrange of the integers, without any of t
arithmetic operators. The values are named by literals which have the same syntax as nam
enumeration type {n_0, , ..., n_k} has in its cluster

```
n0: T;                                -- Denoted T[n0]
. . .
nk: T                                 -- Denoted T[nk]
```

Procs to convert between T and INT are lacking.

Enumeration types in interfaces are painted; each type produced by {...} (i.e., by MKENUM
or MKMDENUMERATION) in an interface has a unique mark. Thus two occurrences of an
enumerationTC always produce two different types unless both are in implementations and
textually identical. In this respect, enumerationTCs are like recordTCs and differ from
constructors.

Anomaly: You can write n_i for T[n_i] in an argument or binding. In these contexts, even if known in the current scope, it denotes T[n_i] and not the value it is bound to in the scop

```
Color: TYPE~{red, blue, green};
red: Color_Color[blue];
c: Color_red
```

leaves c=Color[red], not =Color[blue]. In fact, red=red is false! It is best not to redecla names.

Representation: Conversion between T and a short number can be done with a LOOPHOLE. The representation of n_i is the same as that of the INT i (but understand the representation before using LOOPHOLE there).

The type BOOL or BOOLEAN

This is an enumeration type {FALSE, TRUE}; BOOLEAN is a synonym for BOOL. It also has procs:

```
NOT: PROC[BOOL]_[BOOL]                  -- Denoted by prefix NOT or ~.
IFPROC[U: TYPE, test: BOOL,             -- Denoted by IF test THEN "ifTrue"
            ifTrue, ifFalse: PROC[]_[U]]_[U]  ELSE "ifFalse"
```

The meaning of "ifTrue" and "ifFalse" is that in the construct

```
IF test THEN ifTrue ELSE ifFalse
```

the ifTrue and ifFalse expressions are converted into parameterless procs and passed to IF applies the one selected by test. The other one is never applied, so that expression is n evaluated.

Note that AND and OR look like infix operators on Booleans, but have special evaluation r their arguments, because they are desugared into IF expressions (¶ 3.7). The literals TRU can always be written without qualification.

The type CHAR or CHARACTR

This is an enumeration type {'\000, ..., '\377}; CHARACTER is a synonym for CHAR. CHAR liter are written:

- As 'c for any character c except \, denoting the ith CHAR value, where i is the ASCI character code for c.
- As '\ddd, where each d is an octal digit, denoting the dddBth CHAR value. You cannot CHAR['\ddd].
- As '\c for various values of c, denoting the CHAR values for various non-printing o otherwise confusing characters (see rule 57).
- As dddC, denoting the same value as '\ddd (obsolete).

CHAR also has the following dubious procs:

```
•PLUS: PROC[T, INTEGER]_[T]             -- Denoted by infix +.
•MINUS: PROC[T, INTEGER]_[T]            -- Denoted by infix  .
•DIFF: PROC[T, T]_[INTEGER]             -- Also denoted by infix  .
```

Anomaly: The infix " " cannot be desugared into dot notation, since there are two procs by an infix " " whose first argument is a CHAR. The choice between MINUS and DIFF is based the type of the second argument.

4.7.2 Numeric types

Numeric types have arithmetic operations. There are no numeric type constructors, only t
primitive types INT=LONG INTEGER, LONG CARDINAL, INTEGER, CARDINAL and REAL. All except
REAL are subclasses of whole numbers, corresponding to different finite subsets of the in
are discrete as well (4.7.1). The cluster contains:

```
PLUS: PROC[T, T]_[T]                -- Denoted by infix "+".
MINUS: PROC[T, T]_[T]               -- Denoted by infix " ".
TIMES: PROC[T, T]_[T]               -- Denoted by infix "*".
DIVIDE: PROC[T, T]_[T]              -- Denoted by infix "/". Truncates
                                       toward 0:  (i/j)=( i)/j=i/( j)
ABS: PROC[T]_[T]                    -- Denoted ABS[x].
UMINUS: PROC[T]_[T]                 -- Denoted by prefix " ".
```

4.7.2.1 Whole numbers

This class has:

```
REM: PROC[T, T]_[T]                 -- Denoted by infix MOD.  i=j*(i/j)+i MOD j
```

Considerable confusion surrounds Cedar’s treatment of whole numbers. This section gives
but somewhat idealized description of how it works. Then it tells you the hard facts; fu
of Cedar will adhere more closely to the ideal, and this part will shrink. Finally, it d
obsolete facilities whose use is not recommended.

In general, a whole number type (except the CARDINAL types) is a subrange of INT, which is
[$2^{31}..2^{31}$). This means that all the arithmetic procs work on INTS. If an argument of such
subrange value, it is coerced to INT (this cannot lose information or cause a fault), and
coerced to a subrange type if necessary (with a possible BoundsFault). An arithmetic proc
BoundsFault if its result is not an INT (overflow).

Anomaly: In fact, there are two deficiencies in the implementation:

1) There is no overflow checking on the numeric procs.

2) A subrange with $<2^{16}$ values is called short (currently all subranges have this prop
do INTEGER and NAT), If all arguments are short, the result of an arithmetic proc is
truncated to 16 bits (unless it is evaluated statically). This means that the resu
[$2^{15}..2^{15}$), and may differ from the correct result by some multiple of 2^{16}. You can
proper INT arithmetic by writing at least one argument as LONG[x] rather than x. Thu
the program

```
x, y: [0..10000)_1000;
z: INT_x*y;
w: INT_LONG[x]*y
```

initializes w to 1000000 but z to 16960. Beware. This will also happen if x and y a
declared as INTEGER or NAT, since these too are short.

There are several forms of whole number literal, given in rule 57. The radix may be:

Decimal, the default, or specified by D after the number.

Octal, specified by B after the number.

Hexadecimal, specified by H after the number. A hex number may include the letters
through F, denoting the hex digits with decimal values 10 through 15. It must star
digit in the range 0 through 9, however.

The optional number following the radix character is a scale factor, given in decimal; t
zeros are tacked on the end of the number.

Note that literals are always non-negative; a static negative value can be obtained by a
e.g., 1.

Performance: Short values are represented in one word; other INT values require two words
representation is twos complement, with one more negative than positive value. Arithmeti
efficient on subranges with FIRST=0 (except for INTEGER, which is efficient). Widening a sh
value to INT is more efficient if FIRST=0. Multiply and divide are quite slow when the arg
are not short. Short divide is faster when FIRST=0 than for INTEGER.

Cardinal types

The type LONG CARDINAL has elements in the range [0..2^{32}); CARDINAL is the subrange [0..2^{16})
The arithmetic procs produce answers modulo 2^{32} (or modulo 2^{16} if all arguments are short
cardinals). Use of these types is not recommended, mainly because there are confusing co
and from INT. If you program so that these coercions are never invoked, by never mixing
CARDINAL and INT values, you will avoid thses problems; in the future Cedar will not have
coercions, and cardinal types will be harmless.

Complications

•Current Cedar attempts to do the "right" thing when subranges of INT are mixed with subr
LONG CARDINAL in an arithmetic proc, by supplying various coercions which may lose informat
Do not use these features (unfortunately, the compiler won't check for their non-use); i
to understand them, consult a wizard.

4.7.2.2 The type REAL

Cedar uses the IEEE standard 32-bit floating point arithmetic for REALs. There are REAL lit
syntax given in rule 57; they are rounded to the nearest representable number. The expon
present, indicates the power of 10 by which the number or fraction should be multiplied.
that overflows the representation is a static error; one that underflows is replaced by
denormalized approximation. Note that a REAL literal can begin, but not end, with a decim

Exceptions? Changing modes, etc? Is there an interface?

4.7.3 Subrange types

Each discrete type U has a MKSUBRANGE type constructor; its application is denoted by the
rule 25. The first and last arguments specify the first and last elements of the subrange
and LAST items in the subrange cluster have these values. The number of values in the sub
type is last first+1. The subrange is empty if last<first. It is also possible to make an
with first=FIRST[U] using the EMPTYSUBRANGE type constructor. You cannot make an empty
subrange with last=LAST[U].

In current Cedar the arguments of MKSUBRANGE must satisfy
2^{15}<first<2^{15} AND (last first)<2^{16} 1

There is a subrange class for each discrete type, with the names

```
GROUNDTYPE: TYPE;                       -- The type whose MKSUBRANGE or
                                            EMPTYSUBRANGE proc produced T.
TOGROUND: PROC[x: T]_[GROUNDTYPE]       -- A widening coercion.
FROMGROUND: PROC[x: GROUNDTYPE]_[T]     -- A narrowing coercion; may
                                            raise BoundsFault. Apply by T[x].
```

Note that there are coercions both to and from the ground type. The former cannot lose
information or raise an exception, but the latter raises BoundsFault if its argument is n

subrange. Subranges also inherit all the procs of the ground type unchanged; these procs the same arguments, and the coercions make it convenient to apply them to subrange value are no special arithmetic or comparison procs for subranges.

Representation: If T is a subrange type, FIRST[T] is represented the INT 0 (except for INTE which has 0 represented by 0), and LAST[T] by the INT (LAST[T] FIRST[T]+1). The number of bi required to represent a T value is the n such that

$2^{n-1} < (\text{LAST}[T]\ \text{FIRST}[T]+1) < 2^n$

In current Cedar, a subrange value always fits in one word, because a subrange may not h than 2^{16} values.

4.8 TYPE types

All type values have type TYPE. TYPE is not a general type; it lacks SIZE, NEW and the othe procs nearly all types have. Furthermore, in current Cedar a type can't be passed as a p with two exceptions:

An interface type parameter can be declared in a DIRECTORY statement, and the result interface type can be used to declare an interface parameter in an IMPORTS clause. T argument for this parameter is supplied by an implementation which exports the int type.

An opaque or exported type can be declared in an interface module. An implementati the interface provides the actual argument.

A type also can't be returned as a result, with two parallel exceptions:

an interface type is returned by an interface module;

an exported type is returned by an instance of an implementation.

The other possible uses of a type value are these:

Certain primitives take type arguments: CODE, DESCRIPTOR, ISTYPE, LOOPHOLE, NARROW, NEW, SIZE and a number of type constructors.

A type value appears in a declaration, after a colon; e.g., i: INT.

A type value appears as a value bound to a type name; e.g., T: TYPE~INT.

In current Cedar, type expressions and ordinary expressions do not have the same syntax. severe restrictions on where types can be used ensure that the parser can distinguish th a type is expected. There are a few cases where this is not true, and type names (rule 3 written instead of general expressions: subrange type constructors, ???

The runtime type system (ref ???) provides complete facilities for manipulating types du execution of the program (but currently not for constructiong them). The type values it have the type RTT.Type, rather than TYPE. A RTT.Type can be obtained from a TYPE using the primitive:

```
CODE: PROC [T: TYPE]_[RTT.Type].
```

In a number of cases the syntax T[x] (applying a type value) can be used. Depending on t T, the meaning varies. The cases are summarized here, and described in detail in the app section above:

TYPE applied to a static integer n yields an opaque type of size n (¶ 4.3.3).

An array or record type applied to a group or binding yields a record value; this record constructor (¶ 4.4.2.1, 4.6.1).

A sequence-containing record type applied to a not necessarily static CARDINAL yield record type containing a sequence of definite length, which can only be used in NEW SIZE (¶ 4.4.2.2).

A variant record type applied to a static tag value yields a bound variant type (¶

An enumerated type applied to a name which is one of the enumeration literals yiel corresponding enumeration value (¶ 4.7.1.1).

A subrange type (including NAT, INTEGER, or CARDINAL) applied to a value of its ground type yields a subrange value (¶ 4.7.3);

What about TYPE n?

4.9 Miscellaneous types

CONDITION | MONITORLOCK |UNSPECIFIED | LONG UNSPECIFIED

Incomplete

4.10 Kernel-only types

--kernel only-- exception | DECL | BINDING

Incomplete notes follow

```
exceptions
  NEWEXCEPTIONCODE
  VALUE: TYPE=RECORD[ SELECT tag: {normal, exception, hiddenException} FROM
    normal => [v: *T],
    exception => [code: EXCEPTION[*A, *R], args: *A]
    hiddenException => [ex: VALUE[exception] , depth: INT]
    ENDCASE ]
    CURRENTEXCEPTION
  Eval[e]W WITH BasicEval[e]. SELECT FROM
    e: normap => n,
    ex: exception => ex,
    hex: hiddenException => IF hex.depth=1 THEN hex.ex ELSE [hiddenException[hex.ex, hex.de
     1]
    ENDCASE
```

4.11 Defaults

A default in a type cluster provides a value which is supplied automatically in a bindin value is explicitly given. Example:

```
   PutInt: PROC[i: INT, radix: [0..100]_10]
```

makes PutInt[i~x] short for PutInt[i~x, radix~10]. This is very convenient for infrequentl arguments, or if arguments are added to a widely-used proc.

In summary, the usual cases for defaults and bindings are:

Declaration	n: T_	n: T_e	n: T in domain or range decl.
Binding short for			
n~x n~x	n~x	n~x	n~x
n~ or nothing n~OMITTED	ERROR	n~e	ERROR

Table 4 6: Usual cases for defaults

The table says that you can forbid defaulting by writing the defaultTC T_, and you can p default by writing T_e.

Anomaly: The last column says that if you just write T in a proc domain or range declara default is discarded. This means that you can tell by looking at the declaration whether be defaulting, without knowing anything about the defaulting properties of the types.

The basic idea is complicated by an assortment of features for improving efficiency, whi described in the remainder of this section. Defaulting is controlled by two items in the type T, and by two special values. The cluster items are:

> Default: PROC []_[T], a procedure which supplies a default value. If this item is mi values of T cannot be defaulted. Defaulting is done by coercing the special value to T.Default[].
>
> Trash: PROC []_[T]; a procedure which supplies a trash value of type T, a haphazard collection of bits of the same size as a value of type T. If this item is missing, cannot be trashed. The main virtue of this procedure is that is executes very fast description of TRASH below.

The CHANGEDEFAULT primitive makes a new type with these items modified. It cannot be writte a program, but is invoked by the syntax for defaultTC.

CHANGEDEFAULT: PROC[OldT: TYPE, Default: PROC []_[T], TrashOK: BOOL]_[NewT: TYPE]

NewT has the same predicate and cluster as OldT, except that:

> NewT.Default is Default. If Default is NIL, it is copied from OldT.Default, or omitted missing.
>
> NewT.Trash is copied from OldT.Trash if TrashOK=TRUE; a missing OldT.Trash causes an error. It is omitted if AllowTrash=FALSE.

As described earlier, a type in a proc domain or range which is not a defaultTC has its Trash procs omitted.

The two special values cannot be written explicitly in a program. Instead, they are supp following syntax:

> OMITTED in a binding constructor the syntax n~ , which omits the value, means n~OMIT Then if there is a DefaultProc, OMITTED is coerced to T.Default[] to provide a value There is also a coercion which adds n~OMITTED to a binding which lacks n, so that n left out entirely with the same effect as writing n~ .
>
> In a group (constructor without names), an empty element means OMITTED; note that th group is then coerced to a binding by attaching the binding's names to the group e in order (¶ 2.2.6).
>
> TRASH a binding can specify this value explicitly with the syntax n: TRASH. It is un use TRASH if the program uses the value. Its purpose is to avoid the cost of initia variable which is going to be reinitialized before it is read.

The effect of these rules is that binding [n_1~e_1 ...] to [n_1: T_1 ...] has the same effect
[n_1~ ...], [...], or [, ...] to [n_1: T_1_e_1 ...] (assuming that any free variables have t
both cases).

Primitive types and those returned by primitive type constructors (except CHANGEDEFAULT) ha
Trash proc and a Default proc equal to the Trash proc, with the following exceptions:

CONDITION, MONITORLOCK and PORT have no Trash or Default; they do have an INIT proc which sets any variable to NIL.

REF and PROC types have no Trash and a NIL Default.

Bound variant records have no Trash and a Default which sets the tag value appropri

Record and array types have a Trash or Default if all their component types do; it obvious concatenation of the component procs.

Including the various dangerous uses of TRASH which omit initializations, we get a larger confusing summary table which should be ignored except by efficiency hackers.:

Default type constructor	T_	T_e	T_e | TRASH	T_TRASH	T in domain/ range decl
Default		l [] IN el [] IN e		T.Trash	
Trash			T.Trash	T.Trash	

Declaration						
Binding	short for	n: T_	n: T_e	n: T_e | TRASH	n:T_TRASH	n: T
n~x	n~x	x	x	x	x	x
n~ or nothing	n~OMITTED	ERROR	e	e	T.Trash[]	ERROR
n~TRASH	n~TRASH	ERROR	ERROR	T.Trash[]	T.Trash[]	ERROR

Table 4 7: Complete cases for defaults

4.12 Implies

A type T implies another type T((TgT(for short) if for any value x,
T.PREDICATE[x]gT(.PREDICATE[x]
In other words, if any value that has type T (satisfies T's predicate) also has type T(,
T(. A consequence is that a proc with domain type T(can safely be given a value of type
this value must also have type T(, as required by the proc. We also say that a T value i
a T(value.

If T's predicate includes a test for some mark, then any type which implies T must test
mark or a bigger one. For instance, if R is a variant record type with variants a, b, an
R[a]gR if SIZE[R[a]]=SIZE[R]. In fact, the predicate for R[a] tests for R's mark and for a
to a. In other words, a bound variant value is as good as an unbound one.

From the implementation's viewpoint (and after all, it is the implementation of an abstr
is responsible for attaching marks), two values should have the same mark only if they b
representations with all the properties implied by that mark: occupy at least that much
the proper fields interpreted in the proper way, etc. This is the rationale for marks: t
values which are not acceptable to the same primitives. Of course this is not an enforce
nothing prevents an implementation from unwisely allowing the marks it controls to be ap
unsuitable values.

For example, [0..5]g[0..7] because both occupy four bits and represent the integer unbia
[1..5] does not imply [0..7], because it happens that the implementation biases the repr
a subrange value, so that the value 1 is represented in [1..5] by binary 0000, but in [0
0001. [1..5] and [0..7] must have different marks, but [0..5] and [0..7] can have the sa
might be called "four bit unbiased representation for unsigned integer"), and distinguis
with the rest of their predicates (0<x<5 vs 0<x<7).

For T to imply T(, there must be a proof that T's predicate implies T('s predicate. If T
arbitrary type, and nothing is known about its relationship to other types, or if it tes
mark, then no such proof is possible. As a result, only an argument with syntactic type
acceptable to a T_R proc. For built-in types and type-returning procs, however, the compi
knows the predicates and keeps track of the implications. The implies relations among bu
are (the transitive closure of those) specified in the following table. Any argument omi
type proc applications in the table may take any legal value, but it must take the same
both applications in a single row.

Certain points about the table are of special interest:

The first line says that implies extends elementwise to declaration types.

The line for transfer types (including PROC) says that (D_R)g(D(_R() if D(gD and RgR(. The relation is reversed for the domain types, because a D(_R(proc P(must a any D(, while a D_R proc P only accepts Ds. If P is used in the former context, it guaranteed to get a D(, and that must imply a D.

There are no implications of the form VAR TgVAR U. You might think that TgU should imply this, but it doesn't work, because a VAR can be assigned to, and assigning a [0..7]) to a T (say a [0..5]) clearly won't do. So a VAR T can't be as good as a VA can be assigned a U value. In fact, if there were write-only VARs, the relation wou backwards. This is a reflection of the fact that the only interesting operation on is assignment, which has the type [VAR T, T]_[T]; as we have seen, proc type implic backwards from the domain type implication.

Any argument omitted from the type constructor applications in the table may take any le
but it must take the same value in both applications in a single row.

In current form			In kernel form	
These types	Imply these types	Remarks	These types	Imply these types
[n: T , ...]	[n: T(, ...]	if T⊐T(	[n: T , ...]	[n: T(, ...]
Pointwise extension to bindings. Likewise for groups.				
T	U declared by U: TYPE		T	FULLYOPAQUE
is fully opaque.				
T	U declared by U: TYPE[n]	if SIZE[T]=n ...	T	OPAQUE[n]
... and T has the standard NEW, INIT, ASSIGN and EQUAL procs. U is n-opaque.				
	READONLY T		VAR T	READONLY T
READONLY T	READONLY T(	if T⊐T(	READONLY T	READONLY T (
PROC/ERROR/... [T] RETURNS [U]	PROC/ERROR/... [T(] RETURNS [U(]	if T(⊐T and U⊐U(	MKXFERTYPE[domain~T, range~U]	MKXFERTYPE[domain~T(, range~U(]
Note the reversed implication for the domain type.				
SAFE PROC/ERROR/...	UNSAFE PROC/ERROR/...		MKXFERTYPE[safe~TRUE]	MKXFERTYPE[safe~FALSE]
ARRAY ... OF T	ARRAY ... OF T(	if T⊐T(	MKARRAY[range~T]	MKARRAY[range~T(]
If PACKED=FALSE or SIZE[T]>1. If PACKED=TRUE and SIZET]=1, the number of bits required to represent a T and to represent a T(must be equal when rounded up to the next power of for SEQUENCE and DESCRIPTOR.				
REF T	REF READONLY T		MKREF[range~T, readOnly~FALSE]	MKREF[range~T, readOnly~TRUE]
and likewise for POINTER and LIST.				
REF READONLY T	REF READONLY T(	if T⊐T(	MKREF[range~T, readOnly~TRUE]	MKREF[range~T(, readOnly~TRUE]
and likewise for POINTER and LIST.				
REF T	REF ANY		MKREF[range~T]	MKREF[range~ANY]
ORDERED POINTER TO T	POINTER TO T		MKPOINTER[range~T, ordered~TRUE]	MKPOINTER[range~T, ordered~FALSE]
BASE POINTER	POINTER	and vice versa	MKPOINTER[base~TRUE]	MKPOINTER[base~FALSE]
T[tag: x]	T	if SIZE[T[tag: x]]= SIZE[T]	???	???
A bound variant implies the unbound variant.				
RECORD[n: T]	T	1-element record	MKRECORD[fields~[n: T]]	T
and likewise for MACHINE DEPENDENT RECORD.				
(PROC[A]_[n: T]).RANGE	T	1-element binding	[n: T]	T
(PROC[A]_[T]).RANGE	T	1-element group	CROSS[[T]]	T
T[x...y] etc.	T		T.MKSUBRANGE[x, y]	T
if T.FIRST=x and SIZE[T[x...y]]=SIZE[T].				
T[x...y] etc.	T([x...y] etc.		T.MKSUBRANGE[x, y]	T(MKSUBRANGE[x, y]
if T.FIRST=T(.FIRST and T.LAST<T(.LAST.				

Table 4 8: Implies relations for primitive types

4.13 Coercions

In an application, the argument value must have the proc's domain type (¶ ???). To ensur
does, an expression f [e] which is an application is type-checked by requiring De to impl
domain type D of Df (¶ ???). If it does not, an attempt is made to find a coercion functi
De_D which can map the argument to the required type. If C is found, the application is
rewritten as f [C[e]], which typechecks. We say that e is coerced to the type D.

A coercion may also be done in a binding such as pi: REAL=3; this is actually a special c
application. Note that infix operators, including assignment, are special ways of writin
and hence also do coercions. In particular,

```
x: REAL; x_3
```

will coerce 3 to a REAL.

There are no coercions from VAR T to VAR U; this is because coercing produces a new value,
new VAR would be disjoint from the old one and would increase the size of the state, whic
unlikely to be what is wanted.

Note that if T implies U (see ¶ ???), no coercion from T to U is needed to make an applic
check. Another way of thinking about this: TgU means that there is a coercion function f
U, but it does no computation. This is why REF T can be coerced to REF U if TgU

A group or binding can be coerced element by element. Formally, a declaration type, whi
type of a binding, has one coercion for each coercion that an element type has. These ca
composed to coerce several elements.

There is currently no way for the program to specify coercion procs. However, there is a
of built-in coercions, which are are listed in the following table. None of them loses i
except those from various whole numbers to REAL; in other words, they all have inverses.
them can raise an exception, except a coercion from a base type to a subrange, which can
Runtime.BoundsFault. Any argument omitted from the type proc applications in the table may
any legal value, but it must take the same value in both applications in a single row.

In current form These types can be coerced to these types	Remarks	In kernel form These types can be coerced to these types
[..., n: T , ...]	[..., n: T(, ...]	if T coerces to T(	[..., n: T , ...]	[..., n:T(,...]
This is pointwise extension of coercion to bindings. Likewise for groups.				
$[T_1, ..., T_k]$	$[n_1: T_1, ..., n_k: T_k]$	group to binding	$T_1 X ... X T_k$	$[n_1: T_1, ...; n_k: T_k]$
$[n_1: T_1, ..., n_k: T_k]$	$[n_1:T_1, ..., n_k:T_k, n: T]$	if T has a default	$[n_1: T_1, ...; n_k: T_k]$	$[n_1:T_1, ...;T_k n: T]$
T	T(	if TgT(	T	T(
T[x..y]	T		T.MKSUBRANGE[x, y]	T
T	T[x...y]	may raise Runtime.BoundsFault	T	T.MKSUBRANGE[x, y]
and the same subrange coercions for relative address types.				
INT/INTEGER/ CARDINAL/ LONG CARDINAL	REAL	loses information	same	
POINTER	LONG POINTER		MKPOINTER[long~FALSE]	MKPOINTER[long~TRUE]
and likewise for DESCRIPTOR.				
T[tag: x]	T	bound variant variable to value	??? VAR T	??? T

Table 4 9: Coercions for primitive types

4.14 Dot notation

Cedar provides a single basic mechanism for getting a name looked up in a particular bin rather than in the current scope (¶ 2.4.4):

> If b is a binding, then b.n is the value of n in b; it is an error if there is no e named n.

By a natural extension:

> If T is a type, then T.n is the value of n in T's cluster.

By a somewhat less natural, but very useful further extension (inspired by classical not records, and by Smalltalk):

> If e is an expression not a type or binding, then let P=(De).n.
>
> If P.DOMAIN=[p: D], then e.n is P[e].
>
> Otherwise, if P.DOMAIN=$[p_1: D_1, p_2: D_2, ..., p_n: D_n]$, e.n is l $[p_2: D_2, ..., p_n:$
>
> P[e, p_2, ..., p_n]

In other words, the value of n is obtained from the cluster of e's syntactic type. If it argument, it is applied to e. Otherwise. e.n is a proc which collects the other arguments P wants, and applies P to e, p_2, ..., p_n. In current Cedar you can't do anything with thi apply it immediately: you have to write e.n[...].

There are four major applications for dot notation in current Cedar; they are described below. All use the simple rules just stated (look up n in a binding; in the cluster of a cluster of De and then apply it). But the sources of the clusters used and the procedure the clusters are quite various.

Object notation is the most general, since any opaque or record type T defined in an int
acquires a user-defined a cluster by this method. The current implementation is rather c
the procs in the interface I from which T comes are added to T's cluster, with the names
in I, except those whose names are already in T's cluster. Of course, an element of this
only useful if it takes a T as its first argument.

The interface I from which P is obtained is normally an interface instance I (which is in
not an interface type IT (declared in the DIRECTORY clause), because only the instance pro
proc value for P. Of course if P is bound to an INLINE in IT this is not true. See ¶ 3.3 f
interfaces.

Restriction: In current Cedar, the value for P always comes from the principal imported
IT (see ¶ 3.3.3). This is of no concern if only one IT value is imported. If more than or
imported, however, confusion can result. If it does, consult a wizard.

The cluster for a record type R is formed automatically by the record type constructor,
contains a procedure for each field f: T_f, which takes an R and returns a T_f. There are
clusters for VAR R and READONLY R, in which the procedures take VAR or READONLY R and return
VAR or READONLY T_f.

An interface type yields a binding, which contains those names which are bound in the in
rather than simply declared (usually constants and types, sometimes inline procs). An in
(imported interface) can also be thought of as a binding, with a value for each name in
interface. Actually it is more like a record; its cluster contains a proc for each name declared in the int
returns the exported value when applied to the interface value.

Case	Source for n	De.n	e.n	e.n[p_2~x, ...]
Meaning		can't write this literally.	(De).n[e]	(De).n[e][p_2~x, ...] or (De).n[p_1~e, p_2~x, ...]
Object notation (De must be record or opaque type).	n: PROC[self: T_T declared in same interface I as De. Useless unless De coerces to T.	I.n	WI.n[e], since nWI.n	*
	n: PROC[self: T, p_2: T_2, ...] declared in same I as De. Useless unless De coerces to T.	I.n	No (can't get the value of the curried proc).	WI.n[self~e, p_2~x, ...]
Record	RECORD [..., n: T, ...]	No (can't get the record selector value).	Wa VAR T for field n of record e.	*
Imported interface	IT: DEFS{...; n: T; ...}; DIRECTORY IT: TYPE; IMPORT e: IT;	No (can't get the interface selector value).	W the value exported as n in the e instance of IT.	*
Interface type	IT: DEFS{...; n: T~v; ...}; DIRECTORY e: TYPE IT;	No (it would be TYPE.n).	Wv (need a binding for n, not just n: T).	*

* Only if T is a proc type with the right domain.

Table 4 10: Cases for dot notation in current Cedar