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 (¶ 32. less formal explanation of the ideas of the kernel and the restrictions imposed by Cedar (¶¶ 32.3-2.9). ¶ 32.1 contains an overview or glossary, in which the major t terms used 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 (¶ 34.1).

In addition, there is a one-page grammar for the full language, a shorter grammar for th language, and a two-page language summary which includes the grammar, the desugaring, an examples from ¶ 3,

The document you are reading is nearly complete. A fe missing sections co paragraphs in the style of this one.

Chapter 2. The kernel language

This document describes the Cedar language in terms of a much smaller language, which we usually call the kernel or the Cedar kernel. Cedar differs from the kernel in two ways:

- It has a more elaborate syntax (¶ 3). The meaning of each construct in Cedar is ex by 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 (¶ 4). In the kernel langua these could in principle be programmed by the user, though in fact most are provid special code in the Cedar compiler. In general, you can view these built-in facili like a library, selecting the ones most useful for your work and ignoring the othe

Unfortunately, the current Cedar language is not a superset of the kernel language. Many objects (notably types, declarations and bindings) which are ordinary values in the kern be freely passed as arguments or bound to variables, are subject to various restrictions they can only be written in literal form, cannot be arguments or results of procedures, The long-term goal for evolution of the Cedar language is to make it a superset of the k defined here. In the meantime, however, you should view the kernel as a concise and hope clear way of describing the meaning of Cedar programs.

To help in keeping the kernel and current Cedar separate, reserved words and primitives kernel which are not available in current Cedar are written in SANS-SERIF SMALL CAPITALS, rat than the SERIF SMALL CAPITALS used for these symbols in current Cedar. Operator symbols of t kernel which are not in current Cedar 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 some 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 is a commentary which explains the behind the kernel. It also gives the restrictions imposed by the current Cedar language generality described here; for more on this subject, see ¶ 3. The meaning of the various primitives is given in ¶ 4. ¶ 2.9 describes the incompatibilities between the kernel lan current Cedar, i.e., the constructs in Cedar which would have a different meaning in a k program. For the most part, these are bits of syntax which do not have consistent meanin current Cedar; future evolution of the language will replace them with their kernel equi

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

Values and computations: Application Value Variable Group Binding Argument

The type system: Type Type-checking Mark Cluster Declaration Class

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 given there.

2.1 Overview

This section gives a brief summary of the essential concepts on which the Cedar language
The explanations are informal and incomplete. For more precise but more formal definitio
¶ 2.2; for more explanation, see ¶ 2.3-2.8.

2.1.1 Values and 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 [first~x, last~y]`. There are special ways
many kinds 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
Nearly every type `T` has a corresponding variable type `VAR T`; values of type `VAR T` contain
of type `T`. Every `VAR` type has a `NEW` proc which creates a variable of the type. A variable
represented by a single block of storage; the bits in this block hold the representation

Group: A group is an ordered set of values, often denoted by a constructor like this: [3
"Hello"]. Like everything else, a group is 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]`. If `b` is a binding, `b.n` denotes the value
name `n` in `b`.

Argument: A group or binding constructor written explicitly after an expression denotes
of the value `P` denoted by the expression to the value `a` denoted by the constructor, call
argument. `P` is usually a proc, and `a` is a group or binding, which is bound to its domain
`D` to get the argument which is passed. In making this binding `a` is coerced, if necessary
the declaration:

If it is a group, the names from `D` are attached to its elements to turn it into a bindi
If a name in `D` is missing from `a`, a default value is supplied.
If a value in `a` doesn't have the type required by `D`, it is coerced into another value w

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 specified properties. A value may have many types; i.e., it may be i
these sets. A type also collects together some procs for computing with the value (e.g.,
multiply).

More precisely, a type is 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 the value of 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 range declared for its outermost proc; in general this may depend on the argum
value of e always has this type (satisfies this predicate); of course it may have other

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
This does not imply a total ordering on the marks.

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 parameters satisfy their type predicates. I
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: parameters 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
[x: INT, y: BOOL]. If d is a declaration, a binding b has type d if it has the same set of
each name n the value b.n has the type d.n. A binding b matches d if the values of b can
to yield a binding b(which has type d.

A declaration can be instantiated (e.g., on block entry) to produce a binding in which e
bound to a variable of the proper type; instantiating the previous example yields
[x: VAR INT~(VAR INT).NEW, y: VAR BOOL~(VAR BOOL).NEW].

Class: A class is a declaration for the cluster of a type. For instance, the class Ordere
PROC[T, T]_[BOOL], . . .]. C is a subclass of D if (loosely) C includes at least all the [
pairs in D.

2.1.3 Programs

Name: A name (sometimes called an identifier) appearing in a program denotes the value b
the name in the scope that the name appears in (unless the name is in a pattern before a
(declaration) or tilde (binding), or after a dot or $). An atom is a value that can be u
a name; a literal atom is written like this: $alpha.

Expression: In a program a value is denoted by an expression, which is one of:
a literal value (3 or "Hello");
a name (x or salary);
an application of a proc to other values(Sin[90];
GetProperties[directory, ReadFileName[input]]);
a l-expression, which yields a proc value (l [x: INT] IN (IF x<0 THEN x ELSE x));
a constructor for a declaration or binding ([x: INT~3, y: REAL~3.14]).
If a value is given for each free name in an expression, then it can be evaluated to pro
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 called ENV (for environment) which determines these values. A new scope is introc
the kernel) by IN (after LET or l) or by a REC [...] constructor for a declaration or binc
LET x~3 IN x+5;
LET fact~l [n: INT] IN IF n=0 THEN 1 ELSE n*fact[n 1].

Constructors: Brackets delimit explicit constructors for group, declaration or binding v
all have the form [x_1, x_2, ...], and are distinguished by the form of the x_i:
an expression for a group;
n: e for a declaration;
n~e or n: e~e for a binding.

Recursion: When names are introduced in a constructor in Cedar, this is done recursively
If v is bound to n in a binding constructor, then in expressions in the constructor n h
rather than its value in the enclosing scope. Exception: argument bindings are non-rec
If n is declared in a declaration constructor, then it may not be used in the construct
an ordering of the declarations in the constructor such that a name is used only by la
Exception: declared names may be used in the bodies of l-expressions in the constructo
¶ 3.3.4)
In the kernel, however, constructors are non-recursive unless preceded by REC

Dot notation: The form e.n looks up n in some binding associated with e, and does someth
the result. There are three cases:
If e is a binding, e.n is just the value paired with n in e.
If e is a type, e.n is e.Cluster.n.
Otherwise, e.n is (De.n)[e], and e.n[more args] is usually (De.n)[e, more args].
In all cases you are supposed to think of n as some property or behavior associated with
denotes that property or evokes that behavior.

2.1.4 Conveniences

Coercion: Each type cluster contains To and From procs for converting between values of
and values of other types (e.g., Float: PROC[INT]_[REAL]; this would be a To proc in REAL an
from proc in INT). One of these procs is applied automatically if necessary to convert or
argument value to the domain type of a proc; this application is a coercion. Each coerci
associated atom called its tag (e.g., $widen for INT_REAL or $output for INT_ROPE); several
coercions may be composed into a single one if they have the same tag. The tags thus ser
prevent unexpected composition of coercions.

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, called the argument of the exception, so that arbitrary information can be passed
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 the proc value is safe;
trusted, the programmer asserts that it is safe (but the compiler makes no checks), and
is treated as 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 a time. They all share the same address space. Share
(touched by more than one process) can be protected by a monitor; only one process can e
within the procs of the monitor at a time. So that each process can know what to rely on
necessary to establish an invariant for the monitored data which is established whenever
proc returns or waits. A process can wait on a condition variable within a monitor; othe
can then enter the monitor. The waiting process runs again when the condition is notifie
timeout.

2.1.5 Miscellaneous

Allocation: Cedar has standard facilities for allocating new variables of any type (the
primitive); related variables can be allocated in the same zone. Normally, variables are
automatically by the garbage collector when they can no longer be referenced; such varia
only be referred to by REFS. It is also possible to have variables which are deallocated
FREE, but this is unsafe.

Static: An expression whose value is computed without executing the program is called st
Literals are static, as are names bound to literals, and any expression with static oper
bodies are never static unless they are inline

Pragma: Some language constructs do not affect the meaning of the program (except possib
make a legal program illegal), but only its time and space costs; these are called pragm
are INLINE for proc bodies and PACKED for arrays.

2.2 Kernel definition

This section gives the syntax and semantics of the Cedar kernel language. Motivation, an
explanation of the relation between the kernel and the current Cedar language, can be fo
¶¶ 2.3-2.7. The kernel is subdivided into

A rather austere core; everything can be desugared into this, but it isn’t very re

A set of conveniences; with these, readable programs can be written.

Imperative constructs: statements and loops.

Exception handling.

The format of this section interleaves grammar rules which give the syntax of the langua
which gives the meaning. The meaning of the core is given in English. For other parts of
kernel, it is given by desugaring rules which show how to rewrite each construct in term
if rewriting is done repeatedly, the result is a core program, which may invoke some pri
meaning of these is also given in English. There is also some English explanation of the
but this is only a commentary and does not have the force of law.

See ¶ 3.1 for the notation used in the grammar and desugaring.

2.2.1 The core

In the core, there is syntax only for names, literals, application, l-expressions, a bas
recursive binding construction, and syntactic type; everything else is done with primiti
write anything in the core, however, except to show the desugaring of a kernel construct
reader need not struggle with programs in the ugly core syntax.

Syntax	Syntactic type	Meaning
expression ::=		
n |	Dn	ENV[$n].val
literal |	Dliteral	
e_1 Z e_2 |	$(De_1.\text{RANGE})[e_2]$	-- Standard application. |
l d_1=> d_2 IN e |	d_1_d_2	-- Standard proc constructor. |
L d_1=> d_2 IN e |	d_1_d_2	-- Unchecked standard proc constructor. |
[(n~e), ...]	[(n: De), ...]	-- Vanilla binding constructor. |
FIX d ~ e |	d	-- Recursive binding constructor. |
D e	TYPE	-- Syntactic type.
type ::= e	De --gTYPE--	-- A type is syntactically just an expression.
decl ::= e	De --gDECL--	-- A decl is syntactically just an expression.
name ::=		
letter (letter | digit)...	(Dlitv)n..	-- Appears as an e or in a pattern.
literal ::=		
$ n |	ATOM	-- ATOM literal. |
primitive	Dprimitive	
primitive ::=		
ARROW |	[d: DECL, p: (d_DECL)]_[a: --arrow--TYPE]	
DOMAIN | RANGE |	*[a: --arrow--TYPE]_[t: TYPE]	
MKPAIR |	$[t_1$:: TYPE, first: t_1, t_2:: TYPE, rest: $t_2]$_[v: t_1X$t_2]$	
GROUP |	$[t_1$: TYPE]_[t: TYPE] --tgTYPE	
MKCROSS |	[g: GROUP[TYPE]]_[c: --cross--TYPE]	
CDOTG |	*[t: --cross--TYPE]_[g: GROUP[TYPE]]	
MKBINDD |	[d: DECL, v: d.T]_[b: d]	
BDOTD | BDOTV |	[b: BINDING]_[d: DECL] | [d:: DECL, b: d]_[v: d.T]	
MKBINDP |	[p: PATTERN, t:: TYPE, v: t]_[b: MKDECL[p, t]] --=MKBINDD[d~MKDECL[p, t], v	
LOOKUP |	[d:: DECL, b: d, n: ATOM]_[v: DTOB[d].n]	
THEN |	$[d_1$:: DECL, b_1: d_1, d_2:: DECL, b_2: $d_2]$_[v: d_1 THEND d_2]	
ENV |	*BINDING	
MKDECL |	*[p: PATTERN, t: TYPE]_[d: DECL]	
DDOTP |	*[d: DECL]_[p: PATTERN]	
DDOTT |	*[d: DECL]_[t: TYPE]	
DTOB |	*[d: DECL]_[b: BINDING]--=MKBINDP[p~d.P, v~d.T.G]]	
BTOD |	*[b: BINDING]_[d: DECL]--=MKDECL[p~b.D.P, t~MKCROSS[b.V]]	
THEND |	$[d_1$: DECL, d_2: DECL]_[v: DECL] --=BTOD[DTOB[d_1] THEN DTOB[d_2]]	
BOOL | ATOM |	TYPE	
TRUE | FALSE |	BOOL	
TYPE | DECL | BINDING |	TYPE	-- DECLgTYPE, BINDINGgTYPE
PATTERN |	TYPE	--=GROUP[ATOM]
ANY	TYPE	-- TgANY for any type T

In the kernel we dress up the primitives as follows:

A primitive denoted xDOTy is in the cluster of the type of its argument under the n

A parameter to a primitive declared with :: is the type of some other argument; th
argument for this parameter may be omitted in an application of the primitive.

A name not in a literal (or pattern, in the kernel) denotes the value to which it is bou current environment ENV (A below). An ATOM literal is a value which stands for a name in t primitives which deal with declarations and bindings.

A literal denotes a value according to a rule which depends on its syntax. The core has numeric and ATOM literals, and the primitives enumerated above.

An expression denotes a value according to a rule which depends on its syntax. If the ex a name or literal, the value is the value of the name or literal. The remaining cases ar the following sub-sections. Most of these cases define the value of the expression in te value of its sub-expressions. The sub-expressions may be evaluated in any order.

A. The current environment ENV

The current environment ENV is a binding. The value of the expression n is ENV.n. ENV for expression is the same as ENV for its containing expression, except that:

For the b of a closure being applied, ENV is computed according to B below.

For the e of a FIX, ENV is computed according to E below.

Thus applying a closure and evaluating a FIX are the only ways to change ENV.

B. Application

The value of a standard application is obtained by evaluating e_1 and e_2 to obtain v_1 and applying v_1 to v_2. There are two cases for application:

v_1 is a primitive. The value of the application is a function of v_2 given in the de the primitive.

v_1 is a closure c (C below), with domain declaration d, body b and environment E. T value of the application is the value of the expression b in the environment MKBINDD THEN E (D below). Note that if the closure was made with L, the body must be type- checked when it is applied; a closure made with l was type-checked when it was mad below).

An application type-checks if De_2 implies De_1.DOMAIN (G below).

C. Lambda

The value of a l-expression is a closure, which has three parts:

A domain declaration d, equal to the value of d_1.

A body b, which is the expression e (not the value of e).

An environment E, equal to the current environment ENV.

A l-expression type-checks if

d_1 evaluates to a declaration d.

For any x of type d.T, De implies d_2.T in the environment MKBINDD[d, x] THEN E.

A L-expression type-checks if d_1 evaluates to a declaration; type-checking of the body is until the closure is applied.

D. Pairs, groups and cross types

A pair is the basic structuring mechanism. MKPAIR[x, y] yields the pair <x, y>. Bigger str made, as in Lisp, by making pairs of pairs. When we are interested in the leaves of such we call it a group and call the leaves its elements. A group has type GROUP[T] if all its have type T or are NIL. A flat group is a pair in which first is not a group, and rest is NIL.

The type of a pair is a cross type: MKPAIR[x, y] has type TXU iff x has type T and y has t Cross types are made with MKCROSS, which turns a GROUP[TYPE] into a cross type in the obviou way:

MKCROSS[NIL]=???

MKCROSS[T]=T if T is a type.

MKCROSS[MKPAIR[x, y]]=MKCROSS[x]XMKCROSS[y]

Note that MKCROSS of a flat group is flat. CDOTG goes the other way, turning a cross type i GROUP[TYPE] in which no element is a cross type. Thus MKCROSS is the inverse of CDOTG, but n necessarily the other way around.

E. Bindings

A binding is either NIL, or an <atom, value> tuple, or a <binding, binding> tuple. The pr MKBINDD constructs a binding from a declaration d and a matching value v, i.e. (as the typ MKBINDD indicates), one with the type d.T. The resulting binding has type d, and consists names from d paired with the corresponding values from v. Example:

MKBINDD[[x: INT, b: BOOL], [3, TRUE]] = [x~3, b~TRUE]
= < <$x, 3>, < <$b, TRUE>, NIL > >

In this example, d.T is INTXBOOL.

The declaration and group in this example is written using the syntax of ¶ 2.2.2; in the core they would be MKDECL[p~[$x, $b], t~MKCROSS[[INT, BOOL]]] and MKPAIR[first~3, rest~MKPAIR[first~TRUE, rest~NIL]], where we have written the arguments of these primitives in the kernel syntax.

The primitives BTOD and BTOV return the arguments of the MKBINDD primitive that made the binding. MKBINDP is redundant; it is like MKBINDD, but takes a type instead of a declaratio hence accepts any v with the right structure.

LOOKUP returns the value of the name n in the binding. THEN combines two bindings, giving priority to the first one in case of duplicate names. It works only for flat bindings, i element of each <binding, binding> tuple is an <atom, value> tuple, and the second eleme another <binding, binding> tuple or NIL. The value of b_1 THEN b_2 is another flat binding, by first replacing any tuple <<a, v>, b> in b_2 where a is equal to an atom in b_1 by b, and this binding to replace the final NIL in b_1.

The binding constructor [..] has the value MKBINDP[p~[n, ...], v~[e, ...]].

FIX makes a recursive binding: the value of FIX d_1~e is MKBINDD[d, v], where d is the value ENV and v is the value of e in the environment (LET FIX d~e IN d~e) THEN ENV. Of course in g this computation may not terminate; normally the names in d occur in e only with l-expres and in this case it does terminate. Is this really right? The type checks if De in the latter environment implies DTOT[d].

F. Declarations

A declaration is either NIL, or an <atom, type> tuple, or a <declaration, declaration> tu
primitive MKDECL constructs a decl from a pattern p and a value t of type GROUP[TYPE]. A pa
a GROUP[ATOM], i.e., either NIL, or an atom, or a pair of patterns; the ATOM elements must
different. An application of MKDECL typechecks if t matches p, i.e., if

both p and t are NIL, or

p is an atom and t has type TYPE, or

p is a pair [p_1, p_2] and t is a cross type $t_1 X t_2$ and p_1 matches t_1 and p_2 matches t_2.

The resulting declaration consists of the names from p paired with matching type values

The primitives DDOTP and DDOTT return the arguments of the MKDECL primitive that made the declaration. Thus

DDOTT[NIL]=???;

DDOTT[<$n, T>]=T;

DDOTT[<d_1, d_2>]=DDOTT[d_1]XDDOTT[d_2]

DTOB is redundant; it converts a declaration to a binding in which each name has the corr
type as its value. Thus DTOB[[x: INT, y: REAL]]=[x~INT, y~REAL]. The inverse is BTOD, also
redundant; it is defined only if all the values in the binding are types. THEND combines
declarations just as THEN combines two bindings: D(b_1 THEN b_2)=Db_1 THEND Db_2

G. Types and type-checking

A type is a value consisting of a pair:

the predicate, a function from values to BOOL.

the cluster, a binding.

A value v has type T if T's predicate applied to v is TRUE.

T implies U iff (Ax) T.Predicate[x]gU.Predicate[x].

Typechecking consists of ensuring that the argument of an application has the type speci
domain of the proc (B above). The body of a l-expression can then be type-checked (or th
implementation of a primitive constructed) independently, assuming that the parameter sa
domain predicate. Symmetrically, the result of an application can be assumed to have the
specified by the range of the proc.

To complete the induction, it is also necessary to check that the value of the body of a
has the range type (C above).

The primitive types in the kernel are:

BOOL, with two values TRUE and FALSE.

ATOM, with values denoted by literals of the form$n.

TYPE, a predicate satisfied by any type value.

ANY, a precidate satisfied by any value.

DECL, the type of a declaration (F above).

BINDING, the type of any binding.

Arrow types, the types of procs (C above). An arrow type has a domain type and a r type.

Cross types, the types of pairs (D above).

GROUP[T], the type of any pair in which all the elements have type T.

Declarations, the types of bindings (E and F above).

There are no non-trivial implications among any of these types, except as follows:

DECL≥TYPE; BINDING≥TYPE; GROUP[T]≥TYPE.

ANY≥T for any type T.

$T_1 \times T_2 \geq U_1 \times U_2$ iff $T_1 \geq U_1$ and $T_2 \geq U_2$.

GROUP[T]≥GROUP[U] iff T≥U.

$T_1 \to T_2 \geq U_1 \to U_2$ iff $U_1 \geq T_1$ and (Ax: U_1) (l T_1 IN T_2)[x]≥(l U_1 IN U_2)[x]. Note the reversal of the domains.

$d_1 \geq d_2$ for declarations iff d_1.P=d_2.P and DTOB[d_1].n≥DTOB[d_2].n for each n in d_1.P.

2.2.2 Conveniences

```
expression ::= coreExpression |
  d1 _ d2 |                          ARROW Z [d1, l d1=>DECL IN d2]
  l ( | e1) ( | => e2) IN e3 |       -- The domain defaults to [], the range to x: De3 |
  LET e1 IN e2 |                     ( l De1 IN e2 Z )e1.v -- e1 a binding |
  LET b, ... IN e |                  LET [b, ...] IN e
  IF e1 THEN e2 ELSE e3 |            ( IFPROC[De2, e1, l IN e2, l IN e3] )
  e . n |                            IF DegBINDING THEN LOOKUP Z [De, $n]
                                     ELSE IF DegTYPE THEN LOOKUP Z [De.cluster, $n]
                                      ELSE (LOOKUPC Z [De.cluster, Z$n])|
  e1 [ b,  ... ] | e1 [ e2,  ... ] | e1 . APPLY  Z [b, ...] | e1 . APPLY  Z MKBINDD[De1.DOMAIN
  e1 infixOp e2 |                    e1 . infixOp[e2] |
  e1 AND e2 | e1 OR e2 |             IF e1 THEN e2 ELSE FALSE | IF e1 THEN TRUE ELSE e2 |
  [ ] | [ e1 ( | e2, !.. )] |       NIL | MKPAIR[e1, [ ( | e2, Group ) ]] constructor. |
  PATT p |                           -- Pattern constructor; see the rule for p below. |
  [ b, ... ] |                       b PLUS ... PLUS NIL |
  REC [ (p : t ~ e), ... ] |         FIX [p, ...] : MKCROSS[[t, ...]]~[e, ...] |
  [ d, ... ] |                       d PLUS ... PLUS NIL |
  XX |                               xxxxxx | --Also recursive d maps into this?
  statements | simpleLoop |         -- See ¶ 2.2.3
  but                                -- See ¶ 2.2.4.
infixOp ::=
  X                                  MKCROSS
  PLUS
  THEN
literal ::= coreLiteral |
  digit digit ... |                  INT       -- Numeric literal, giving the decimal re
declaration ::=                      -- A d is not an e; a d must be before ~ or after LE
  p: t |                             MKDECL[ PATT p, t] |
  [(p: t), ... ]                     [p, ...]: MKCROSS[[t, ...]] separate names and types
binding ::=                          -- Only the [...] form is an e; a b must be written
  p ~ e |                            MKBINDP[PATT p, e] |
  d ~ e |                            MKBINDD[d, e]
pattern ::=                          -- Note: a pattern is not an e; it can appear only
                                           or after PATT in the kernel.
  n |                                -- PATT n=$n
  [p1, ...]                          -- PATT [p1, ...]=[PATT p1, ...]
primitive ::= corePrimitive |
  LOOKUP | LOOKUPC |                 -- Fill in types
  PLUS |
  IFPROC |
```

The precedence of operators in e is: (highest) [], Z, infixOps (all the same), BUT, IN (l
left associative.

2.2.2.1 Expression syntax

Most of this is straightforward sugar. LET adds the binding e_1 to ENV in evaluating e_2. Th
case for b, ... simply allows the [] which normally enclose a binding constructor to be
case; see ¶ 2.2.2.2. IF wraps e_2 and e_3 in l's so that they don't get evaluated; the IFPRO
chooses the one to evaluate and applies it.

The dot notation has three cases.

For a binding it just looks up n in the binding.

For a type it looks up n in the type's cluster.

For anything else, it looks up n in the cluster of De and applies the result to e. LOOPUPC primitive does something special if it finds a proc which takes more than on argument: it splits the proc into one which takes the first argument and returns a taking the remaining arguments. This ensures that if De.n is such a proc P, the exp e.n[a, b] will desugar into something equivalent to P[e, a, b].

The usual syntax for application is a proc e_1 followed by an explicit binding constructor of application may depend on the type of e_1, via the APPLY element of its type; for a proc by the standard apply operator Z, APPLY is the identity. If e_1 is followed by an group rat binding constructor, the argument is obtained by binding the group to the declaration wh domain.

Infix operators desugar straightforwardly into application; note that the choice of proc by the type of the first operand only. AND and OR are not ordinary infix operators, since evaluate no more than necessary; this is expressed by the desugaring into IF.

The remaining expression syntax is various constructors, described in the next section, imperative and exception features described in later sections.

2.2.2.2 Group, binding and declaration constructors

A bracketted sequence of expressions (e.g., [1, 2, 3]) denotes a flat group with its ele same order (e.g., MKPAIR[1, MKPAIR[2, MKPAIR[3, NIL]]]. Thus a group constructor is just like LIST function in Lisp. A pattern is a similar construct, except that it contains names w for the corresponding ATOM literals; PATT yields the group obtained by replacing each name the literal $n. After desugaring a pattern always appears after PATT and hence is always into an atom or a GROUP[ATOM].

Brackets are also used to delimit binding and declaration constructors. They are disting each other, and from group constructors, by the presence of ~ in each element of a bindi constructor, and : in each element of a declaration constructor. The elements of a bindi declaration constructor are sugar for applications of the MKDECL, MKBINDP and MKBINDD primit The constructor itself strings the resulting declarations into a big one using the PLUS o which is just like THEN except that it does not allow duplicate atoms; the motivation for allow the names and corresponding types or values to be written together, instead of fac primitives require. As a result, values made from constructors are always flat.

Note that these constructors do not nest, except that a d can be [(p: t), ...]. This is d~e form of binding; e.g., if DivRem returns two INTs, you can write [d: INT, r, INT]~DivRem instead of [d, r]: INTXINT~DivRem[...].

The REC binding constructor is sugar for FIX which exactly parallels the non-recursive one

2.2.3 Imperatives

These constructs are generally used together with non-functional procs.

```
statements ::= { e; ... }          IF (ISVOID[e]) AND ... THEN [] ELSE ERROR
                                   -- Ordering by non-prompt evaluation.
simpleLoop ::= SIMPLELOOP statements   LET REC [loop] ~ FN { statements; loop[] } IN loop[]
                                   -- Only an exception (such as EXIT) will terminate t
```

Each e in the statements must evaluate to VOID; this is to catch mistakes like writing x+
statement. The definition of AND ensures that the e's are evaluated left-to-right.

The simpleLoop is the standard way to express a loop in terms of recursion. You are supp
use an exception to get out of this loop; Cedar provides a number of convenient ways to
such as EXIT and RETURN.

2.2.4 Exceptions

An exception is treated as a special value returned from an application. The exception v
contains an exception code and an args value which may be of any type. When an applicatio
an exception value, it immediately abandons the application and returns the exception va
application is strict. There has to be some way to stop this, or the first exception wou
of the program. The HIDE primitive takes any value and returns a variant record of type H
turns:

- a normal value into the normal variant, with the value in its v field;
- an exception into the exception variant, with the code in its code field and the arg
  its args field.

UNHIDE takes a HEX value and returns the original unhidden value.

An exception code has the type EXCEPTION[T], where T is a declaration which is the type of
args; it is the domain of the exception, and (DEXCEPTION[T]).DOMAIN=T. An exception value i
constructed by the primitive

```
RAISE: [T:: TYPE, code: EXCEPTION[T], args: T]
```

Thus the args always has the type demanded by the code.

This is dressed up with the following syntax.

```
but ::= e BUT { butChoice; ... }
```

```
LET v(~HIDE[e]_IN  (
   IF ISTYPE[v(, HEX.normal] THEN UNHIDE[v()]
   ELSE IF ISTYPE[v(, HEX.exception] THEN
      LET h(~NARROW[v(, HEX.exception ] IN
        LET selector(~h(.code ENSBUTCHOICE
   ELSE ERROR )
```

```
butChoice ::= e1 => e2 |
  e1 , e1, !.. => e2 |
  ANY => e2
```

```
IF selector(=e1 THEN LET MKBINDD[De1.DOMAIN, h(.args] IN
IF (selector(=e1) OR ... THEN e2 |
IF TRUE THEN e2
```

A BUT expression evaluates e. If it is a normal value, that is the value of the BUT. If it
exception, each butChoice in turn gets a look at it. If one of them likes it, then it su
of the BUT; otherwise the exception is the value.

The e_1 in a butChoice must evaluate to an exception code. If there is just one, and it ma
in the exception, then args in the exception is bound to the domain of the code, and e_2 i
in that environment. If there is more than one, then e_2 is just evaluated in the current
An ANY butChoice matches any exception, but of course doesn't bind the arguments.

2.3 Doing computations

This section describes the basic mechanisms for doing computations, and the kinds of val
can be manipulated by Cedar programs.

2.3.1 Application

The basic mechanism for computing in Cedar is applying a proc to argument values. A proc
mapping

> from argument values and the state of the computation,
>
> to result values, and a new state of the computation.

The state is the values of all the variables.

A proc is implemented in one of two ways:

> By a primitive supplied as part of the language (whose inner workings are not open
> inspection).
>
> By a closure, which is the value of a l-expression whose body in turn consists of
> expression, which may contain further applications of procs to arguments, e.g., l
> x+3. When a closure is applied, the parameters declared after the l are bound to t
> arguments, and then the body after IN is evaluated in the new environment thus obta

In Cedar, each parameter value thus obtained is used to initialize a variable, which is
named by the parameter in the body. Thus the body can assign to the parameters. Use of t
feature is not recommended.

Note that when a l-expression is evaluated to obtain a closure its body is not evaluated
saved in the closure, to be evaluated when the closure is applied. Some constructs (IF, S
OR) are defined (see ¶ 2.2.2 and ¶ 3.8) by wrapping l-expressions around some arguments,
applying them only when certain conditions hold; e.g., IF b THEN f [x] ELSE g [y] evaluates
is TRUE and g [y] iff b is FALSE.

Application is denoted in programs by expressions of the form f [arg, arg, ...]. If the v
closure, this expression is evaluated by evaluating f and all the arg's, and then evaluat
of the closure with the formal parameters bound to the arguments (unless an exception va
up; see ¶ 2.6.2). Thus to evaluate (l [x: INT] IN x+3)[4]:

> evaluate the l-expression to obtain a closure;
>
> evaluate the argument 4 to obtain the number 4;
>
> evaluate x+3 with x bound to 4 to obtain the number 7.

The first two evaluations can be done in either order.

To evaluate a primitive application such as x+3, evaluate the arguments, and then invoke
primitive on those arguments to obtain the result and any state change. With a few excep
assignment and dereferencing or following references), primitives are functions and can
of as tables which enumerate a result value for each possible combination of arguments.
primitive can therefore be viewed as a simple table lookup using the arguments as the ta

Actually there may be one more step in an application. If an argument doesn't have the t
expected by the proc, the argument is coerced to the proc's domain type if possible. If
can be found, there is a type error. Coercion is discussed further in ¶ 2.6.1 and ¶ 4.13

Most procs take a binding as argument, in which the various parts of the argument are na
OpenFile: PROC[name: ROPE, mode: Files.Mode] takes a binding with two values named name and m
It might be applied like this: P[name~"Budget.memo", mode~$read]. If the names are missing

is a positional coercion which supplies them left-to-right, see ¶ 2.3.6. There is also a coercion that supplies missing parts of the binding; see ¶ 4.11.

If f is neither a primitive nor a closure, the meaning of applying it is defined by the its type; this case is discussed further in ¶ 4.4.

There are many ways of writing applications other than f [x]. In fact, many Cedar primiti be the values of expressions, and can only be applied by writing some other construct. T desugaring rules show how large parts of the Cedar syntax denote various special kinds o application. In each case, the meaning is defined by the standard meaning of application specific meaning of the primitives involved; see ¶ 4.1.

This is partly because of history, and partly because specialized syntax makes the program more readable. Futu evolution of the language will improve the situation.

Functions and order of evaluation

An expression is functional if

> its value does not depend on the state, but only on the values bound to its free v and
>
> evaluating it does not change the state.

As a consequence of this definition,

> Two identical functional expressions in the same scope will always have the same v

A proc is a function if every application of it is functional. It doesn’t matter when or times a function is applied; the order of evaluation doesn’t matter for functions. Thus functions can be thought of as mathematical functions for many purposes. Note that a con be regarded as an application of a function of no arguments.

Non-functional procs, on the other hand, are more complicated objects. Cedar makes no fo distinction, either in syntax or in the type system, between functions and procs. Howeve not define the order of evaluation in an expression, except that:

> all arguments are evaluated before a proc is applied;
>
> because of the desugaring of IF, SELECT, AND and OR into l-expression, the order of evaluation for these expressions is determined by the first rule;
>
> statements separated by semi-colons are evaluated in the order they are written.

As a consequence, two applications of non-functions should not be written in the same st unless they don’t affect each other; if this is done the effect of the program is unpred

An expression is guaranteed to be functional if it only applies functions; thus if f is non-functional proc, and x a variable, f [3] is functional and p[3] and p[x] may not be. f [x] may not be functional, because it is sugar for f [x.VALUEOF], and VALUEOF is not a fu value of a l-expression is a function if its body is functional. There are more complica guaranteeing that an expression is functional, just as for any other interesting propert

Because the values of variables constitute the state, it is only the existence of variab non-functional procs to exist. In particular, the VALUEOF proc which returns the value of is non-functional (because its result depends on the state), and the ASSIGN proc which cha value of a variable is non-functional (because it changes the state).

2.3.2 Values

A Cedar program manipulates values. Anything which can be denoted by a name or expressio
the program is a value. Thus numbers, arrays, variables, procedures, interfaces, and typ
values. In the kernel language, all values are treated uniformly, in the sense that each

- passed as an argument,
- bound to a name, or
- returned as a result.

These operations must work on all values so that application can be used as the basis fo
computation and l-expressions as the basis for program structure. In addition, each part
or type of value has its own primitive operations. Some of these (like assignment and eq
defined for most types. Others (like addition or subscripting) exist only for certain sp
(numbers or arrays). None of these operations, however, is fundamental to the language.
assignment or equality has the same status as any operation on an abstract type supplied
implementor; thus INTEGER.ASSIGN has the same status as IO.GetInt. In practice, of course, s
syntax is usually used to invoke these operations, and the implementations are not Cedar
open to inspection by the editor or debugger. A complete description of the primitives s
the language can be found in Chapter 4, organized by the type of the main operand. Table
an alphabetized index of these descriptions.

Restrictions: In current Cedar, however, there are restrictions on values which are type
or bindings: they can only be arguments or results of modules, and hence are first-class
in the modelling language, and not within a module. Also, declarations and bindings cann
constructed or bound to identifiers within a module. Unions are also restricted: they ca
appear inside records. Nonetheless, it is simplest to emphasize the uniform treatment of
and consider separately the restrictions on types, declarations, bindings and unions. Fu
will improve this situation.

Restriction: In current Cedar you can only use dot notation for some operations of built
the procs which access record fields, and others as noted in Table 4 5. As a substitute,
various syntactic forms which are sugar for dot notation: infix, prefix and postfix oper
functions, and funny applications. These desugarings are given in rules 20-24 of the Ce
grammar in ¶ 3.

2.3.3 Variables

Certain values, called variables, can contain other values. A variable containing a valu
has type VAR T. If the variable doesn’t allow the value to be changed, the type is READONL
is not the same as T, because there may be a VAR T value which is the same container. The
contained by a variable (usually called the value of the variable) can be changed by ass
value to the variable. The set of all variables accessible from the process array consti
of the computation; these are all the variables which can be reached from any process, a
variable which cannot be reached cannot affect the computation. Note that a variable val
container, which like all values is immutable; it may help to think of it as (the addres
of storage. The contents of a variable can be changed by assignment. Thus the value of a
can change, even though the value that is the variable is immutable.

A suitable abstract representation for a VAR T is a value of type [Get: []_T, Set: T_[]].
representation is not used in Cedar, but it clarifies the way in which variables fit int
system: VAR TgVAR U only if T and U have the same predicate, because the Get proc requires
TgU and the Set proc requires UgT. READONLY T corresponds to [Get: []_T] and a write-only
variable type would be [Set: T_[]].

There is a coercion (an automatically applied conversion; see ¶ 2.6.1) from VAR T to T, s
variable can be passed without fuss as an argument to a proc which expects a value.

Restriction: In current Cedar, variables generally cannot be passed as arguments or resu
exception is that an interface can declare a variable (called an exported variable) for
implementation supplies a value; this is normally written x: VAR INT in the interface, bu
historical reasons it is also possible to write just x: INT. Certain primitives (e.g., de
or POINTER) return variables, a variable can (indeed, must) be passed as the first argumer
ASSIGN, and a variable can be bound to a name by a declaration in a LET or block (LET x: IN
binds a VAR INT value to x). For the most part, however, a program which wants to handle
must do so at one remove, through procs or REFS (or, unsafely, POINTERS).

A variable is often represented by a block of storage; the bits in this block hold the r
of its value. All the built-in VAR types are represented in this way. A variable u overla
variable v if assigning to u can change the value of v. The primitive ASSIGN procs have th
that

if r and s are REFS, then r^ overlaps s^ iff r=s.

For any variables u and v with the same VAR type, u overlaps v iff u=v, provided that no
program has given overlapping blocks of storage to the two variables (if u and v have di
types, one might be contained in the other).

The role of variables in non-functional expressions is discussed in ¶ 2.3.1.

2.3.4 Groups

There is a basic mechanism for making a composite value out of several simpler ones. Suc
composite value is called a group, and the simpler values are its components or elements
x+1, "Hello"] denotes a group, with components 3, the value of x+1, and "Hello". The mai
of explicit groups is for passing arguments to procs without naming them (these are some
called positional arguments). This is done by binding the group to the declaration which
domain type of the proc; the result is a binding which is the argument the proc expects.
P: [x: INT, y: REAL]_[. . .], the application P [2, 3.14] is sugar for P [[x: INT, y: REAL]
which is equivalent to P [x~2, y~3.14].

A group has a type which is the cross type of its component types: if x has type T and y
then [x, y] has type TXU. Thus for syntactic types, D[e_1, e_2, ...]=De_1XDe_2X ... The X typ
constructor is associative, and type implication (¶ 2.4.2) extends to cross types elemen
are types, there is a coercion called MKCROSS from [T_1, T_2, ...] to T_1XT_2X ...; because of
explicit cross type is usually not needed.

Restriction: Current Cedar provides no way of making cross types except as domain and ra
of a proc type (or other transfer type); e.g., PROC [INT, REAL]_[BOOL, ATOM]. There are no p
taking groups except the group-to-binding coercions. Hence the only thing to do with a g
pass it to one of the built-in coercion procs by writing it as a proc argument, or to a
constructor as described in the next section. Current Cedar does not have X, but it does
MKCROSS to cross type coercion described in the last paragraph and illustrated in the exa

2.3.5 Bindings

A binding is a group in which each element has a name. Thus, it is an ordered set of [na
pairs. There are three main uses for a binding:

- As an argument in an application. Thus, if P is a proc with type PROC[i: INT, b: BOO
argument must be a binding such as [i~3, b~TRUE]. The application then looks like t
P[i~3, b~TRUE]. A binding argument is sometimes called a keyword argument list. See
next section for details.
- In a LET expression, to give names to values in the scope of the LET. Thus,

```
LET i~3, b~TRUE IN (IF b THEN i+5 ELSE 0)
```

has the value 8. Current Cedar doesn't have LET expressions, but a binding at the b
of a block has the same effect. See ¶ 2.5.4 on scopes for details.

- As a way of collecting and naming a set of related values. A value can be extracte
the set using dot notation. Thus if b is the binding [i~3, b~TRUE], the value of b.
current Cedar this only works for interfaces; see ¶ 3.3.4 and ¶ 4.14 for details.

A binding is usually denoted by a constructor, which takes the form

```
[i~3, b~TRUE]
```

or redundantly (if there are no coercions)

```
[i: INT~3, b: BOOL~TRUE]
```

in which the types are specified explicitly (but you can't write the second form as the
an application). See ¶ 2.5.5 on constructors for details.

2.3.6 Arguments

When a group or binding is bound to a declaration (d~v), there are various conversions ca
coercions which may be applied to the values. This usually happens when the arguments of
application are bound to the parameter declaration.

First, if v is a group rather than a binding, it is coerced to a binding by attaching th
to the elements of v in order. Thus in

```
[a: INT, b: REAL]~[2, 3.14]
```

the group constructor is coerced to [a~2, b~3.14].

Next, if v is shorter than d, elements of the form n~OMITTED are appended, where n is the
corresponding name from the declaration. Thus in

```
[a: INT, b: REAL]~[2]
```

the group constructor is coerced to [a~2, b~OMITTED].

Now the items of the binding are matched by name with the items of the declaration. Ther
error unless the names match exactly. The remaining coercions are done on individual ite
from the declaration and the corresponding n~v from the binding. If v has type t, all is
Otherwise, if there is a sequence of coercions from the type of v to t, these are applie
such sequence exists, there is an error. In particular, there is a coercion from OMITTED t
value for t, if any. Thus in

```
[a: INT_0, b: REAL_1.1]~[b~3.14]
```

the group constructor is coerced to [a~0, b~3.14], and in

```
[a: INT_0, b: REAL_1.1]~[]
```

it is coerced to [a~0, b~1.1]. Coercions are discussed in ¶ 2.6.1 and ¶ 4.13, defaulting

An important special case is constructors for record and array values. A record type has
construction proc; e.g.,

```
R: TYPE=RECORD[a: INT, b: REAL_0.]
```

has a proc R.CONS of type PROC[a: INT, b: REAL_0.]_[R]. Thus R.CONS[a~2, b~3.1416] constructs
record value. There is also a coercion from BINDING to the particular binding type RB whic
domain type of R.CONS, so that

```
r1: R_[a~2, b~3.1416]
```

is short for

```
r1: R_R.CONS[a~2, b~3.1416].
```

Composing the positional coercion from GROUP to RB with R.CONS makes

```
r1: R_[2, 3.1416]
```

also short for the previous line.

The same scheme works for arrays, but only an array indexed by an enumeration has a
corresponding binding which can be written; the elements of an array indexed by numbers
have names which can be written in a binding. However, the group constructor still works

2.4 The type system

This section describes the way in which types can be used to make assertions about the p
which the compiler can verify. It also discusses the role of types in organizing the nam
program.

2.4.1 Types

Types serve two independent but related functions in Cedar:

- A type contains an assertion about some property of a value, e.g., that it is a wh
between 0 and 10 represented in a single machine word. A value which has the prope
said to be of that type, or to have that type.

  The assertion part of a type is called its predicate. It is a function which accep
value (of any type) and returns TRUE iff the value satisfies the assertion.

- A type contains a collection of named procs (and perhaps other values) related in
useful way. Most often, the procs of type T take a value of type T as their first a
For example, INT has PLUS, TIMES and MINUS procs (usually written as infix or prefix
operators) which can be applied to INTs. The dot notation (see ¶ 2.4.4) makes it ea
refer to the procs in a type's collection.

  The collection part of a type is called its cluster. It is simply a binding. No ru
enforced about what kind of values are in the binding. However, the idea is that t
is an interface for manipulating values of the type (perhaps the main or even only
interface). As with any interface, a tasteful choice of names and values is import

The predicate and the cluster serve rather different purposes:

  The predicate provides the basis for type-checking (¶ 2.4.2). The most important f
type-checking is to guarantee the integrity of abstract data types; this is done w
predicates called marks (¶ 2.4.3).

  The cluster provides the basis for convenient naming of a large collection of proc
other values (¶ 2.4.4). Clusters are organized into a hierarchy of classes (¶ 2.4

Like everything else which can be named, a type is a value. Hence there is nothing speci
binding a type value to a name. If T is a type expression, the binding

```
U: TYPE~T
```

binds T's value to U. In the scope of U, T and U are completely interchangeable (provided
rebound). Furthermore, with two exceptions, all type expressions are functional: identic
expressions in the same scope denote the same type value. The exceptions are the record
enumeration type constructors, which make a distinct type each time they are used (by co
a new mark; see ¶ 2.4.3).

Caution: The AMTypes interface does distinguish between T and U as a convenience in debug
but it also provides a procedure GetCanonicalType for obtaining the type value in the sens
described.

Restriction: Current Cedar has a number of restrictions on the use of TYPE values, given

2.4.2 Type predicates and type-checking

Type predicates provide a way of making assertions in the program which can be checked
mechanically. These assertions take the form of declarations for the formal parameters o
general the checking must be done during execution. Thus, if the program says

```
a: ARRAY [0..10] OF INT_ALL[0];
i: INT_s.ReadInt;
s.PutF[ a[i] ];
```

there must be a check that i>0 and i<10 just before the expression a[i] is evaluated. Thi
a bounds check; if it fails there is an exception called Runtime.BoundsFault. Where did th
come from? Note that a[i] is short for Da.APPLY[a, i], and Da.APPLY is SUBSCRIPT, the subscri
procedure for ARRAY [0..10] OF INT. The type of SUBSCRIPT is PROC[array: ARRAY [0..10] OF INT,
index: [0..10]]_[VAR INT]. So when i is passed as the index argument, the declaration of SU
says it must have the type [0..10]. The predicate for this type is

```
l [x: ANY] IN HASMARK[x, INT] AND LET y: INT~x IN y>=0 AND y<=10.
```

Leaving the HASMARK term for later discussion, we see that the rest of the predicate is th
the bounds check.

The type system is designed, however, so that most assertions can be checked statically
by examining the text of the program without running it. Static checking has three obvio
advantages:

- It reports any errors after a single examination of the programming, leaving none
  kind) to be discovered later in Peoria.
- It introduce no cost in time or space for run-time checking.
- The compiler can take advantage of the assertions to generate better code.

Of course, there is a corresponding drawback: the assertions made by parameter declarati
be simple enough that the compiler can reliably prove or disprove them.

The proofs done for typechecking have exactly the same form as program correctness proof
on preconditions and postconditions. Consider a proc whose value is the l-expression

```
l [x: T]=>[y: U] IN e.
```

The domain declaration [x: T] is a precondition for the body e. This means that any appli
the proc must satisfy this condition. As a consequence, the body e can be analysed on th
assumption that the precondition holds, i.e., that x has type T. Similarly, the range de
is a postcondition for the body. This means that given the precondition, any evaluation
produce a value y which has type U. In summary, for the body we assume the precondition a
must establish the postcondition.

To make this hang together, each application must establish the precondition; this means
argument must have the domain type. In return, the application can assume the postcondit
means that the result of the application has the range type. Thus we have a linkage:

```
argumentgdomaingrangegresult
```

The result in turn will be the argument of another application. In this way the proof is
larger and larger expressions, and finally to the whole program. In summary:

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

These proofs require showing that an expression always has a particular type T. This is
observing that every expression has a unique syntactic type U, which is the type of ever
of that expression; e.g., an application always has the range type of its proc (see belo
detailed discussion of syntactic type). If every value of type U has type T, we are done
usefulness of type implication. One type implies another, TgU, iff (Ax) T[x]gU[x]. If two
are equal, each implies the other. However, there are many other useful cases of implica
instance, VAR INT implies READONLY INT. The type implications in current Cedar are given in
¶ 4.12.

Of course, not all arguments are applications. The kernel grammar gives the other possib
argument expressions, and we enumerate the proof rules for each:

- A literal is like a zero-argument proc: it has a known range (e.g., 3 has type INT,
  type CHAR).

A name has the type specified in its declaration or binding.

If there is only a declaration n: T (e.g., x: INT), it must be the domain dec of a l-expression, and we have already seen how to ensure that the n's value type T when the resulting proc is applied.

If there is a binding n: T~e for the name (e.g., x: INT~3), we must check tha type T.

A l-expression l [x: T]=>[y: U] IN e has the type [x: T]_[y: U]. This works for the discussed in the next paragraph.

A binding constructor [x~e, y~f] has the type of the corresponding declaration, [x: Df].

There is one more link in the chain. An application f [x] has an arbitrary expression fo necessarily a l-expression. The requirement is that f must have a proc type, say D_R; D i domain type and R the range type. Since the type of l D=>R IN e is D_R, satisfying the precondition D for the application is the same as satisfying the precondition D for the and similarly in reverse for the postcondition. The value of f may be a primitive rather closure obtained from a l-expression. In this case, the implementation of the primitive depend on the precondition and must still establish the postcondition, but since the imp cannot be examined (within the framework of Cedar) we can say nothing about how this is accomplished. Example: INT.PLUS, which is implemented by the machines 32-bit add instructi

In a proc type D_R, D and R may be declarations which provide names for the arguments and results. In general, the expression R may include names declared in D. The range type of application then depends on the argument values.

Restriction: In current Cedar only modules have such types; the type returned by an inte the interfaces exported by an implementation, may depend on the interface and implementa parameters.

As a by-product of the type-checking proof rules just given, a syntactic type is derived expression e in the program. It is denoted by De, and computed as follows:

for a name, the declared type;

for a literal, its type;

for an application, the range type (which may depend on the argument);

for a l the obvious proc type;

for a binding constructor, the declaration obtained by pairing the names with the types of the value expressions.

Typechecking ensures that whenever e is evaluated, the resulting value will have type De may have other types as well, i.e., it may satisfy other predicates). The main use of sy in connection with dot notation (see ¶ 2.4.4).

In order to carry out the proofs described above, the compiler must either compute the v types, including those denoted by complex expressions such as ARRAY [i..j] OF INT, or it m able to prove the equality of unevaluated type expressions. For the most part, current C requires the former approach; hence a type expression must have value which the compiler compute. Such a value is called static; the rules for static values are given in ¶ 3.9.1

2.4.3 Marks

By this point you may have thought of asking why the assertions provided by type predica
worth all this fuss. The reason is simple: they are the basis for authenticating values
type, so the implementation can be sure that it is working on properly formed values. Su
are the implementer of an abstraction, e.g., Table. You provide operations to Lookup a ke
table, to Insert a [key, value] pair, and to Enumerate the items in the table. A Table is
a REF to a record containing a sorted array a of items and an INT n which gives the numbe
items. Lookup is implemented by binary search. All three operations are programmed on the
assumption that elements 0 through n 1 of a are sorted, and that n is smaller than the si
array. They will not work properly if these assumptions are not satisfied, and indeed th
to subscript the array with an out-of-bounds index or to violate other requirements of t
abstractions they depend on.

Here is a lower level, but perhaps more dramatic example. The dereferencing operation ^
REAL returns a VAR REAL, which can, for instance, be assigned to, as in the program fragme

```
r: REF REAL_NEW[REAL_1.0];
. . .
r^ _ 3.14159
```

A REF REAL is represented by the address of a four-byte block of storage which holds a REA
the assignment to r^ stores the four bytes which represent 3.14159 into that block. If s
REF BOOL finds its way into r, the assignment will still store four bytes, since it doesn'
better. But the REF BOOL points to a two-byte block; the other two bytes that will be modi
belong to some unrelated variable, which will be clobbered without warning.

The second example is scarier because the consequences of the bug seem more unpredictabl
both cases, however, the fundamental problem is the same: even if the implementation is
the wrong thing happens because it is given an improper value to work on. Or to make the
point in different words, the implementation cannot be held responsible for bad results
its operations, if it has no control over the validity of the arguments it receives.

So that the implementation of an abstraction can take responsibility for correct operati
must be a way to authenticate a value of the abstract type. In Cedar this is done by pla
on the value; think of it as a little flag stuck into the value. The mark uniquely ident
abstract type, and authority to affix it is under the control of the implementation. A c
implementation will mark only values which have the properties needed for a representati
abstract value, and if no one else can affix the mark, the implementation can be sure th
value with the mark has the desired properties.

A mark can be thought of as an abbreviation for an assertion or type invariant which cha
proper abstract value, such as Table or REF REAL. Such an assertion can be quite complex.
Table example, it would say that the representation is a record of the proper form, that
the array size, and that the first n array elements are sorted. In the REF REAL example, i
that the address points to a block of storage such that at least the first four bytes do
other blocks. Such assertions are not easy to write down formally, and proving them is c
beyond the power of any existing program. So the abbreviations are not a mere convenienc
necessity.

A new mark can be created on demand by the primitive

```
CREATEMARK: PROC[Rep: TYPE, tag: UNIQUEID]_[m: MARK, Affix: [Rep]_[TYPEFROMMARK[m]] ]
```

The primitive HASMARK tests a value for the presence of a mark, so HASMARK[x, m] tests x fo
presence of the mark m. Affix adds the mark to a Rep value.

Restriction: MARK, UNIQUEID, CREATEMARK, HASMARK and TYPEFROMMARK are not accessible in current Cedar. Record and array type constructors provide some access to CREATEMARK, as described below. The ISTYPE primitive, also described below, is closely related to HASMARK.

With these facilities, it is easy to create a new abstract type. Choose its representati
obtain a new mark m. TYPEFROMMARK[m] with an appropriate cluster added is the new abstract
The implementation must use Affix to mark only values which satisfy the properties it dem

The type returned by TYPEFROMMARK[m] has the predicate

```
l [x: ANY]=>[BOOL] IN HASMARK[x, m]
```

and an empty cluster. Except for subranges and bound unions, all types in current Cedar
predicate of this form. The built-in types (INT, BOOL etc.) come with such predicates, and
in type constructor procs (ARRAY, RECORD etc.) obtain a mark from CREATEMARK. So that two
invocations of ARRAY [0..10] OF INT will produce the same type, ARRAY and most of the other
constructors use a canonical encoding of the constructor and its arguments for the UNIQUE
hence are functional. RECORD and ENUMERATION produce a different type each time they are
invoked, so they obtain fresh unique identifiers. Since the program cannot invoke CREATEMA
directly, we need not explain how to prevent forgery of UNIQUEIDS. Future versions of Cedar wil
this problem.

In current Cedar you make a new abstract type by declaring in as an opaque type in an in

```
T: TYPE[ANY]
```

This generates a new mark, and declares T to be a type which has that mark. You get such
by explicitly painting some other type, normally in an implementation which exports T to
interface which declared it:

```
T: PUBLIC TYPE~Interface.T PAINTED RECORD [...].
```

See ¶ 4.3.4 for more details.

The implementation actually stores a mark with each variable allocated by NEW. Such a var
be referenced by a REF, and in particular by a REF ANY value. The type of a REF ANY value c
tested at runtime using the primitive

```
ISTYPE: PROC[x: ANY, U: TYPE]_[BOOL]
```

If De is REF ANY and RT=REF T, then the value of ISTYPE[e, RT] is TRUE iff the predicate for
just tests for mark m, and x^ has the mark VAR m. ISTYPE is described in detail in ¶ 4.3.1
the WITH ... SELECT construct and the NARROW primitive, which are more powerful operations
up from ISTYPE.

For other values, there is no mark actually stored; instead, types must be computable st
using the methods described in the last section. The AMTypes interface, however, gives a
refer to any value in a uniform way, and to test its type at runtime.

There is only room for one mark on a variable, and this must encode all the marks that t
actually carries. We arrange for this by imposing a partial order on the marks, and requ

> The set of marks on a value must have a maximal element.
>
> Every mark smaller than the maximal one must be on the value.

With these rules, a single mark stored on the value is enough to code all the others.

In current Cedar, a value actually has only one mark, since:

> The only way to create a new mark is with the record or enumeration type construct
> by declaring an opaque type.
>
> When you paint a type T with the mark of an opaque type, T must be a record or
> enumeration type, and the opaque type mark replaces the mark it had before.

Note that VAR T, READONLY T and T are different types with different marks, although VAR
TgREADONLY T, and there is a coercion VALUEOF from either one to T.

2.4.4 Clusters and dot notation

It is convenient to associate with a type the procs supplied by its implementor for deal
values of the type. This is done by putting these procs into the type's cluster. The clu
binding which is part of the type value (the predicate is the other part). There are no
about what goes into the cluster. However, there is a special dot notation which makes i
to populate T's cluster with procs which take a T as their first argument. The usual eff
this: t.n is sugar for Dt.n[t], and t.n[other args] is sugar for Dt.n[t, other args].

For example, if t has type T, and a proc [T, INT]_[BOOL] is in T's cluster under the name
the proc can be applied by an expression like t.P [3], which is sugar for Dt.P [t, 3]. Th
looked up only in T's cluster, not in the current scope. If Q: [T]_[INT] is also in the c
be applied with t.Q, which is sugar for Dt.Q [t].

The general rule that makes this work is the following: t.n is sugar for LOOKUPC[Dt, $n][t
LOOKUPC[Dt, $n] is just Dt.n, except that if Dt.n is a proc that takes several arguments,
into a proc that takes the first argument and returns a proc taking the remaining ones.
LOOKUPC[Dt, $n][t] will be a proc taking the remaining arguments, and t.n[other args]=LOOKU
$n][t][other args] will be the same as Dt.n[t, other args].

Dot notation can also be used to obtain values from a binding or from the cluster of a t
any application: T.P would be the proc named P in the previous example. The possible case
notation in current Cedar are described in detail in ¶ 4.14.

Restriction: There is currently no way to explicitly construct clusters. The built-in ty
constructors have clusters; they are described in detail in ¶ 4. In addition, there is a
provide a cluster for an opaque or record type in an interface: every proc name in the i
put into the type's cluster. For a record, the procedures supplied by the record constru
in the cluster, and they win if there are name conflicts. There is one of these clusters
in each imported interface value; if a module imports more than one value of the same in
however, there are severe restrictions (see ¶ 3.3.3).

2.4.5 Declarations

A declaration is the type of a binding. Thus, the binding [x~3, y~3.14] has the type of
INT, y: REAL]. All the relationships among types, and between types and values, are carrie
elementwise to decls and bindings; the elements are matched up by name rather than by po
A decl itself simply has the type DECL.

A decl is made up of two parts: the names or pattern, and the types. The basic operation
making decls, MKDECL, takes a pattern and a type. Thus MKDECL[PATT[x, y], INTXREAL]=[x: INT,
REAL]. In general, a pattern is one of NIL, a simple name, or a pair of patterns, just lik
expression. Similarly, a type argument to MKDECL is one of NIL, a type, or a cross type. T
must decompose in a way which matches the pattern. Normally, as in Lisp, we deal only in
patterns, where the first element of a pattern is always a name. Such flat patterns are
denoted by constructors of the form [x, y, ...]. The reason for defining things in terms
that it makes it much simple to write down precise rules for the semantics, using struct
induction on the values.

The main use of a decl is to type-check a binding. The basic binding constructor is MKBIN
where d is a decl and e is matching group or binding. If e is a binding, then its structu
must match the structure and names of d, and each element of e must have the type demand
the corresponding component of d, after a possible coercion. Thus MKBINDD[[x: INT, y: REAL]
y~3.14]]=[x~3, y~3.14]. This may seem pointless, but it has two important uses:

> Such a binding is used to bind the argument of a proc to the domain declaration. E
> though the resulting binding is the same as the argument, the type-checking is ess

There may be coercions involved, so that the resulting binding is not the same. Co
on the component values are discussed in ¶ 2.6.1. There are also coercions on the
itself, which can default missing elements; these are discussed in ¶ 2.3.6.

If e is a group, it is first coerced to a binding by attaching the names from the decl, ¶ 2.3.6. Thus in MKBINDD[[x: INT, y: REAL], [3, 3.14]] the second argument is coerced to [x y~3.14], and things then proceed as before.

Bindings may also be used in LET expressions. Here the types are often redundant, and it to use the MKBINDP primitive to bind the value directly to a pattern. The syntactic type o is the decl whose type is the syntactic type of the value. Thus [x~3, y~3.14] is short fo MKBINDP[PATT[x, y], [3, 3.14]]; its syntactic type is MKDECL[[x, y], D[3, 3.14]]=MKDECL[[x, y INTXREAL]=[x: INT, y: REAL].

A decl D in a block is interpreted somewhat differently. It becomes the argument of the primitive, which turns the type of the decl D.T into the corresponding VAR type VT=D.T.MKVA allocates a new value v of type VT, and makes the binding MKBINDP[D.P, v] over the scope o block. Thus

```
  {x: INT; y: REAL; S}
```

becomes

```
  LET [x, y]~[VAR INT, VAR REAL].NEW IN S
```

Here D=[x: INT; y: REAL], VT=[VAR INT, VAR REAL], and v=[VAR INT, VAR REAL].NEW. Note that the types might have defaults, which are used to initialize the values as part of the NEW

Actually this is a bit oversimplified, since NEWFRAME has to separate the bindings in the the decls, construct the variable binding just described from the decl, and then combine binding from the block. Thus

```
  {x: INT; y: REAL; z~TRUE; S}
```

becomes

```
  LET [x, y, z]~([VAR INT, VAR REAL].NEW PLUS [TRUE]) IN S
```

or more readably

```
  LET x~VAR INT, y~VAR REAL, z~TRUE IN S
```

Anomaly: In Cedar the names in a block are introduced recursively, so that the d’s and b to each other. It is possible for a binding or type to refer to a value which has not ye initialized, with undefined results. See ¶ 3.4.1 for a further discussion of this point.

2.4.6 Classes

Another important use of a declaration is to characterize the cluster of a type. Since t just a binding, it is characterized by its type, which is a decl. When used for this pur called a class. See ¶ 4.1 for further discussion of classes, and an enumeration of the p of Cedar.

2.5 Programs

This section describes how meaning is assigned to kernel programs.

2.5.1 Structure of programs

A kernel program is an expression, which is either atomic (a name or literal), or is an which involves sub-expressions: the proc being applied, and the arguments. The concrete treats certain kinds of expressions specially: modules, blocks (which introduce new vari return no value), and statements (which return no value). All desugar into simple expres however, and are treated identically in the kernel.

2.5.2 Names

A name is a part of a program which usually serves to denote a value. There are two cont
which the occurrence of a name n denotes a value:

It may occur as an expression. Then n denotes the value bound to it in the scope i
the expression appears (see ¶ 2.5.4 for details).

It may occur after a dot, as in e.n. Then the expression e.n denotes the binding fo
supplied by e (see ¶ 2.4.4 and ¶ 4.14 for details):

the value bound to n in e, if e is a binding;

the value bound to n in the cluster of e, if e is a TYPE;

roughly (De).n[e] otherwise.

There are also two defining contexts for a name n (see ¶ 2.5.5 for details):

It may occur before a ~ in a binding constructor, as in n~e. The value of e is the
bound to n in the binding denoted by the constructor (see ¶ 2.3.5 for details).

It may occur before a : in a declaration constructor, as in n: t. The value of t is
n in the declaration denoted by the constructor (see ¶ 2.4.5 for details).

These constructors are usually recursive in Cedar; that is, the expression n elsewhere i
constructor denotes the value bound to n in that constructor; see ¶ 2.5.6 for details. I
they are non-recursive unless preceded by REC.

A name is not a value, but there are values of type ATOM which are related to names. An a
a print name which is a rope (an immutable sequence of CHARs). A name following a $ is an
literal; $n denotes the atom with print name n. Other properties of atoms are described

Caution: Current Cedar has several complications in its treatment of names:

•In an argBinding[27], n: e may be written instead of n~e. The syntactic context dist
this from a declaration, but this usage is not recommended.

An argBinding is not recursive: in {a~1; f[a~3, b~a+1]} b is bound to 2, not to 4.

The declaration in an import list is non-recursive: IMPORT M is short for IMPORT M:
and the second M denotes its binding in the currounding scope (i.e., the binding s
by the DIRECTORY). Inside the body of the module, of course, M denotes the imported
parameter.

Names which appear in an enumerationTC[54] are treated specially; see ¶ 4.7.1.1 for d

2.5.4 Scope

A scope is a region of the program in which all names retain the same meanings (note tha
names denote variables, which can change their values in the same scope, but each name c
to denote the same variable). In the kernel there are only three constructs which introd
scope, l, LET and REC. In current Cedar, these are sugared in a variety of ways: modules,
lists, proc bindings, blocks, exit labels, open, iterators, safeSelects and withSelects.
straightforward desugarings, however.

2.5.5 Constructors

The kernel has constructors, denoted [...], to make expressions which denote group, decl
binding values more readable. There is one flavor of constructor for each class:

A binding constructor is a list of binding elements (b in the kernel syntax) of th
or d~e. The presence of the ~ distinguishes it from the others. Here d is a decl e

(not a declaration), and p is a pattern, in which the names are being defined rath
evaluated.

A decl constructor is a list of decl elements (d in the syntax) of the form p: t. T
of the : without any ~ distinguishes it from the others. Again, p is a pattern.

A group constructor is a list of expressions. Note that decl and binding elements
expressions, although constructors are expressions.

Constructors are useful for making decls and bindings where the names are literal. This
normal case, and in fact the only case in current Cedar. If you want to make them out of
decls, for instance to bind an expression to a decl which is tha value of a name dn, you
constructor; [dn~e] would bind the value of e to the name dn, not to the decl which is it
You have to write the decl-constructing primitive directly: MKDECL[d, e].

The only kinds of constructor you can write in current Cedar are:

Decl constructors for proc domains and ranges, and for records and unions (fields[43]
syntax).

Binding constructors for arguments in an application, or as an expression alone if
or array value is needed (argBinding[27] in the syntax).

2.5.6 Recursion

In the kernel, you get recursive definition of names only if you write REC (or the unsuga
FIX) explicitly. In Cedar, on the other hand, decls and bindings are normally recursive,
argBindings and import lists.

The recursion is legal in a block or interface body (although anomalies are possible in
when names are used before they are defined; see ¶ 3.4.1). In fields it is illegal.

2.6 Conveniences

2.6.1 Coercion

A coercion is a proc which is automatically applied under some circumstances to map a va
one type T (called the source) to a value of another type U (called the dest), e.g. from
Coercions are obtained from the clusters of the types involved. The coercion mechanism a
new functionality, since the programmer could always write the applications himself, but
important in concealing some of the distinctions made by the type system when they are d
rather than helpful.

There is exactly one (desugared) context in which a coercion is applied: when an express
syntactic type T appears as an argument in an application which expects a value of type
means that there is a binding n: U~e. Since nearly all Cedar constructs are desugared to
coercions are widely applicable. The only (desugared) context in which there is no coerc
the first operand of dot, since in that case the cluster of the operand is used to inter
which is the second operand. Thus in the expression e.n, it is always De, the syntactic t
is used to look up n, regardless of the fact that this expression may appear as an argume
parameter of type U. If e is not a type or binding, however, then e.n desugars to P[e], w
P=LOOKUPC[De.Cluster, $n],and in the application of P, e does appear as an argument and can
coerced. Usually the cluster for T is set up with procs which take an argument of type T
domain of P is De and no coercion happens. This isn't always true, though; a subrange T o
inherits the arithmetic procs of INT, for example, and there is a coercion from T to INT w
is applies.

If TgU it is sometimes natural to think in terms of a coercion from T to U that is implem the identity function. In fact, implication is stronger than that, since it propagates t type constructors, including PROC, when coercion does not. Implication is discussed in ¶ ¶ 4.12.

There is a rather general rule for finding coercions from the clusters of types, though much practical importance in current Cedar, since there is no way for the user to define The rule goes like this. Each cluster may have a From item and a To item. T.From should c pairs with type [tag: ATOM, proc: T_U], and T.To of pairs with type [tag: ATOM, proc: U_T]. the tags for the moment. Consider the binding n: U~e, where De=T, and TgU is false. For e proc P in T.From or U.To we try n: U~P[e].

> If P: T_V is in T.From, it maps e to a value of type V, and we have to bind n: U~P[e VgU we are done; otherwise we can recurse on this sub-problem.
>
> If P: V_U is in U.To, we have to bind m: V~e. If TgV we are done; otherwise we can recurse on this sub-problem.

The whole process fails if no path of coercion procs takes us from T to U. The search can when all paths have been explored, and a particular path can be abandoned when a type ap on it for the second time. Since the search is done statically (by the compiler), and si of an attempt to coerece T to U can be cached, the time required for the search is not a

There are two obvious difficulties with this scheme. First, it may transform erroneous a into legal ones, but coercing an argument is ways not intended by the programmer. Second than one path of coercion procs may exist, and different paths may give different result second difficulty can be avoided, and the first minimized, if every coercion proc P is c it has a (partial) inverse, and $P^{-1}[P\{x\}]=x$ for all x in P.DOMAIN. This says that a coercio lose information, and that different paths give the same answer. Sometimes this is not f for the narrowing coercion from INT to [0..5). The following rule gives the builder of cl over proliferating coercions:

> If two procs on a coercion path have non-nil tags, they must have the same tag.

In general, coercions that don’t lose information can have NIL tags, and others should ha tags.

The coercions in current Cedar are described in ¶ 4.13. All have NIL tags, and none loses information except the subrange narrowing. Note that coercions extend componentwise to g and bindings.

2.6.2 Exceptions

The basic idea behind exceptions is to extend the value space, so that it includes not o values, but also a set of exception values. An exception value has the special property it appears in an application, it becomes the value of the application, so that it propag through the control stack of the program until it finally abecomes the value of the whol Of course this isn’t always what is wanted, so there is a special HIDE construct which is ordinary application, but takes its argument value, ordinary or exception, and bundles i record which is a normal value. Then ordinary code can be used to test for the exception appropriate action. This construct is sugared to give distinctive ways of catching an ex kernel with BUT (¶ 2.2.4), and in Cedar with ENABLE, EXITS and REPEAT (¶ 3.4.2). Cedar has t kinds of exception: GOTO labels and ERRORs, which must be raised and caught separately, an slightly different semantics.

The main point of this treatment is that it does not require continuations or any other explanation of how control is transferred to catch an exception. The view is that except simply a convenience feature; the same job could be done by returning a slightly larger each proc, with an appropriate status code.

An exception consists of a code and an optional argument value. The type of the code is
where T is the type of the argument which does with it. GOTO labels always have empty arg
The argument is a way of passing some information along in addition to the identity of t
exception.

A proper treatment of exceptions in the type system would require that each proc range i
the exceptions that can emerge from an application of the proc. In fact, this is not req
possible in current Cedar.

Cedar also has signals, which historically were viewed as a kind of exception but now ha
different interpretation, as a way of obtaining dynamic rather than static scoping for n
are discussed in ¶ 3.4.3.1.

2.6.3 Finalization

This subject is discussed in ¶ 3.4.3.1.

2.6.5 Concurrency

This subject is discussed in ¶ 4.10, where the Cedar facilities for writing concurrent p
given. Writing good concurrent programs, or even correct ones, is another matter, which
the scope of this manual to more than hint at. Unfortunately, an adequate reference is l

2.7 Miscellaneous

The different kinds of allocation are discussed in ¶ 4.5. Static values are defined in ¶

2.7.1 Pragmas

A pragma is a construct that does not change the meaning of the program, except perhaps
something illegal which was legal without the pragma. Its purpose is to affect the imple
generally by requesting optimization to favor one criterion over others. The pragmas in
Cedar are:

INLINE, which causes a proc body to be expanded inline when it is applied. See ¶ 3.5
details.

PACKED, which causes array components that fit in 8 or fewer bits to be packed, and
expense of more expensive code to access them.

CHECKED, which forbids application of unsafe procs in a block, and adds runtime chec
for some primitive procs which are otherwise unsafe (in particular, narrowing to a
and assigning a proc).

PRIVATE, which forbids access to items in an interface or instance except to modules
EXPORT (or SHARE) it.

MACHINE DEPENDENT, which allows positions of record fields and representation values
enumeration elements to be specified (strictly, it is the absence of MACHINE DEPENDEN
that is the pragma)

2.8 Relations among groups, types, declarations and bindings

Cedar has are four closely related basic ways of building product values from simple val
given precise meanings in ¶ 2.2.1 and ¶ 2.2.2):

a group is simply an n-tuple of values (see ¶ 2.3.4);

a X-type is the type of a group (if x: T and y: U then [x, y]: TXU) (see ¶ 2.4.5);

a binding is an n-tuple of [name, value] pairs (see ¶ 2.3.5);

a declaration is the type of a binding, an n-tuple of [name, type] pairs (see ¶ 2.

Figure 2 1 illustrates the relations among these kinds of objects. In current Cedar most
objects can be constructed and manipulated only as interfaces and instances. In the kern
modeller, all of them are first-class citizens. The primitives which go between them are
¶ 2.2.

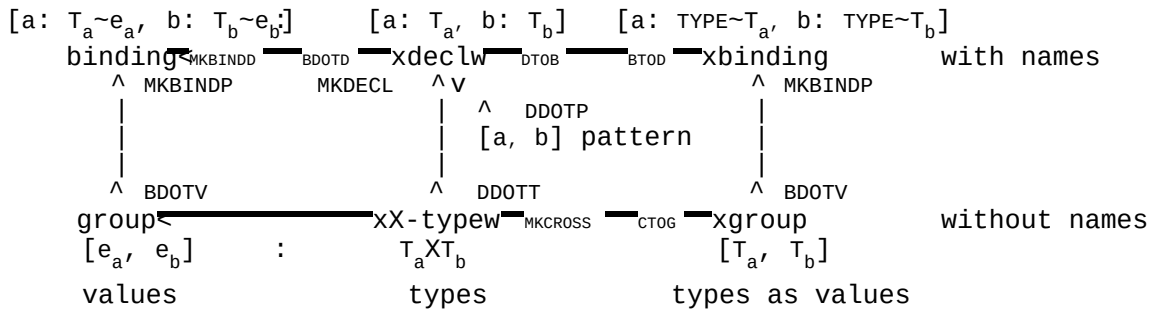

Figure 2 1: Relations among groups, types, bindings and decls

2.9 Incompatibilities with current Cedar

Most of the syntax is current Cedar is an extension (or sometimes a restriction) of kern
There are a few things that have different meanings in the kernel, however, and these ar
sources of confusion:

Type expressions in Cedar do not have the same syntax as ordinary expressions and
appear in the same contexts, for the following reasons:

The use of adjectives for variants (red Node).

The use of _ for specifying a default value for a type vs its use for assign

The use of {} for enumeration types vs its use for a block.

In addition to writing n: t~e for a binding, you can also write n: t=e (in a module
or block) and n: e (in an argBinding). The most unfortunate consequence is that a
argBinding can look like a kernel decl constructor!

Target type overloading for enumeration identifiers (red instead of Color.red or $re
union constructors ([rator~$plus, rands~binary[...]]) is incompatible with the kerne
rules for the meaning of names.

It is now possible to avoid all the conflicting constructs except the relatively harmles
defaults, {} for enumeration, and union constructors.

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 ¶ 2. The desugaring, together with the definitions of the ker primitives used in it, are the authority for the meaning; the informal explanation is ju reading pleasure. However, paragraphs beginning Anomaly or REstriction document properti Cedar not captured in the desugaring. The primitive procs and types of Cedar are specifi

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. The Cedar Ma longer examples which do something interesting, and also illustrate the use of the stand packages.

There are several summaries which may be useful as references:

A two-page summary of all the syntax, desugaring and examples in this chapter (CLRMSumm.press).

A one-page summary of the full syntax (CLRMFullGram.press).

A shorter and less cluttered summary of the syntax for the safe language; it also number of constructs which are obsolete or intended only for efficiency hacking (CLRMSafeGram.press).

The chapter begins with a description of the notation (¶ 3.1) The next sections deal sys with the rules of the grammar, explaining peculiarities of the syntax and giving the sem

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

¶ 3.3, rules 1-5: Modules.

¶ 3.4, rules 6-10: Blocks, OPEN, ENABLE, EXITS.

¶ 3.5, rules 11-13: Declarations and bindings.

¶ 3.6, rules 14-18: Statements.

¶ 3.7, rules 19-27: Expressions.

¶ 3.8, rules 28-35: Conditional constructs: IF and SELECT.

¶ 3.9 treats various miscellaneous topics. ¶ 4 deals with the syntax and semantics of ty

The order of the grammar rules is:
module, block, declaration, statement,
expression, conditional
type,
name, literal
and top-down within these.

3.1 Notation

This section describes the notation used in the grammar, the desugaring, and the comment
this chapter.

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 a
ELSE, IN, or OR, or it may be absent. If the separator is ";", a trailing ";" is option
item; !.. is just like item; ... but there is at least one occurrence.
A terminal is a punctuation character other than bold ()?|, or an ampersand, or a word in
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.
Special symbols mark constructs with special properties:
,=unsafe;
•=obsolete;
„=machine-dependent;
µ=efficiency hack.

The grammar is written so that a non-terminal never expands to the empty string. When an
of a rule is optional, that is always indicated explicitly by "?" or "..." .

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]
e=expression[19]
n=name[56] (identifier)
s=statement[14]
t=type[36]
I'm afraid this means that you must learn the meaning of these six abbreviations in orde
the grammar.

With the exception of these abbreviated non-terminals, each use of a non-terminal is cro
referenced with a small superscript number[59], unless the non-terminal is defined in one o
few rules. If a non-terminal (other than e, t or n) is used in more than one rule, then
that use it are listed in a comment after its definition.

Except for the entries in Table 3 1, a terminal symbol appears in only one rule. These d
do not lead to ambiguity. In most cases they are harmless, since the symbol has essentia
meaning in each case, and the rules are separate only for greater readability, to highli
use of a construct, or for historical reasons. In some cases, however, the symbol has qu
meanings in different rules. These are marked on the left as follows

- The rules marked with • are obsolete and should be avoided.

6 In the rules marked with * the symbol has a different meaning than in the others,
confusion is quite possible. The programmer should bear these cases in mind.

0 In the rules marked with * the symbol has a different meaning than in the others,
context is sufficiently clear that confusion is unlikely.

A superscript^{xn} indicates that the terminal is repeated n times in that rule.

	Symbols	Rules	Explanation
0	( )	19, 25, *51.1, *54	expr, subrange, *position in record or enumeration
	[ ]	19, 25, •37, 43, 51	constructor/built-in/funnyAppl, subrange, application, •typeName, fields, mdFields
0	{ }	2, 6, 8, 13, *54	interface body, block, enable, machine code, *enumerationTC
	,	2, 3, 6, 7, 9, 17, 27, 29, 30, 32, 34, 35, 43, 51, 52	see note in ¶ 3.2.
	;	6, 8, 10, 17, 27, 30, 33, 35	see note in ¶ 3.2.
0	:	1, 2, 3, 5, •7, 11, 13, 18, •27, 33, •34, •51, *51.1, 53	introducing names with types, except *51.1=position, •=open, •27=argBinding •34=withSelect
	.	19, 37	dot notation for e is repeated for types
0	..	25x4, *51.1	subrange, *position$_0$
*	21, *53	infixOp, *tag	
	+	21, 58	infixOp, exponent
		20, 21, 58	prefixOp, infixOp, exponent
•	=	•13, 22	•binding, infixOp
	=>	6, 9, 17, 31, 33, 35, 52	exits, enable, repeat, select choicesx4, unionTC
6	_	14, 16, 18, 21, *55	e_STATE, iterator, e, *defaultTC
0	~	2, 3, 13, 20, *22, *27	interface,implementation,b,argBinding,*unaryOp,*relOp
	~~	7, 34	open, withSelect
6	ANY	*9, 40, 43	*enable, variableTC, fields
6	CODE	*13, 23	*new exception, convert t to e
	ENDCASE	31, 52	select endChoice, unionTC
0	ERROR	*19, *24, 41.1	*expression, *funnyAppl, transferTC
	IMPORTS	2, 3	interface and implementation
	IN	18, 22	iterator, relOp
	LONG	38x2, 45.1, 48	cardinal/unspecified, pointer, descriptor
	NOT	20, 22	prefixOp, relOp
•	NULL	14, •27, •52, •55	statement, •argBinding, •unionTC, •defaultTC
	PACKED	44, 45	array, sequence
	SELECT FROM	29, 32, 34, 52	selectx3, unionTC
	SHARES	2, 3	interface and implementation
0	SIGNAL	*24, 41.1	*funnyAppl, transferTC
	TRASH	27x2, 55x2	argBinding, defaultTC
	TRUSTED	6, 13	block and machine code
6	USING	1, *5	directory, *locks
0	WITH	*32, 34	*safeSelect, withSelect

Table 3 1: Terminal symbols appearing in more than one rule

3.1.2 Notation for desugaring

The right-hand column is desugaring into the Cedar kernel language, or in a few cases in comments describing the meaning in English. This is a purely textual transformation; i.e on the text of the program, not on the values. The rewriting is done one rule at a time; step of rewriting involves elements from exactly one rule. The desugaring is specified b informal but straightforward rewriting rules, in which:

An occurrence of a non-terminal (written in bold) denotes the text produced by tha terminal in the grammar rule.

A | reflects a corresponding alternation in the grammar rule, ? reflects a corresp optional item in the grammar rule, and (bold parentheses) are for grouping as in a 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 that n 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 e (expression) and t (type) in the desugaring is implicitly parenthe the desugared program parses as the rewriting rule indicates. To reduce clutter, these p are not written in the desugaring rules.

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

Examples: the following rule for subranges:

```
subrange ::= ( typeName | ) (
  ( [ e1 .. e2 ] | ) [ e1 .. e2 ]      (typeName | INT).MKSUBRANGE ( [e1, ( e2 | e2.PRED ) ] )
  ( (e1 .. e2] | [.. (e2 ))                                        [e1.SUCC, ( e2 | e2.PRED )
```

generates these desugarings

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

Names introduced in the desugaring are written with one or more trailing prime ("(") cha 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.

3.1.3 Notation for the commentary

Each section of the commentary begins with grammar rules, desugaring and examples for pa the language. It continues with text which explains the meaning of the constructs. Gener meaning is fairly clear from the desugaring, and this text is short. For blocks and espe modules, however, there are many non-obvious implications of the desugaring, and a numbe restrictions; these constructs have a lot of explanatory text.

Some kinds of information are put into specially marked paragraphs, which begin with one following italicized words:

> Anomaly: the meaning of this Cedar construct is not explained by desugaring into t kernel, but by the special rule given here.
>
> Caution: here is an implication of the definition which might surprise you.
>
> Performance: facts about the time or space required by some construct.
>
> Representation: the values of a data type are represented in terms of other types
>
> Restriction: a construct is not fully general, and will cause a static error unles additional conditions stated here are satisfied.
>
> Style: advice about good Cedar style.

Symbols written in SANS-SERIF SMALL CAPITALS are in the kernel but not in current Cedar. The superscript notation used to cross-reference non-terminals in the grammar is also used i examples, usually to point to a rule whose example introduces a name.

3.2 The lexical structure of programs

```
56 Name ::= letter (letter | digit)... -- But not one of the reserved words in Table 3 2.
57 literal ::= num | num ?( B | D ) ?num ) |    -- INT literal, decimal if radix omitted or D, octal
  digit (digit | A | B | C | D | E | F) ... | num H ?num |   -- INT literal in hex; must start with digit. |
  ?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 | e) ? (+ | - ) num          -- Optionally signed decimal exponent.
59 num ::= digit !..
60 extendedChar ::= space | \ extension | anyCharNot'"Or\
61 extension ::= digit1 digit2 digit3 |-- The character with code digit1 digit2 digit3 B. |
            N | R | T | n | B |                 -- CR, '\015 | TAB, '\011 | BACKSPACE, '\010 |
            F | L | ' | " | \                   -- FORMFEED, '\014 | LINEFEED, '\012 | ' | " | \
```

Examples

```
m, x1, x59y, longNameWithSeveralWords: INT;
n: INT~1+12D+2B3+2000B                  -- = 1+12+1024+1024
       +1H+0FFH;                        --  +1+255
r1: REAL~0.1+.1+1.0E 1                  -- = 0.1+0.1+0.1
            +1E 1;                      --  +0.1
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 ¶ 4, as part treatment of type T.
- A name[56], not one of the reserved words in Table 3 2. Note that case is significant names.
- A reserved word, which is a string of uppercase letters that appears in the list o words in Table 3 2. A reserved word may not be used as a name, except in an ATOM literal.
- A punctuation symbol: any printing character not a letter or digit, and not part o the two-character sequences below. The legal punctuation symbols in programs are:

  ```
  ! @ # $ ~ *   + = | ( ) { } [ ] _ ^ ; : ' " , . < > /
  ```

  The following ASCII characters are not legal punctuation symbols (and must no appear in a program except in an extendedChar[60]):

  ```
  % & \ ?
  ```

  Note that Cedar uses a variant of ASCII which includes the characters ← (instead of _) and ↑ (instead of the circumflex ^). Note also that the character written - here is the ASCII minus character, code 55B, and not any of the various dash or typographer's minus characters with other codes, which are not standard ASCII set.

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

~=	not equal[22]
<=	less than or equal[22]
~<	not less than[22]
>=	greater than or equal[22]
~>	not greater than[22]
=>	chooses[8, 17, 30, 31, 33, 35, 52]
..	subrange constructor[25, 51.1]
~~	bind by name[6, 34]

ABS	EXIT	MONITORED	RETURNS
ALL	EXITS	NARROW	SAFE
AND	EXPORTS	NEW	SELECT
ANY	FINISHED	NIL	SEQUENCE
ARRAY	FIRST	NOT	SHARES
ATOM	FOR	NOTIFY	SIGNAL
BASE	FORK	NULL	SIZE
BEGIN	FRAME	OF	START
BOOL	FREE	OPEN	STATE
BOOLEAN	FROM	OR	STOP
BROADCAST	GO	ORDERED	STRING
CARDINAL	GOTO	OVERLAID	SUCC
CEDAR	IF	PACKED	TEXT
CHAR	IMPORTS	PAINTED	THEN
CHARACTER	IN	POINTER	THROUGH
CHECKED	INLINE	PORT	TO
CODE	INT	PRED	TRANSFER
COMPUTED	INTEGER	PRIVATE	TRASH
CONS	INTERNAL	PROC	TRUSTED
CONTINUE	ISTYPE	PROCEDURE	TYPE
DECREASING	JOIN	PROCESS	UNCHECKED
DEFINITIONS	LAST	PROGRAM	UNCOUNTED
DEPENDENT	LENGTH	PUBLIC	UNTIL
DESCRIPTOR	LIST	READONLY	USING
DIRECTORY	LOCKS	RECORD	WAIT
DO	LONG	REF	WHILE
ELSE	LOOP	REJECT	WITH
ENABLE	LOOPHOLE	RELATIVE	ZONE
END	MACHINE	REPEAT	
ENDCASE	MAX	RESTART	
ENDLOOP	MIN	RESUME	
ENTRY	MOD	RETRY	
ERROR	MONITOR	RETURN	

Table 3 2: Reserved words and predefined names

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 number of initial whitespace characters or comments.

The whitespace characters are space, tab, and carriage return. A Tioga node bounda
also treated as a whitespace character.

A comment is one of:

A sequence of characters beginning with --, not containing -- or a carriage
and ending either with -- or with a carriage return.

A Tioga node with the comment property.

Note that whitespace and comments are not tokens, but may appear before or after any tok
are token delimiters, and hence cannot appear in the middle of a token. Whitespace and c
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 3 2) and most names with predefined meanings (Table 4 5) are up entirely of upper case letters. They should not be rebound by the program; in some bu cases the compiler forbids their rebinding. All are at least three characters long excep following:

DO GO IF IN OF OR TO.

A note on lists of items and their separators:

Semi-colons are used to separate declarations, bindings and statements in a body[10], separate choices in a select statement[29, 32, 34] or in an exits[6, 17] or enable[8, 27.1].

Commas are used to separate declarations in fields[43, 51] (i.e., in a proc domain or recordTC or a unionTC), bindings in an application[27] or an open[7], choices in a sele expression[29, 32, 34] or in a unionTC[52], expressions in a choice[6, 9, 17, 30, 35, 52], items i exports or shares lists[2, 3].

In general these sequences may be empty, and an extra separator at the end is harmless w is some kind of closing bracket, except when the sequence is bracketed with [].

The braces which delimit a block[6], interface body[2], choices in an enable[8], or MACHINE CODE may be replaced by BEGIN and END reserved words. BEGIN replaces "{" and END replaces "}". I one brace is replaced, its matching partner must also be replaced. The braces delimiting enumTC[54] may not be replaced by BEGIN and 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 [LET n_u (n_d~REST RICT)[n_d, [$n_u; ... ] ] ), ...
                        ( interface | implementation )  implementation )
2 interface ::= n_m, !.. : ?CEDAR DEFINITIONS r(~[ n_m: INTERFACETYPE[[ n_m, ...]] ] IN (imports | l
  ?locks (imports | ) ?•(SHARES n_s, ...)-- SHARES allows access to PRIVATE names in n_s.
  ~ ?•access[12] { ?open[7] (d | b); !.. }LET REC n_m~open [ ?(l(~locks, ) (d | b), ... ]  IN MKIN

3 implementation ::= n_m : ?CEDAR      LET r(~REC [(n_e: n_e)  , ... , FRAME: TYPE n_m ,
                                                                  n_m: FRAME,CONTROL: PROGRAM]
  ?safety ( PROGRAM ?drType[42]  |     IN (imports | l=>r() IN
                    MONITOR ?drType[42]  (( || (locks)OCK:MONITORLOCK IN LET_l(IN( LOCK)IN | ))
  (imports | )                         LET b(~NEWPROGINSTANCE[block].UNCONS IN
  ?(EXPORTS n_e, ...)                       [ (n_e~BINDDFROM[n_e, b() ), ... , FRAME~MK]INTTYPE[blo
  ?•(SHARES n_s, ...)                         n_m~b( , CONTROL~b(.n_m] where the block body is
  ~ ?•access[12] block .                    [( | ( | l(~locks,)) (d | b), ... , n_m: PROGRAM
3.1imports ::= IMPORTS ( (n_iv : | ) n_it ), [(.n_it-:n_it), 3...]=>r( IN LET (n_iv | n_it)~ n_it PLUS n_it.
4 safety ::= SAFE | UNSAFE  --In 3, 41.
5 locks ::= LOCKS e ?( USING n_u: t)     l ?( [n_u : t] ) IN e
```

Examples

```
DIRECTORY                                   -- For BufferImpl below.
  Rope: TYPE USING [ROPE, Compare],         -- There should always be a USING clause
  CIFS: TYPE USING [OpenFile,Error,Open,ReadProc],
                                            --   unless most of the interface is used
  IO: TYPE IOStream,
  Buffer: TYPE;                             --   or it is exported.

Buffer: DEFINITIONS ~
  Handle: TYPE~REF BufferObject;
  BufferObject: TYPE=Rope.ROPE
  New: PROC RETURNS[h: Handle];
  Get: PROC[h: Handle] RETURNS[BufferObject];
  Put: PROC[h: Handle, o: BufferObject];

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: 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 secrets or 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
The others are the subject of this section.

There are two kinds of modules: interface modules (written with DEFINITIONS) and implementa
modules (written with PROGRAM or MONITOR). They have the same header (except that interface
have no EXPORTS list); it defines the parameters and results of the module viewed as a pro
and specifies the name n_m of the module. The bodies (following the ~) are different. Tabl
summarizes the structure of modules and their types; it omits a number of details which
in rules 1-3 and explained in the text.

Example	Module	Module type	Result	Result type
`DIRECTORY Rope, IO;` `  Match: DEFINITIONS~{...}`	Interface module	`[Rope: TYPE Rope, IO: TYPE IO]` `  _[TYPE Match]`	Interface	`TYPE Match`

`DIRECTORY Match, Rope, IO;`
`  MatchImpl: PROGRAM`
`  IMPORTS R: Rope, I: IO`
`  EXPORTS Match~{...}`
	Implementation module	`[Match: TYPE Match,`
		`  IO: TYPE IO, Rope: TYPE Rope,`
		`  R: Rope, I: IO]_[Match]`
			Exported instance	Match

Table 3 3: Interface and implementation modules

The ensuing sub-sections deal in turn with:

¶ 3.3.1: Modules as procedures, and the interface or instance values obtained by a
them.

¶ 3.3.2: How modules are applied.

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

¶ 3.3.4: Interface module bodies and interfaces.

¶ 3.3.5: Implementation module bodies; the EXPORTS list.

¶ 3.3.6: SHARES and access[12].

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

CEDAR in ¶ 3.4.4.

MONITOR and LOCKS in ¶ 4.10.

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 compiler's command line),

Instances of interfaces, declared in the IMPORTS list. These arguments are also supp
the 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.

For an implementation module, a binding whose values are instances:

one interface instance for each interface it exports;

one for the program instance, also called a global frame;

one for the program proc derived from the module body (¶ 3.3.2.1), called
CONTROL.

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

An interface (sometimes called an interface type) is a type, as the latter name suggests
declaration (obtained from the declarations which constitute the module body), with an e
cluster that includes all the bindings in the module body that don't use declared names
the example, the Buffer interface (obtained by applying the Buffer module to the argument
in its DIRECTORY) has declarations for New, Get, and Put, and its cluster includes values f
and BufferObject.

An interface instance is a value whose type is an interface; such values are the results
instantiating implementation modules. In the example, BufferImpl returns (exports) an ins
Buffer.

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 an implementation (PROGRAM or MONITOR) module body,
just like any proc frame (¶ 3.3.5). Normally code outside the module does not deal with
directly, but only with the exported interface instances. In the example, BufferImpl expo
program instance for the module and a CONTROL proc.

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); hence it 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 it returns (since its arguments need not be written explicitly);
the implementation module which exports the interface;
its instance (again, since its 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.

In the example at the start of this section, BufferImpl is an implementation module with parameters:

Four interface parameters, declared in the DIRECTORY: Rope, CIFS, IO and Buffer.

Three instance parameters, declared in the IMPORTS: Files (of type CIFS), IO (of type and Rope (of type Rope). Since the instance parameters are declared in an inner sco instance Rope is the one visible in the module body; the interface Rope is visible header. The same is true for IO, but both the interface CIFS and the instance Files visible in the body.

When BufferImpl is compiled, the four interface parameters must be supplied, in the form (compiled) interface modules named Rope, CIFS, IO and Buffer. When BufferImpl is instantiat (normally by loading it), the three instance parameters must be supplied, i.e. there mus instantiated implementation modules which export the Rope, CIFS, and IO interfaces. Norma there will be one of each, and the entire program will consist of eight modules:

the interface modules Rope, CIFS, IO and Buffer;

implementation modules normally named RopeImpl, CIFSImpl, IOImpl and BufferImpl, each exporting an instance of the corresponding interface

The instantiated BufferImpl exports an instance of Buffer, which can then be used as a par some other module.

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 interfaces) 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 will 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 error possible in the second stage is supplying an unsuitable argume

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.

When there are multiple instances of the same module with the same interface param
they automatically get the same code.

We've always done it that way.

3.3.2.1 Initializing a program instance

The statements in the body of an implementation module form the body of a proc called th
program procedure. The function of this proc is to initialize an instance of the module.
instance PI may be uninitialized, because no code in the module is executed when the ins
made. It is the job of the program proc PP(to initialize PI, perhaps using the PROGRAM ar
if there are any. Until PP(has been called, PI is not in a good state. It would be bette
the PROGRAM arguments along with the imported instances, and call PP(as part of making PI
that PI is never accessible in its uninitialized state. But it isn't done that way; henc
programmer must ensure that PP(is called before any use is made of PI. To confuse things
not an ordinary procedure but a PROGRAM, and it must be called using the START construct (s
¶ 4.4.1). Note that in addition to the statements of the module body, PP(also contains
specific initialization code for any variables or non-static values in the instance; e.g
value of x will not be 3 until after PP(has been called.

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(ta
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 proces
the module while PP(is running, it will not wait, but will run concurrently with PP(. Th
unlikely to be right. It is unwise to rely on a start trap to initialize a monitor modul
explicitly with START.

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 in
instance, and may still be called automatically by a start trap.

The program proc is bound to the name CONTROL in the result of an implementation module it
type is PROGRAM[] RETURNS[] (otherwise the proc Runtime.ReportStartFault is bound to CONTROL).
This allows the modeller (and binder) to get access to PP so as to control the order in w
modules are started.

3.3.3 Parameters to modules: DIRECTORY and IMPORTS

The interface parameters of a module are declared in the DIRECTORY. An interface I has typ
n, where n is any one of the names given before DEFINITIONS in the header of the interface
that produced I. The INTERFACETYPE primitive in the desugaring takes a list of names and re
type which implies TYPE n for each n in the list. The reason for allowing several names i
conversion of an interface from one name to another; both names can continue in use for
The use of these names provides a clumsy check that the proper interface is supplied as
argument. DIRECTORY n: TYPE and DIRECTORY n are both short for DIRECTORY n: TYPE n.

An interface is a type which can only be used:

Before a dot (¶ 4.14), to obtain a value from its cluster, which simply consists o 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] RETURNS [BOOL]~INLINE {
  IF Length[r1]~=Length[r2] THEN ... }
```

A call of Rope.Compare in BufferImpl is perfectly all right, even though Rope.Length in Buff would be an 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 (e.g., x~3) is just the value that the interface binds to the name (in this case, the va 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 I.x has the meaning (namely 3) whether I is the interface or the instance.

Anomaly: Nmaes bound to inline procs in an interface do not appear in the interface bind only in an instance. This somewhat duboius rule ensures that clients won't have to add imports lists if a proc stops being an inline.

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 (types constants) bound in its interface parameters, and does not need values for any names declared there (procs and variables). If an interface module does have IMPORTS, however, and there is more than one instance of any import interface around, then there is a restriction on the argument values. Suppose that Int1 imports Int2, and that module P imports Int1. Then Int1 may only import one instance of Int2, and if P also imports Int2, the principal of Int2 in P must be the same as the value of Int2 imported by the Int1 imported by P. For example, with

```
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=Int2V.

3.3.4 Interface module bodies

The body of an interface module I is a collection of bindings (e.g., x: INT~3) and declar y: VAR INT or P: PROC[a: INT] RETURNS [REAL]).

Restriction: Only certain things may follow the ~ in one of the bindings[13]:

If it is an expression, it must be static (¶ 3.9.1).

If it is a block (providing the body of a proc), it must be INLINE (because there is
place to put the compiled code).

It may not be CODE.

The result of applying an interface module is an interface (¶ 3.3.2), which is a type I
applying the primitive MKINTTYPE to the d's and b's of the body. This type is simply the d
obtained by collecting the declarations in the body, with a cluster which is extended to
the bindings of the body. However, MKINTTYPE omits any inline proc bindings from the type'
cluster, instead leaving the proc declarations in I. It puts an extra item BINDING in I's
the inline procs in it. When an instance Inst of I is imported, the binding actually impo
PLUS I.BINDING. This slightly dubious arrangement ensures that clients don't have to change
lists if a proc stops being inline. This policy is not extended to other items, however,
they might change from being bound in the interface to being interface variables.

The interface returned by
```
  Red, Blue, Green: DEFINITIONS~...
```
has the types TYPE Red, TYPE Blue, and TYPE Green.

The types and expressions in declarations and bindings may refer to other names in the b
usual, but they may not refer to names introduced in the declaration, except that:

Any declared name may be used

in the body of an INLINE, or

after a "_" in a defaultTC[55] in the fields[43] of a transferTC[41] which is the ty
decl in the interface's body.

A declared (opaque) type may be used anywhere.

For example, if an interface contains
```
I: DEFINITIONS~
  x: INT~3;
  y: VAR INT;
  T: TYPE[ANY]
```
then the following may also appear in the interface:
```
  xx: INT~x+1;
  P: PROC RETURNS[INT]~INLINE {RETURN[x+y]};
  Q: PROC [INT_y];
  V: TYPE~RECORD[f: REF T, g: U]
```
but the following are illegal:
```
  xy: INT~y+1;
  U: TYPE~INT_y;
  W: TYPE~ARRAY [0..y] OF INT;
```

The values of the bindings can be accessed directly by dot notation in any scope in whic
interface value is accessible. Thus if the value of the previous interface module is bou
because J: TYPE I appeared in the DIRECTORY, then J.x is equal to 3. The declarations canno
accessed directly (J.y is an error).

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[ANY] or TYPE[e]; the type being declared is an opaque type or exported type, dis
in ¶ 4.3.4. The expression e must be static. TYPE[ANY] or TYPE[e] is not allowed in a
ordinary declaration; except in an interface, a type name must be bound to a type
when it is introduced.

VAR T, or READONLY T for any type T except TYPE; this is an interface variable; discus
in ¶ 3.3.4.1 below. •In an ordinary declaration in a block you can't write VAR T, b
write simply T; you can also write simply T here, but this is not recommended

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 (¶ 3.3.6). This can be useful if the name is used in a type constructor
in the interface, but its value should not be accessible to the client.

3.3.4.1 Interface variables

An interface variable v gives clients of an interface direct access to a variable in a p
namely the variable which is exported to v. This is the only kind of variable parameter
Cedar.

•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 sooner.

Performance: An interface variable can be read and (if not READONLY) set directly, which i
significantly faster than 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. If you use a private interface variable and inline Get and Set
pay nothing in performance, but retain the option of changing the proc definitions later

•You can get direct access to all the variables of a module by using a POINTER TO FRAME typ
(¶ 4.5.3).

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 PRO
proc, which has parameters and results like any other. PP(is special in one way: it has
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., the frame of PP() are retained after the proc returns; in fact, forever (unless RU
used to free the frame). Procs local to the block can access these values in the usual w
of exported names can also be accessed through interfaces, as explained below; see ¶ 3.3

As with any proc (¶ 3.5.1), the frame of PP(includes the parameters and results from Imp
as well as the names introduced in the block's d's and b's. It also includes an addition

```
Imp: PROGRAM T~PP(
```

where Imp is the name of the module and T is its drType.

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: PUBLIC T~x is in the body of Imp, then n~x is in ExI. This is slightly peculiar kind of binding; as in an ordinary binding, x must be coerceable (¶ 4.13). Note that n must 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 is in ExI. UNBOUND is a special value with the following properties:

For a proc P, it causes a Runtime.UnboundProcedure signal on any application of

For a variable v, it causes a Runtime.PointerFault error on any reference to v

For a type T, it causes no problem.

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 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 CONTROL: PROGRAM[] RETURNS [], whose value is the program proc PP(if that has no arguments, and otherwise Runtime.ReportStartFault.

•One item for the type of the module's global frame, namely FRAME~TYPE Imp.

•One item for Imp itself, namely Imp: FRAME. The value of this item 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 the program proc, the global frame type, and the program instance.

•You can pass FRAME as an argument to a DIRECTORY parameter I: TYPE Imp; like an interface; provides access to constants bound in the module, and allows you to declare an IMPORTS par whose argument will be a program instance of the module. From I you can also obtain a fi Cedar type POINTER TO FRAME[I]; see ¶ 4.3.5. I's cluster includes a coercion from I to POINT FRAME[I], and the proc COPYIMPLINST (applied by the funnyAppl NEW), which is the same as the proc of the same name in cluster of POINTER TO FRAME[I].

•You can import Imp into another module (by writing DIRECTORY Imp ... IMPORTS ImpInst: Imp .. and obtain access to all the variables and procs of the program instance.

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.

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

```
x: PUBLIC T
```

In the Cedar syntax these colons occur in the declarations[11] and bindings[13] in bodies[10], f and interface modules[2], and in the tag[53] of a unionTC. You can set a default access for a names in a module[2, 3] or record[50] by writing PUBLIC or PRIVATE just before the { or RECORD; th overridden by an explicit PUBLIC or PRIVATE inside. By default, an interface is PUBLIC and a implementation is PRIVATE.

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

- from within M;
- from a module which EXPORTS M.
- •from a module which SHARES M; avoid this feature.

This does not mean that the name is invisible, but rather that it is an error to use it OPENed. Thus in

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

if x is bound in M (and not hidden by a USING clause), the call of f is equivalent to f [M
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 (even if the private compon
not mentioned and have defaults).

3.4 Blocks, OPEN and ENABLE

```
6 block ::= ?(CHECKED | UNCHECKED | TRUSTED)
  { ?open ?enable ?body
     ?(EXITS (n, !..=>s); ...) }
  --In 3, 13, 15.
7 open ::= OPEN ( n ~~ e | e ), !.. ;
  In 2, 5, 17. •The ~~ may be written as :.
8 enable::=ENABLE ( enChoice |
  In 5, 17.
9 enChoice ::=( e, !.. | ANY ) => s
  In 7, 27.1.
10 body  ::= (d | b); !.. ; s; ... | s;
  In 5, 17.
```

```
open LET n((, ... : EXCEPTION~NEWLABEL[] , ...
  IN ((body enable)BUT { (n((, ... => s ); ... } )
-- But n(( is not visible in s.
( LET n~l_open IN e.DEREF |    --The IN before !.. is a separat
  LET BINDP[D(DEREF)P,
    OPENPROCS[D((e.DEREF))P, l IN e.DEREF] ] ) IN !.. IN
BUT ( { enChoice }
  {enChoice; ...}enChoice)
( e | ANY ), ... => { s; REJECT; EXITS
    Retry(=>GOTO Retry((14; Cont(=>GOTO Cont((14 }
SET NEWFRAME[ REC [(d | b), ...] ].UNCONS IN { s; ...}
```

Examples

```
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[UXie]; -- Use a binding if a name's value is fixed.
  x: INT_7;                              -- Better to initialize declared names.
    {OPEN b~~buffer;                     -- A statement may be a nested block.
      ENABLE  {                          -- Multiple enable choices must be in {}.
      Files.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: CHECKED, UNCHECKED and TRUSTED are treated in ¶ 3.4.4 on safety.

open[7]: a combination of sugar for LET and call by name; see ¶ 3.4.2.

enable[8]: 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).

The statements in a block are evaluated in the order they are written. The initializatio
in the d's and b's are also evaluated in the order they are written; this may be importa
have side effects, although that should be avoided.

3.4.1 Scope of names and initialization

The names introduced in the block body's d's and b's (i.e., appearing before a : or ~) a
the body with the values supplied by the d's and b's, except in inner scopes where they
reintroduced; they are not known elsewhere in the block. The frame of the block is a bin
a value for each 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 d's and b's, but before
This means that n.VALUEOF is trash in all the d's and b's before its binding. Symmetricall
refers to a name introduced in a following decl or non-static binding, it will get a tra
Compiling with the "u" switch will yield a warning in this case. Note that only attempts
the value of n get trash; n may appear anywhere in a l-expression, and all will be well a
the l-expression is not 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 before anything in the block is executed (this is done by the NEWFRAME
the desugaring).

Anomaly: However, the INIT proc is executed (to set a REF or transfer value to NIL), and a
initialization specified by a defaultTC[55] in T is done at the same time that a non-static
would be evaluated. As with a binding, n.VALUEOF is trash before this time. Furthermore, a
(unwise) assignment to n before this time will be overridden by the defaultTC.

Caution: The failure to initialize RC variables is a safety loophole, since the trash ca
and used as an address.

Style: The expression in a binding or defaultTC should be functional, or at least it sho
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 forbidden, and ASSIGN for proc values gives an error unless the proc is local
instance (which has a retained frame).

Caution: An unchecked program can get into trouble with dangling references to frames, h

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 the variables bound to its names
style to use blocks freely 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 cluster of De does not include DEREFERENCE or UNWRAP;
>
> e^.DEREF if it includes DEREFERENCE;
>
> e.UNWRAP.DEREF if it includes UNWRAP.

In other words, a reference value is dereferenced and a single-component record or bindi
replaced by the component, repeatedly if necessary, to obtain a non-reference value. In
e.DEREF must be a record, interface or instance.

The second, nameless, form of open gives an expression without binding it to a name: { OP
...}; e.DEREF 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 bound to the field values (or variables, for a
>
> An application returns a binding, though the call-by-name feature makes it unwise
> an application in an open.
>
> 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 (
above), and introduces bp's names in the block with a LET. Thus in the program

```
R: TYPE~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: Nameless open should be used with discretion, with the smallest practicable scope
if the value being opened is very familiar, or heavily used, or both. Nameless open can
confusion, since it is not obvious from the text of the program where to find the bindin
names it makes known. It should never be used when evaluation of e has a side-effect.

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

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 (enChoice[9]) has a list of expressions with exception values, •or ANY,
=>. If ANY appears, it must be the last enChoice. If the exception is equal to one of the
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[16]:

A RETURN is not allowed in an enChoice.

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
d's or b's.

•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 d's or

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 e appears before the =>, then within the enChoice statement the n
De.DOMAIN are declared and initialized to the arguments of the exception. With multiple
expressions, or ANY, the arguments are inaccessible. •The use of ANY is not recommended.

Note that an error is caught by an enChoice with a matching exception value, not by one
matching name. Normally an error E will be declared in some interface, its value will be
by a binding of the form E: PUBLIC ERROR ... ~ CODE, and both the signaller and the enChoic
refer to this value by the name E. In this case, it is natural to think of the binding a
name. However, it is possible to have a different name for this exception value, e.g. by
ERROR ... ~ E. It is also possible to bind some other exception value to E in a scope whi
some enChoice examined when the signal is raised. Thus in the silly program

```
E: ERROR~CODE;
F: ERROR~E;
{ENABLE E=>{--Handler 1--...};
  E: ERROR~CODE;
  {ENABLE E=>{--Handler 2--...};
    IF switch THEN ERROR F ELSE ERROR E;
```

if switch is true handler 1 will be used, and if it is false handler 2 will be used.

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 w
the ERROR. As each application is stopped, it is finalized. Aside from invisible housekee
finalization confusingly consists of executing an enChoice which catches the ERROR UNWIND.
programmer can write any cleanup actions he likes in this statement.

Caution: If the finalization raises another ERROR which it does not catch, it will itself
with very confusing consequences. It isn't very useful to know exactly what happens then
this situation.

Anomaly: In fact, things are a bit more complicated. When a signal or error is propagate
enChoice statement is called as a proc from the SIGNAL or ERROR which raises the exception
control leaves the statement by a GOTO (including EXIT, CONTINUE, RETRY or LOOP, but not
RETURN, which is forbidden in an enChoice), the finalization is done. This means that the
statement is executed before any finalization. This is useful for signals, which often r
cases, however, notably if finalization would release monitor locks, this can cause trou
problem by exiting from the enChoice immediately with a GOTO.

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

Conceptually, a signal is quite different from an error; in fact, it is very much like a
call. The only differences are:

> The proc to be called is an enChoice which is found exactly as though the signal w
> error. The effect of this is that SIGNAL P[args] binds the proc name P to the proc b
> dynamically, by searching up the call stack for a binding of P. This is just the w
> binds free variables, except that a binding for P can only be found in an enChoice
> the frame of a proc.
>
> Actually this is not quite right. Like an error handler, the signal proc is not found by matching names
> matching exception values. This point is discussed in detail above.
>
> The enChoice can be terminated by a GOTO out of its body, unlike an ordinary proc.
> GOTO exception is treated exactly like a GOTO out of an enChoice for an error; it ca
> the intervening frames to be finalized.

The implementation, however, treats errors and signals in a very similar way; the only d
that you cannot resume an error (return from the enChoice). In fact, you can invoke a si
ERROR, which prevents it from being resumed; avoid this feature. In the future, however,
distinction between signals and errors will be reflected more clearly in the implementat

Anomaly: The desugaring gives no explanation of how RESUME works, since it does not turn
enChoice for a signal into a proc at all. This is a defect.

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.

Anomaly: A GOTO skips any UNWIND enChoices that intervene between the GOTO and its matching
EXITS. This is the only way to escape from a block without executing the UNWIND. You can av
this anomaly by not nesting UNWIND enChoices within blocks that have EXITS.

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 blo

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 (except ERROR, which is automatically safe). Th defaults are provided to make it convenient to:

write new programs in the safe language;

continue to use old, unsafe programs without massive editing.

An unsafe proc value never has a safe type, and hence cannot be bound to a name declared safe type. This applies to enable choices for signals as well as to procs. In both cases be checked or trusted if the type is safe. ERRORs are treated differently, however, becau view that an ERROR is a value returned from an application, unlike a signal which calls t enChoice expression. Hence the enChoice for an ERROR is treated just like any statement i enclosing block, 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.

APPLY of

a descriptor (because it involves dereferencing a pointer);

a computed sequence;

a record containing a computed sequence;

a base pointer.

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

withSelect[34].

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.

LOOPHOLE which produces a RC value (¶ 4.5.1).

3.5 Declaration and binding

```
11 declaration ::= n, !.. : ?access[12] varTC[40], ...
   In 2, 10, 43. VAR, READONLY only for interface var.
12 access ::= PUBLIC | PRIVATE
   In 2, 3, 11, 13, 50, 51, 53.
13 binding ::= n, !.. : ?access[12] t ~ ( n, ... ~LET x( : t ~ (
   e |                                     e |
   t2  -- if t=TYPE  |                     t -2 -- Same as e except for conflicting syntax. |
   CODE |                                  NEWEXCEPTIONCODE[]  --tgSIGNAL or ERROR |
   ?INLINE ?(ENTRY | INTERNAL) block[6]  |  l [d(: t.DOMAIN] IN LET r(~NEWFRAME[t.RANGE].UNCONS
                                             INNET(r( IN {t.DOMAIN~d(; block; RETURN}
                                                   BUT {Return(()(=>r(})
   „, ?TRUSTED MACHINE CODE {(e, ...); ...} MACHINECODE[(BYTESTOINSTRUCTION[e, ...]), ...]
   )                                       ) IN  x(  -- e is evaluated only once.
   In 2, 10. •The ~ may be written as =.
   Block or MACHINE CODE only for proc types.
   •ENTRY and INTERNAL can also be before t.
```

Examples

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

H: TYPE~Histogram11; Size: INT~10;       -- Implementation: Binds a TYPE and INT.
HistValue: PUBLIC TYPE~HV40.1;           --            PUBLIC for exports.
baseHist: PUBLIC H_NEW[HistValue_ALL[17]];            An exported variable
x, y: HistValue_[ 20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0];  with initialization.
FatalError: ERROR[reason: ROPE]~CODE;    --            Binds an error.
Setup: PROC [h: Handle3, a: INT]~ENTRY {...};         Binds an entry proc.
i,j,k: INT_0; p,q: BOOL; lb: Label; main: Handle;
```

Declarations are explained in ¶ 2.2.1F and ¶ 2.4.5. Their peculiarities in the different
they can appear are explained elsewhere:

- interfaces in ¶ 3.3.4;
- blocks in ¶ 3.4.1;
- fields in:
  - domains and ranges in ¶ 4.4;
  - records and unions in ¶ 4.6;

Access is explained in ¶ 3.3.6.

Bindings are explained in ¶ 2.3.5. There are several special forms of binding given in r
however, which are defined here. See also ¶ 3.7 on argument bindings. Note that the e in
is evaluated just once, even if several names are bound.

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 type36 ra
an expression19 after the ~.

A name with a signal or error type can be bound to CODE; this use of CODE is not all
anywhere else. See ¶ 4.4.1 for details on the meaning of this.

„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 semicolons. Each instruction is assembled from a list of expressions
by commas. An expression in the list is evaluated to yield a [0..256) static value
forms one 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[55] is turned into a parameterless proc which is bound to Default in the typ cluster (¶ 4.11).

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 an exception is turned into a proc with domain and range given by exception type (¶ 3.4.3.1).

The expression following LOCKS in a module heading is turned into a proc according to a peculiar rule (¶

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 implicitly declares a variable for each it domain and range; these variables have the names of the domain and range items, and thei is the entire block, not just the block body. The domain variables are initialized to th and the range variables in the usual way according to their types. Then the block, with tacked on the end, is evaluated. A RETURN exception in the block is caught, and the curre of the range variables are the result of the l-expression. The only other way out of the raise an ERROR.

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

Anomaly: 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 local, i.e., deno which was bound to a proc body in the same module as the call. A proc which is local to proc, rather than bound in the body of an implementation, is about 20% slower to call. I introduces some overhead when its parent proc is called, and its access to non-static na introduced in its parent proc is slower than access to other names. A call and return fo ordinary, non-local proc takes about 10 times as long as the statement x_y+z.

The attributes ENTRY and INTERNAL can be used only in a MONITOR; they are discussed in ¶ 4.1

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 sometimes the cost of argument passing, and it may allow constant arguments to participa evaluation within the proc.

Restrictions: An INLINE proc may not be:

Recursive.

Exported.

Used as a proc value except in an application. Thus you cannot, for example, assig proc variable.

The argument of FORK.

Accessed from the cluster of a POINTER TO FRAME type.

Anomaly: An inline proc binding in an interface is not accessible from a DIRECTORY argumen must import the interface.

Performance: Excessive application of inline procs will result in much larger compiled c Excessive definition of inline procs will result in much larger data structures in the c hence in larger symbol table files, and a greater chance of overflowing the compiler’s c following cases are efficient:

An inline proc in an implementation which is called zero or one times.

An inline proc which has a simple body, no locals, no named results, and no access formals after potential side effects.

3.6 Statements

```
14 Statement ::= sS                          { SIMPLELOOP {sS; GOTO Cont((; EXITS  Retry((=>NULL};
  In 6, 10, 17, 19.                             EXITS Cont((=>NULL }
15 sS ::=e1_e2 | e | block6 | escape | [loop e1_e2].NOVOID | e --must yield VOID-- | --all four yield VO
16 escape ::= GOTO n | GO TO n |             HEX[exception[code~ n((, args~NIL]] |
  EXIT | CONTINUE | •LOOP | •RETRY |        GOTO ( Exit(17 | Cont(9 | Loop(17| Retry(9) |
  (RETURN | RESUME) ?e |                    { ?(r(13_e;) GOTO (Return(13 | Resume(13) } |
  •REJECT | ,„e _ STATE                      THISEXCEPTION[]  | DUMPSTATE[e]

17 loop ::= (iterator | )                    { ( iterator | done(~FALSE; Next(: PROC~{}; )
  (WHILE e | UNTIL e | )                        { Test(~l IN (NOT e | e | FALSE);
  DO  ?•open7 ?•enable8 ?body10                  { OPEN | ENABLELOOP {
                                                    IF Test([] OR done( THEN GOTO FINISHED;
                                                    { enable body EXITS Loop(=>NULL }; Next
  ?(REPEAT (n, !..=>s); ...) ENDLOOP            EXITS Exit(gNULL; (n, !..gs); ...; FINISHEDgNULL}}}

18 iterator ::= THROUGH  e |                 FOR x(: e IN | e
                    FOR (n : t | µn)         ( n: t; | )
                                ( ( | DECREASING)(: INTYPE~e; done(: BOOL_Range(.ISEMPTY;
                                              Next(: PROC~{ IF n ( >Range(.LAST | <Range(.FIRST )
                                                THEN done(_TRUE ELSE n_n.(SUCC | PRED) };
                                              n_Range(.(FIRST | LAST); |
                                    _ e1 done(: BOOL~FALSE; Next(: PROC~{n_e2}; n_e1 ) ;
  e is a subrange. In FOR n: t ... , n is readonly except for the assignment in the iterator's desugaring.
```

Examples

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

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

THROUGH [1..5) DO i_i*i ENDLOOP;     -- Raises i to the 16th power.
FOR i: INT_1, i+2 WHILE i<8 DO j_j+i ...; -- Accumulates odd numbers in [1..8).
FOR l: Label_lb, l.Next WHILE l#NIL DO ...; Sequences through a list of Labels.
```

Cedar makes a distinction between expressions and statements. This distinction is most e
defined in terms of a special type called VOID, which is equivalent to the empty declarat
is the range type of a PROC [...]_[], and it is also the result type of a block, control,
statement. An expression whose value is a VOID can be used as a statement, and cannot be
an ordinary value in a binding (since it wouldn't have the right type). If you want to c
which returns values as a statement, you must assign 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. Note t
grammar is ambiguous here, since there are two parsings of e_1_e_2 as a statement; the one
the rule for statement is preferred.

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 rather complicated construct, as the desugaring shows. This i
the CONTINUE and RETRY statements, which respectively terminate and repeat the statement
containing the enable[9] in which they appear. The desugaring shows exactly what this means
various obscure cases. CONTINUE and RETRY are legal only in an enable choice (¶ 3.4.2), and
may not appear in a declaration at all. •RETRY should be avoided everywhere, since it int
loop into the program in a distinctly non-obvious way.

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

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

A loop[17] is repeated indefinitely until stopped by an exception, or by the iterator[18] or
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
> 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 th
> 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.
>
> •LOOP causes the next repetition of the loop to start immediately.

Anomaly: You cannot write GOTO FINISHED.

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 (i.e., in the FINISHED clause), the value of v is undef
omit the declaration and simply name an already declared variable, it will be used as th
variable, and will not be read-only; it will still be undefined after the loop is termin
iterator. Avoid this feature.

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 finishes the loop after a repetition which leaves v>last. The > case can o
FOR v IN ..., when an out-of-range value is assigned to v in the loop body. DECREASIN
reverses the order in which the elements of the subrange are used. The subrange ne
be static. Note that the subrange is evaluated only once, before execution of the
begins.

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

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

Note that the WHILE or UNTIL test is made with v equal to its value during the next repeti
that both tests are made before the first repetition, so that zero repetitions are possi

3.7 Expressions

```
19 expression ::= n | literal | application[26] |
  (e | typeName[37]) . (9) n |
  prefixOp e | e1 infixOp e2 |                    e . prefixOp | e1 . infixOp [e2] |
  e1 relOp (4) e2 |                               ( l [x(: De1, y( :De2 ])[e1, e2] ) |
  e1 AND (2) e2 | e1 OR (1) e2 |                  IF e1 THEN e2 ELSE FALSE | IF e1 THEN TRUE ELSE e2 |
  e ^ (9) | •STOP | ERROR |                       e . DEREFERENCE | STOP[] | ERROR NAMELESSERROR |
  builtIn [ e1 ?( , e2, !..) ?applEn[27] ] |    e1 . builtIn ?( [e2, ... ?applEn ] ) |
  funnyAppl e ?( [?argBinding[27] ?applEn[27]] ) | funnyAppl ?( [argBinding ] ) |
  [ argBinding[27] ] |                          --Binding must coerce to a record, array, or •local
  s | subrange[25] | if[28] | select[29] | safeSelect[32] | •withSelect[34]
  Precedence is in bold in rules 19-21.  All operators associate to the left except _, which associates
  to the right. Application has highest precedence. Subrange only after IN or THROUGH. s only in if [28] and sel

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

25 subrange ::= (typeName[37] | )                LET t(~(typeName | INT) , first(~( e1 | e1.SUCC ) IN
                ) e1 .. e2 )( [] || ()             t(.MKSUBRANGE[first(, (e2 | e2.PRED )] BUT
  --In 19, 39, 48.                                    {BoundsFault=>t(.MKEMPTYSUBRANGE[e1]}

26 application ::= e [?argBinding ?applEn](~e, a(~[argBinding] in (. APPLY Za(?applEn)
27 argBinding ::= (n ~ (e | | •TRASH ))(n !~ .(e | OMITTED | TRASH) ), !.. |
                  (e | | •TRASH ), OMITTED | TRASH ), ...
  In 19, 26. •TRASH may be written as NULL, ~ as :.
27.1 applEn ::= ! enChoice[9]; ...-- In 19, BUT 26. { enChoice ... }
```

Examples

```
lv: LabelValue13_[ i, 3, "Hello", 31.4E-1,
  g[x]+lb.f+j.PRED, NIL ];              -- A constructor with some sample
                                       --   expressions.
p1: PROCESS RETURNS [INT]_FORK f[i, j];  -- FunnyAppls take one unbracketted
ERROR NoSpace; WAIT bufferFilled;        --   arg; many return no result, so
RT: RTBasic.Type_CODE[LabelValue13];     --   must be statements.
h[ 3, NOT(i>j), i*j, i_3, i NOT >j, p OR q, "Short"];  -- An application with sample expressions.
lv19_[first~0,last~5,x~3.2,g~2,f~5,r~NIL,s~"Hi"];      -- Short for lv_LabelValue13[...].
[first~i, last~j]_lv19;                -- Assignment to VAR binding
                                       --   (extractor).

b: BOOL_i IN [1..10]; FOR x: INT IN (0..10) DO ...;  -- Subrange only in types or with IN.
b_( c IN Color54(red..green] OR x IN INT[0..10));    -- The IN is redundant.

fh_Files.Open[name~lb.s, mode~Files.read,  -- Keywords are best for multiple args.
  ! AccessDenied=>{...}; FatalError=>{...}];  -- 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 defaultTC55).
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 proc[23], or the funnyAppl[24] in which the fi
follows the proc name without any brackets. All of these constructs desugar into dot not
(¶ 2.4.4, ¶ 4.14); this means that the procs come from the cluster of the first argument
exceptions to this rule are ALL, CONS for variant records and lists, LIST, and the single-a
forms of LOOPHOLE and NARROW, and VAL; all of these get the proc from the target type of the
expression (¶ 4.2.3). All the primitive procs are described in ¶ 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

The order of evaluation for arguments of an application, and therefore for operands in a
expression is not defined (unless the operator is AND or OR). However, the arguments are
one at a time, and all arguments are evaluated before the proc is applied. In particular
assignment which executes completely behaves as though both left and right operands are
completely evaluated before any assignments are done, even if the left side is a binding
[a~x, b~y.f].

Rules 19-21 give the precedence for operators: ^ and . are highest (bind most tightly) a
lowest. All are left-associative except _, which is right-associative. Application has s
precedence.

Style: The precedence rules are sufficiently complex that it is wise to parenthesize exp
depend on subtle differences in precedence.

The first operand of assign can be an argBinding[27] whose value is a variable group or bin
one whose elements are variables; this is sometimes called an extractor. The second argu
typecheck if it is a group or binding with corresponding elements which can be assigned
variables. Usually the second argument is either an application which returns more than
or a record-valued expression. You can omit elements of the left argBinding to discard t
corresponding values; however, you can’t write TRASH in the left operand. Note that the r
operand is fully evaluated before any variables are changed by the assignment.

The expresssion ERROR is short for raising a nameless ERROR exception. You should think of
call to the debugger, appropriate for a state which "can’t occur".

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"]. RETURN WITH ERROR is explained in ¶ 4.10.

Anomaly: The funnyAppl NEW e actually stands for e.COPYIMPLINST. See ¶ 4.4.1 and ¶ 4.5.3.

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

A subrange[25] denotes a subrange type; see ¶ 4.7.3. Standard mathematical notation for ope
closed intervals is used to indicate whether the endpoints are included in the subrange.
can also be used after IN in an expression or iterator; in these contexts it need not be

You can write enable choices[9] after a ! inside the brackets of an application[26], built-in
funnyAppl[24]. See ¶ 3.3.2 for the semantics of this. Note that only an exception returned
application is caught by these choices, not one resulting from evaluating the proc or ar

An argBinding[27] denotes a binding for the arguments of an application. You can omit a
[name, value] pair n~e in the binding if the corresponding type has a default, or you ca
name without the value expression (e.g., n~) with the same meaning. You can also write
(•or 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
ELSE ...

The ";" is "," in an expression; also in 32- and 34.

30 choice ::= ((| relOp[22]) e_1), !.. => e_2

IF (selector((= | relOp) e_1) OR ...) THEN e_2

31 endChoice ::= ENDCASE (| => e_3)

ELSE (NULL | e_3)

In 29, 32, 34.

32 safeSelect ::= WITH e SELECT FROM
safeChoice; ... endChoice[31]

LET v(~e IN
safeChoice; ... endChoice
ELSE ...

33 safeChoice ::= n : t => e_2

IF ISTYPE[v(, t] THEN LET n : t_NARROW[v(, t] IN e_2

34 •withSelect ::= WITH (n_1 ~~ e_1 | • eOPEN e_1)
SELECT (| ,e_{11}) FROM
withChoice; ... endChoice[31]

IF v(~~e_1 IN LET n(~(n_1 | NIL), type(~Dv(,
selector(~(e_1.TAG | e_{11}) IN withChoice; endChoice
-- e_{11} must be defaulted except for a COMPUTED variant.

•The ~~ may be written as :.

35 •withChoice ::= n_2 => e_2 |
n_2, n_2, !.. => e_2

IF selector(=n_2 THEN OPEN
(BINDP[n(, LOOPHOLE[v(,type(.n_2]] | BINDP[n(, v(])

Examples

```
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: 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 Node52~...; WITH dn~~nr SELECT FROM See rule 52 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 to yield a value selector(, a
the choices is tested in turn. Within each choice, each expression e_1 preceding the => is
in turn with selector(; the comparison is selector(relop e_1 if e_1 is preceded by a relop; o
selector(=e_1. If any comparison succeeds, the expression e_2 following the => is evaluated
value of the select. If no comparison succeeds, the next choice is tried. If no choice s
expression e_3 following the ENDCASE is evaluated to yield the value of the select; e_3 defau
NULL, and hence must be present when the select is not a statement to prevent a type erro

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 to mean another specific selector value which you do
bother to mention. Another acceptable form is SELECT TRUE FROM ..., which selects the first
that succeeds, and is sometimes easier to read than a long sequence of ELSE IF's.

Performance: If the e_2 are static and select subsets of the selector values, the average
subsets is not too large, and the density of unselected values is not too high, a select
an indexed jump, which executes in a time independent of the number of choices.

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, REF V, or (LONG) POINTER TO V, where V is a variant record. Each choice specifies
possible type that the selector might have, and declares a name which is initialized to
value if it has that type. Thus, the example tests for r having the types REF INT and REF
has REF INT, the first choice's e is evaluated; within e, rInt is a variable initialized t
and has type REF INT. Likewise for REF REAL and the second choice. As with an ordinary sele
ENDCASE expression is evaluated (with no new names known) if none of the other choices suc
Note that safeSelect does ordinary binding by value, not the binding by name done in ope
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_{11} in the withSelect. It is entirely up to the programmer to supply a value. If the tag is not COMPUTED, e_{11} must be omitted and the selector value is e_1.T

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_2 following => is evaluated as with an ordinary select. If exactly one is give

e_2 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_2 following => is in the scope of

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

3.9 Miscellaneous

This section deals with various topics that are not naturally associated with particular grammar rules.

3.9.1 Static values

An expression has a static value if the compiler can compute the value. In Cedar, an exp a static value (is static for short) if it is:

a literal;

a name bound to a static value;

an application to static arguments of

a proc declared INLINE with a static body, or

a primitive which is not a loop, a REAL primitive (except unary minus, ABS or INTTOREAL), ASSIGN, @ or NEW. Note that IF and SELECT are evaluated.

Performance: The compiler evaluates all static expressions, not just type expressions. T important for efficiency.

3.9.2 Size restrictions

Current Cedar has the following restrictions on the sizes of values:

- A record type T must have T.SIZE<2^{16}.
- A row type T must have T.SIZE<2^{28} and T.RANGE.SIZE<2^{16}.

- A type T with T.SIZE>2^{16} lacks the following procs:
  ALL
  ASSIGN
  CONS
  DESCRIPTOR
  INIT
  NEW
- A subrange type T must have
  0<T.LAST T.FIRST<2^{16}
  2^{15}<T.FIRST<2^{15}
  T.LAST<(IF T.FIRST THEN 2^{15}+T.FIRST ELSE 2^{16})

2.9.3 Checking

Possible errors arising from certain primitive operations are checked, and cause ERROR ex
they occur, in a CHECKED block, or if the compiler's "u" switch is on:

Dereferencing NIL.

Narrowing an out-of-range value to a subrange type.

Assigning a local proc to a proc variable (in CHECKED blocks only).