A Description of the Cedar Language

A Cedar Language Reference Manual

Butler W. Lampson

CSL-83-15 December 1983 (Printed November 1986) [P83-00016]

© Copyright 1983, 1986 by Xerox Corporation. All rights reserved.

Abstract: The Cedar Language is a programming language derived from Mesa, which in turn is derived from Pascal. It is meant to be used for a wide variety of programming tasks, ranging from low-level system software to large applications. In addition to the sequential control constructs, static type checking and structured types of Pascal, and the modules, exception handling, and concurrency control constructs of Mesa, Cedar also has garbage collection, dynamic types, and a limited form of type parameterization.

This report describes the Cedar language. Except for chapter 2, it is written strictly in the style of a reference manual, not a tutorial. Furthermore, it describes the entire language, including a number of obsolete constructs and historical accidents. Hence it tells much more than you probably want to know. A summary of the safe langauge and comments throughout the manual suggest which constructs should be preferred for new programs.

CR Categories and Subject Descriptors: D.3.2 [Programming Languages]: Language Classifications – Cedar, extensible languages; D.3.1 [Programming Languages]: Formal Definitions and Theory – semantics

Additional Keywords and Phrases: kernel language, polymorphism, data types

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

The work described here was completed in late 1983 but not published at that time. It attempts to provide a reasonably formal and precise definition of the Cedar programming language. The then current version of the language was perceived as an inadequate base for a number of planned extensions to the language and supporting environment; on the other hand, there was already a large body of Cedar code that could not simply be abandoned. These problems are dealt with by defining a small but powerful kernel language plus a mapping of existing Cedar constructs into that kernel. The kernel language introduces value spaces and operations over them that go well beyond what has been available in any implemented version of the Cedar language; it was to provide the basis for extension and simplification. The mapping from existing Cedar into the kernel provides not only a migration path for existing code but also a definitional method.

This report should be of interest to students of programming languages and their definitions. Most of the interesting ideas of the Cedar language appear in the kernel, which is described in Chapter 2. Such readers should note that the formalism used to describe the kernel has several known shortcomings. Its treatment of so-called dependent types is somewhat cavalier. A subsequent report by Burstall and Lampson ("A Kernel Language for Modules and Abstract Data Types," Digital Systems Research Center, September 1984) includes a more careful treatment of such types in a language very similar to the kernel. The present treatment also glosses over most of the definitional problems raised by the possibility of concurrent evaluation.

The report should also be of interest to Cedar programmers. Chapters 3 and 4 constitute the most complete, precise and accurate definition of the implemented Cedar langauge that has appeared to date. For a reader willing to make the effort to assimilate the concepts introduced in Chapter 2, this report can serve as an interim reference manual. The later chapters are painfully honest and complete; as the abstract notes, they say much more than anyone probably wants to know. As of March 1986, the only known differences between the description and implemention, other than minor bugs in each, are the following:

- The improved syntax for ENTRY and INTERNAL has not been implemented; these attributes must still precede the type in a procedure declaration (Section 3.5).
- Sections 3.3.4 and 4.3.4 document an improved design for opaque types that was never implemented. In current Cedar, opaque types behave as they do in Mesa.
- According to Section 4.14, if *P* is a procedure taking one argument, its application to *x* using dot notation is written without brackets, as *x.P*. In current Cedar, the alternative form *x.P*[] is also accepted.

Both classes of readers should note that many parts of the kernel language have never been implemented in their full generality. Some of the current developers and users of the Cedar language would not even agree that the directions of evolution suggested by the kernel language are desirable or feasible. The claims about long-term goals and promised improvements in this report should therefore be taken as the personal opinions of the author.

Ed Satterthwaite, March 1986

Chapter 1. Introduction

The Cedar language is a programming language derived from Mesa, which in turn is derived from Pascal. It is meant to be used for a wide variety of programming tasks, ranging from low-level system software to large applications. In addition to the sequential control constructs, static type checking and structured types of Pascal, and the modules, exception handling, and concurrency control constructs of Mesa, Cedar also has garbage collection, dynamic types, and a limited form of type parameterization.

This manual describes the Cedar language. Except for the overview material in § 2.1 and the discussion of concepts in §§ 2.3-2.7, it is written strictly as a reference manual, not a tutorial. Furthermore, it describes the entire language, including a number of obsolete constructs and historical accidents. Hence it tells much more than you probably want to know. A summary of the safe language and comments throughout the manual, suggest which constructs should be preferred for new programs.

The manual is organized into three major parts:

> Chapter 2: A description of a much simpler *kernel* language, in terms of which the current Cedar language is explained. This description includes:
>
> > An overview or glossary, in which the major technical terms used in the kernel are briefly defined (§ 2.1).
> >
> > An informal explanation of the ideas of the kernel and the restrictions imposed by current Cedar (§§ 2.3-2.9)
> >
> > A precise definition of the kernel (§ 2.2). Most readers will probably find this rather hard going.
>
> Chapter 3: The syntax and semantics of the current Cedar language. The semantics is given precisely by a *desugaring* into the kernel. It is also given more informally by English text. 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 given as well as an English definition of its meaning. This chapter is organized according to the *class* hierarchy of the primitive types (§ 4.1).

In addition, there is a one-page grammar for the full language, a shorter grammar for the safe language, and a two-page language summary which includes the grammar, the desugaring, and the examples from § 3. The tables in §§ 4.1-2 summarize the types and primitives.

To find your way around:

> First read chapter 2, except for § 2.2.
>
> Then consult the table of contents, or the index, for the topics of interest to you. The full grammar (at the end) and the class hierarchy (Table 4−1) may also be useful as starting points.
>
> The manual is extensively cross-referenced. Section titles and numbers appear at the top of each page. The summaries and tables also point to the section in which each construct is defined.

Acknowledgements: Rod Burstall and Ed Satterthwaite helped me greatly in clarifying the ideas presented in § 2. Ed was also indispensable in getting an accurate description of the current Cedar language. Bill McKeeman's work on an earlier Cedar language description was the starting point for this manual. Will Crowther, Jim Horning and Lyle Ramshaw read part or all of the manual carefully, and made many helpful comments. Several other Cedar programmers have pointed out errors or omissions. Of course, I am responsible for the errors that remain.

Chapter 2. The kernel language

This document describes the Cedar language in terms of a much smaller language, which we will 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 explained 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 which 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 or *primitive* types and procedures (§ 4). In the kernel language all of these could in principle be programmed by the user, though in fact most are provided by special code in the Cedar compiler. In general, you can view these built-in facilities much like a library, selecting the ones most useful for your work and ignoring the others.

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

To help in keeping the kernel and current Cedar separate, reserved words and primitives of the kernel which are not available in current Cedar are written in SANS-SERIF SMALL CAPITALS, rather than the SERIF SMALL CAPITALS used for those symbols in current Cedar. Operator symbols of the 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 entirely separate invention. Most Cedar constructs have *simple* translations into the kernel. Those which do not (e.g., some of the features of OPEN) are considered to be mistakes, and should be avoided in new programs.

Roadmap

§ 2.1 gives a brief summary of each major idea in the kernel, which may be helpful as an introduction and reminder. Most of the chapter (§§ 2.3-2.8) is an informal explanation of the concepts behind the kernel. Usually, terms are defined and explained before they are used, but some circularity seems to be unavoidable. Both this and the explanations in §§ 2.3-2.7 are given under five major headings, as follows:

Values and computations

The type system

Programs

Conveniences

Miscellaneous

There is also a sketch of the restrictions imposed by the current Cedar language on the generality of the kernel; for more on this subject, see § 3. The meaning of the various built-in primitives is given in § 4. The incompatibilities between the kernel language and current Cedar are described in § 2.9, i.e., the constructs in Cedar which would have a different meaning in a kernel program. For the most part, these are bits of syntax which do not have consistent meanings in current Cedar; future evolution of the language will replace them with their kernel equivalents.

§ 2.2 precisely defines the syntax and semantics of the Cedar kernel language, the former with a grammar, and the latter by explaining how to take a program and deduce the function it computes and the state changes it causes. The kernel definition follows the ordering of the kernel grammar. This section is rather difficult to read, and you may prefer to skip it.

2.1 Overview

This section gives a brief summary of the essential concepts on which the Cedar language is based. The explanations are *informal* and *incomplete*. For more precise but more formal definitions, see § 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* for short) to *arguments*. When the proc is finished, it returns some *results*, which can be discarded or passed as arguments to other procs. The application may also change the values of some variables. In the program an application is denoted by (the denotation of) the proc followed by square brackets enclosing (the denotation of) the arguments: *f* [*first*~3, *last*~*x*+1]; here the ~ symbol binds the value of the expression on the right to the name on the left. There are special ways of writing many kinds of application: *x*+1, *person.salary*, IF *x*<3 THEN *red* ELSE *green*, *x*←7.

Value: An entity which takes part in the computation (i.e., acts as a proc, argument or result) is called a *value*. Values are *immutable*: they are not changed by the computation. Examples: 3, TRUE, "Hello", λ [*x*: INT] IN *x*+3; actually these are all expressions which denote values in an obvious way. The λ-expression denotes a proc value *P*; the name *x* is called a *parameter*. When *P* is applied to an *argument*, the parameter *x* is bound to the argument.

Variable: Certain values, called *variables*, can *contain* other values. The value contained by a variable *v* (usually called the value *of v*) is returned by *v*.VALUEOF, and can change when a new value is *assigned* to *v*. In addition to its results, a proc may have *side-effects* by changing the values of variables. Nearly every non-variable type *T* has a corresponding variable type VAR *T*; values of type VAR *T* contain values of type *T*. Every VAR type has a NEW proc which creates a variable of the type. A variable is usually represented by a single block of storage; the bits in this block hold the representation of its value. A variable may be *local* to a proc, or it may be created by an explicit call of NEW, and referred to by a REF or pointer value.

Group: A *group* is an ordered set of values, often denoted by a *constructor* like this: [3, *x*+1, "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 constructor like this: [*x*: INT~3, *y*: BOOL~TRUE] (or simply [*x*~3, *y*~TRUE], in which the types of the names are the syntactic types of the expressions). If *b* is a binding, *b.n* denotes the value of the name *n* in *b*. Note the difference between binding and assignment: one introduces a new name with a fixed value; the other changes the value of a variable.

Argument: A binding constructor written explicitly after an expression (e.g., *Copy*[*from*~*x*, *to*~*y*]) denotes application of the value *P* denoted by the expression to the value *a* denoted by the constructor, called the argument. *P* is usually a proc, and *a* is a binding, which is bound to *P*'s domain declaration *D* to get the argument which is passed. In making this binding *a* is *coerced*, if necessary, to match the declaration:

> 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 (if possible) into another value which does.

The constructor can also be for a group, in which case the names from *D* are attached to its elements to turn it into a binding.

2.1.2 The type system

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

More precisely, a type is a value which is a binding with two items:

> Its *predicate*, a function from values to the distinguished type BOOL. A value *has* type T if 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 of the type. For *any* expression e, the expression $e.f$ denotes the result of looking up f in the cluster of e's syntactic type ∇e, 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 type) D and range (result type) R has the type $D \rightarrow R$. Every *expression* e has a *syntactic type* denoted ∇e, e.g., the *range* declared for its outermost proc; in general this may depend on the arguments. The value of e always has this type (satisfies this predicate); of course it may have other types as well.

Mark: Every value carries a set of *marks* (e.g., INT or ARRAY; think of them as little flags stuck on top of the value). The predicate HASMARK tests for a mark on a value; it is normally used to write 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 m. 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 value. 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 satisfy the predicate of the domain type; this is a special kind of pre-condition for executing the proc. The proc body can then rely on the fact that the arguments satisfy their type predicates. It must establish that the results satisfy the predicate of the range type; this is a special kind of post-condition which holds after executing the proc. Finally, the caller can rely on the fact that the 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 this: [x: INT, y: BOOL]. If d is a declaration, a binding b has type d if it has the same set of names, and for each name n the value $b.n$ has the type $d.n$. A binding b *matches* d if the values of b can be *coerced* 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 each name is 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 *Ordered* is [T: TYPE, LESS: PROC[T, T]→[BOOL], . . .]. C is a *subclass* of D if (loosely) C includes at least all the [name, type] pairs in D.

2.1.3 Programs

Name: A *name* (sometimes called an identifier) appearing in a program denotes the value *bound* to the name in the *scope* that the name appears in (unless the name is in a pattern before a colon (declaration or binding) or tilde (binding), or after a dot or $). An *atom* is a value that can be used to refer to 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 value to a group or binding value – *GetProperties*[*directory*, *input*];

a *λ-expression*, which yields a proc value – λ [*x*: INT]=>[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 produce a value. Thus an expression is a rule for computing a value. The entire program is a single expression, made up of sub-expressions according to the five constructs above.

Scope: A scope is a region of the program in which the value bound to a name does not change (although the value might be a variable, whose *contents* can change). For each scope there is a binding called ENV (for *environment*) which determines these values. A new scope is introduced (in the kernel) by IN (after LET or λ) or by a REC [...] constructor for a declaration or binding; e.g.,

LET *x*~3 IN *x*+5;

LET REC *Fact*~λ [*n*: INT]=>[*r*: INT] IN (IF *n*=0 THEN 1 ELSE *n***Fact*[*n*−1]) IN *Fact*[4].

The first expression evaluates to 8, the second to 24.

Constructors: Brackets delimit explicit constructors for group, declaration or binding values. They 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_1~e_2 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* has the value *v*, rather than its value in the enclosing scope. Exception: argument bindings are non-recursive.

If *n* is declared in a declaration constructor, then it may not be used in the constructor, unless there is an ordering of the declarations in the constructor such that a name is used only by later declarations. Exception: declared names may be used in the bodies of λ-expressions in the constructor (see § 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 something with 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 (∇*e.n*)[*e*], and *e.n*[more args] is usually (∇*e.n*)[*e*, more args]. Recall that ∇*e* is the syntactic type of *e*.

In all cases you are supposed to think of *n* as some property or behavior associated with *e*; *e.n* denotes that property or evokes that behavior.

2.1.4 Conveniences

Coercion: Each type cluster may contain *To* and *From* procs for *converting* between values of the type and values of other types (e.g., *Float*: PROC[INT]→[REAL]: this would be a *To* proc in REAL and a *From* proc in INT). One of these procs is applied automatically if necessary to convert or *coerce* an argument value to the domain type of a proc: this application is a *coercion*. Each coercion has an 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 serve to prevent unexpected composition of coercions: all are NIL currently, however.

Exception: There is a set of *exception* values. An expression e denotes a value which is either of type ∇e or is an exception. Whenever an exception value turns up in evaluating an expression e_1, it immediately becomes the value of e_1, unless (in the kernel) e_1 has the form e_2 BUT {...}. The {...} tests for exception values and can supply an ordinary value, or another exception, as the value of the BUT expression. An exception value may contain an ordinary value, called the *argument* of the exception, so that arbitrary information can be passed along with an exception.

Finalization: When a variable is no longer accessible, the storage it occupies is freed (automatically in the safe language). Before this is done, a *finalization* proc in the cluster of the variable's type is called to do any other appropriate resource deallocation. Finalization is done by separate processes, and hence must be explicitly synchronized with the rest of the program. The local variables of a proc or other scope may also be finalized (using UNWIND): this is done synchronously (§ 3.4.3A).

Safe: The *safety invariant* says that all references are legal, i.e., each REF *T* value is NIL or refers to a variable of type *T*. A proc is *safe* if it maintains the safety invariant whenever it is applied to arguments of the proper types. If a proc body (λ-expression) is
checked, the compiler guarantees that the proc value is safe;
trusted, the programmer asserts that it is safe (the compiler makes no checks): the proc value is safe:
unchecked, the compiler makes no checks and the proc value is unsafe.
It is best to write checked code whenever possible. However, checked code cannot call unsafe procs (since the compiler then cannot guarantee safety).

Process: Concurrency is obtained by creating a number of *processes*. Each process executes a single sequential computation, one step at a time. They all share the same address space. Shared data (touched by more than one process) can be protected by a *monitor*: only one process can execute within the procs of the monitor at a time. So that each process can know what to rely on, there must be an *invariant* for the monitored data which is established whenever a monitor proc returns or waits. A process can *wait* on a *condition variable* within a monitor: other processes can then enter the monitor. The waiting process runs again when the condition is *notified*, or after a *timeout*.

2.1.5 Miscellaneous

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

Static: An expression whose value is computed without executing the program is called *static*. Literals are static, as are names bound to literals, and any expression with static operands. Proc bodies are never static unless they are inline, and often not then.

Pragma: Some language constructs do not affect the meaning of the program (except possibly to make a legal program illegal), but only its time and space costs: these are called *pragmas*. Examples 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, and an explanation of the relation between the kernel and the current Cedar language, can be found in §§ 2.3-2.8. Since this section is rather formal, you are advised to read the rest of the chapter first, and then return here if you want a more precise definition.

The kernel is subdivided into

A rather austere *core*; anything can be desugared into this, but not very readably (§ 2.2.1).

A set of *conveniences*; with these, readable programs can be written (§ 2.2.2).

Imperative constructs: statements and loops (§ 2.2.3).

Exception handling (§ 2.2.4).

The format of this section interleaves grammar rules which give the syntax of the language with text which gives the meaning. The meaning of the core is given in English. For other parts of the kernel, it is given by desugaring rules which show how to rewrite each construct in terms of others; if rewriting is done repeatedly, the result is a core program, which may invoke some primitives. The meaning of these is also given in English. There is also some English explanation of the desugaring, 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

The Cedar core is a minimal subset of the kernel, barely adequate as a base into which the rest of the kernal can be desugared. In the core, there is syntax only for names, literals, application, λ-expressions, a basic and a recursive binding construction, and syntactic type; everything else is done with primitives. We never write anything in the core, however, except to show the desugaring of a kernel construct. Thus the reader need not struggle with programs in the ugly core syntax.

Many readers may be happy with the kernel definition given in the other sub-sections of § 2.2, and may wish to avoid the formalism of this section.

Table 2 – 1 gives the core syntax (in the first column), together with a comment suggesting the meaning of each construct (in the last column). The meaning is given in detail in § 2.2.1A-G. The middle column gives the syntactic type of each construct. For readability, this is written in the full kernel language, with a few conventions:

a * in front of the syntactic type indicates that it gives less information that one would like. For instance, DDOTP has type DECL→TYPE, which says nothing about the fact that the type is a cross type whose structure matches the structure of the decl.

A parameter to a primitive declared with :: is the type of some other argument; the argument for this type parameter may be omitted in an application of the primitive, in which case it is supplied as the syntactic type of the other argument. For instance, *p*: [*t*:: TYPE, *x*: **t**]→[...] can be applied with *p*[*x*~3], which is short for *p*[*t*~INT, *x*~3].

A bold name is a reference to another parameter, e.g., **t** in the previous example.

In the kernel, a core primitive named *x*DOT*y* is in the cluster of the type of its argument under the name *y*. Thus DDOTP is in the cluster of DECL under the name P, so that *d*.P=DDOTP[*d*] if *d* is a decl.

Syntax	*Syntactic type*	*Meaning*
expression ::=		
n |	∇n	ENV.**n**
literal |	∇literal	
e_1 ▸ e_2 |	(∇e_1.RANGE)[e_2]	-- Standard application. |
λ d_1 => d_2 IN e |	d_1→d_2	-- Standard proc constructor. |
Λ d_1 => d_2 IN e |	d_1→d_2	-- Unchecked standard proc constructor. |
[(n~e), !..]	[(n: ∇e), ...]	-- Vanilla binding constructor. |
FIX d ~ e |	d	-- Recursive binding constructor. |
∇ e	TYPE	-- Syntactic type.
type ::= e	∇e --⇒TYPE--	-- A type is syntactically just an expression.
decl ::= e	∇e --⇒DECL--	-- A decl is syntactically just an expression.
name ::=		
letter (letter | digit)...	(∇ENV).n	-- Appears as an e or in a pattern.
literal ::=		
$ n |	ATOM	-- ATOM literal. |
primitive	∇primitive	
primitive ::=		
ARROW |	[*d*: DECL, *p*: (**d**→DECL)]→[*a*: --arrow--TYPE]	
DOMAIN | RANGE |	*[*a*: --arrow--TYPE]→[*t*: TYPE]	
MKPAIR |	[t_1:: TYPE, *first*: $\mathbf{t}_1$, t_2:: TYPE, *rest*: $\mathbf{t}_2$]→[*v*: $\mathbf{t}_1$×$\mathbf{t}_2$]	
GROUP |	[t_1: TYPE]→[*t*: TYPE]	--*t*⇒TYPE
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*~*v*]
LOOKUP |	[*d*:: DECL, *b*: **d**, *n*: ATOM]→[*v*: DTOB[**d**].**n**]	
THEN |	[d_1:: DECL, b_1: $\mathbf{d}_1$, d_2:: DECL, b_2: $\mathbf{d}_2$]→[*v*: $\mathbf{d}_1$ THEND $\mathbf{d}_2$]	
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	-- DECL⇒TYPE, BINDING⇒TYPE
PATTERN |	TYPE	-- = GROUP[ATOM]
ANY |	TYPE	-- *T*⇒ANY for any type *T*
HIDE |	[*t*:: TYPE, *v*: **t**]→[*h*: HEX]	-- See § 2.2.4
HEX	TYPE	-- See § 2.2.4

Table 2 – 1: The core language

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

A literal denotes a value according to a rule which depends on its syntax. The core has only 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 expression is a name or literal, the value is the value of the name or literal. The remaining cases are discussed in the following sub-sections. Most of these cases define the value of the expression in terms of the 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 a sub-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 v_2, 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 definition of the primitive. The core primitives are defined throughout § 2.2.1, the Cedar primitives in § 4.
>
> v_1 is a closure c (C below), with domain declaration d, body b and environment E. The value of the application is the value of the expression b in the environment
>
> MKBINDD[d, v_2] THEN E
>
> (E below). Note that if the closure was made with Λ, the body must be type-checked when it is applied; a closure made with λ was type-checked when it was made (C below).

∇e_1 must be an arrow type. An application type-checks if ∇e_2 implies ∇e_1.DOMAIN (G below). The type of the application is obtained by applying ∇e_1.RANGE to v_2. In simple cases, ∇e_1.RANGE is a constant. For instance, NOT: BOOL→BOOL has RANGE=λ BOOL=>TYPE IN BOOL. However, the result type may depend on the argument value. Thus

∇MKBINDD.RANGE=λ [d: DECL, v: d.T]=>TYPE IN [b: d]

so that MKBINDD[[i: INT], 3] has type [b: [i: INT]] to go with its value [b~[i~3]].

C. Lambda

The value of a λ-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 when the λ is evaluated.

A λ-expression type-checks if

> d_1 evaluates to a declaration d.
>
> For any x of type d.T, ∇e implies d_2.T in the environment MKBINDD[d, x] THEN E.

A Λ-expression type-checks if d_1 evaluates to a declaration; type-checking of the body is deferred 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 structures are made, as in Lisp, by making pairs of pairs. When we are interested in the leaves of such a structure, we call it a *group* and call the leaves its *elements*. A group has type GROUP[*T*] if all its elements have type *T* or are NIL. A *flat* group is a pair in which *first* is not a group, and *rest* is a flat group or NIL.

The type of a pair is a cross type: MKPAIR[*x*, *y*] has type *T*×*U* iff *x* has type *T* and *y* has type *U*. Cross types are made with MKCROSS, which turns a GROUP[TYPE] (i.e., a group whose elements are types) into a cross type in the obvious way:

MKCROSS[NIL] = NILTYPE

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

MKCROSS[MKPAIR[*x*, *y*]] = MKCROSS[*x*]×MKCROSS[*y*]

Note that MKCROSS of a flat group is flat. CDOTG goes the other way, turning a cross type into a GROUP[TYPE] in which no element is a cross type. Thus MKCROSS is the inverse of CDOTG, but not necessarily the other way around.

E. Bindings

A binding is either NIL, or an <atom, value> tuple, or a <binding, binding> tuple. The primitive MKBINDD constructs a binding from a declaration *d* and a matching value *v*, i.e. (as the type of MKBINDD indicates), one with the type *d*.T. The resulting binding has type *d*, and consists of the 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 INT×BOOL.

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 pattern instead of a declaration, and hence accepts any *v* with the right structure, regardless of the component types.

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, in which the first element of each <binding, binding> tuple is an <atom, value> tuple, and the second element is another <binding, binding> tuple or NIL. The value of b_1 THEN b_2 is another flat binding, obtained by first replacing any tuple <<*a*, *v*>, *b*> in b_2 where *a* is equal to an atom in b_1 by *b*, and then using this binding to replace the final NIL in b_1.

The binding constructor [(*n*~*e*), ...] 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 of d_1 in ENV and *v* is the value of *e* in the environment (LET FIX *d*~*e* IN *d*~*e*) THEN ENV. Of course in general this computation may not terminate; normally the names in *d* occur in *e* only in the bodies of λ-expressions, and in this case it does terminate. The FIX typechecks if ∇*e* in the latter environment implies DTOT[*d*].

F. Declarations

A declaration is either NIL, or an <atom, type> tuple, or a <declaration, declaration> tuple. The primitive MKDECL constructs a decl from a pattern p and a value t of type GROUP[TYPE]. A pattern is a GROUP[ATOM], i.e., either NIL, or an atom, or a pair of patterns; the ATOM elements must all be 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 $t_1 \times t_2$ 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 from t.

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

> DDOTT[NIL]=NILDECL;
>
> DDOTT[<$*n*, *T*>]=*T*;
>
> DDOTT[<d_1, d_2>]=DDOTT[d_1]×DDOTT[d_2]

DTOB is redundant; it converts a declaration to a binding in which each name has the corresponding 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 two declarations just as THEN combines two bindings: $\nabla(b_1$ THEN $b_2) = \nabla b_1$ THEND ∇b_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 $(\forall x)\ T.Predicate[x] \Rightarrow U.Predicate[x]$.

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

$T \Rightarrow$ ANY for any type T.

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

GROUP[T]⇒GROUP[U] iff $T \Rightarrow U$.

$T_1 \rightarrow T_2 \Rightarrow U_1 \rightarrow U_2$ iff $U_1 \Rightarrow T_1$ and $(\forall x\colon U_1)$ $(\lambda\ T_1$ IN $T_2)[x] \Rightarrow (\lambda\ U_1$ IN $U_2)[x]$. Note the reversal of the domains.

$d_1 \Rightarrow d_2$ 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

Table 2−2 gives the syntax and semantics for kernel expressions. Most of this is straightforward sugar. LET adds the binding e_1 to ENV in evaluating e_2. The separate case for b, ... simply allows the [] which normally enclose a binding constructor to be omitted in this case; see below. IF wraps e_2 and e_3 in λ's so that they don't get evaluated; the IFPROC primitive 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 ∇e and applies the result to e. The special LOOPUPC primitive does something special if it finds a proc which takes more than one argument: it splits the proc into one which takes the first argument and returns a proc taking the remaining arguments. This ensures that if $\nabla e.n$ is such a proc P, the expression $e.n[a, b]$ 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. The kind of application may depend on the type of e_1, via the APPLY element of its type; for a proc applied by the standard apply operator ▸, APPLY is the identity. If e_1 is followed by an group rather than a binding constructor, the argument is obtained by binding the group to the declaration which is e_1's domain.

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

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

<table>
<tr><td>expression ::= coreExpression |</td><td></td></tr>
<tr><td>$d_1 \rightarrow d_2$ |</td><td>ARROW ▶ [d_1, λ d_1 =>DECL IN d_2]</td></tr>
<tr><td>λ (| e_1)(| => e_2) IN e_3 |</td><td>-- The domain defaults to [ ], the range to ∇e_3 |</td></tr>
<tr><td>LET e_1 IN e_2 |</td><td>(λ ∇e_1 IN e_2) ▶ e_1.V -- e_1 a binding |</td></tr>
<tr><td>LET b, ... IN e |</td><td>LET [b, ...] IN e |</td></tr>
<tr><td>IF e_1 THEN e_2 ELSE e_3 |</td><td>(IFPROC[∇e_2, e_1, λ IN e_2, λ IN e_3]) [ ] |</td></tr>
<tr><td>e . n |</td><td>IF ∇e⇒BINDING THEN LOOKUP ▶ [∇e, $n]
ELSE IF ∇e⇒TYPE THEN LOOKUP ▶ [∇e.cluster, $n]
ELSE (LOOKUPC ▶ [∇e.cluster, $n]) ▶ [e] |</td></tr>
<tr><td>e_1 [b, ...] | e_1 [e_2, ...] |</td><td>e_1 . APPLY ▶ [b, ...] | e_1 . APPLY ▶ MKBINDD[∇e_1.DOMAIN, [e_2, ...]</td></tr>
<tr><td>e_1 infixOp e_2 |</td><td>e_1 . <u>infixOp</u>[e_2] |</td></tr>
<tr><td>e_1 AND e_2 | e_1 OR e_2 |</td><td>IF e_1 THEN e_2 ELSE FALSE | IF e_1 THEN TRUE ELSE e_2 |</td></tr>
<tr><td>[ ] | [e_1 (| e_2, !..)] |</td><td>NIL | MKPAIR[e_1, [(| e_2, !..)] -- Group constructor. |</td></tr>
<tr><td>PATT p |</td><td>-- Pattern constructor; see the rule for p below. |</td></tr>
<tr><td>[b, !..] |</td><td>b PLUS ... PLUS NIL |</td></tr>
<tr><td>REC [(p : t ~ e), ...] |</td><td>FIX [p, ...] : MKCROSS[[t, ...]]~[e, ...] |</td></tr>
<tr><td>[d, !..] |</td><td>d PLUS ... PLUS NIL |</td></tr>
<tr><td>XX |</td><td>xxxxxx | --Also recursive d maps into this?</td></tr>
<tr><td>statements | simpleLoop |</td><td>-- See § 2.2.3</td></tr>
<tr><td>but</td><td>-- See § 2.2.4.</td></tr>
<tr><td>infixOp ::=</td><td></td></tr>
<tr><td>×</td><td>MKCROSS</td></tr>
<tr><td>PLUS</td><td></td></tr>
<tr><td>THEN</td><td></td></tr>
<tr><td>literal ::= coreLiteral |</td><td></td></tr>
<tr><td>digit digit ... |</td><td>INT -- Numeric literal, giving the decimal representa</td></tr>
<tr><td>declaration ::=</td><td>-- A d is not an e; a d must be before ~ or after LET or DECL.</td></tr>
<tr><td>p: t |</td><td>MKDECL[PATT p, t] |</td></tr>
<tr><td>[(p: t), ...]</td><td>[p, ...]: MKCROSS[[t , ...]] -- to separate names and types</td></tr>
<tr><td>binding ::=</td><td>-- Only the [...] form is an e; a b must be written after LET. |</td></tr>
<tr><td>p ~ e |</td><td>MKBINDP[PATT p, e] |</td></tr>
<tr><td>d ~ e |</td><td>MKBINDD[<u>d</u>, e]</td></tr>
<tr><td>pattern ::=</td><td>-- Note: a pattern is not an e; it can appear only before ~ or :, or after PATT in the kernel.</td></tr>
<tr><td>n |</td><td>-- PATT n = $n</td></tr>
<tr><td>[p_1, ...]</td><td>-- PATT [p_1, ...]=[PATT p_1, ...]</td></tr>
<tr><td>primitive ::= corePrimitive |</td><td>-- Fill in types</td></tr>
<tr><td>LOOKUP | LOOKUPC |</td><td></td></tr>
<tr><td>PLUS |</td><td></td></tr>
<tr><td>IFPROC |</td><td></td></tr>
</table>

The precedence of operators in e is: (highest) [ ], ▶, infixOps (all the same), BUT, IN (lowest). All are left associative.

Table 2 – 2: Kernel expression syntax and semantics

Constructors

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

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

Note that these constructors do not nest, so they can only be used to build flat values. The only exception is that a d can be [(*p*: *t*), ...]. This is intended for the *d*~*e* form of binding; e.g., if *DivRem* returns two INTs, you can write [*d*: INT, *r*: INT]~*DivRem*[...] instead of [*d*, *r*]: INT×INT~*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′~(λ IN { **statements**; loop′[ ] })] IN loop′[ ] -- Only an exception (such as EXIT) will terminate the loop.

Each *e* in the statements must evaluate to VOID, which is a distinguished null value; this is to catch mistakes like writing *x*+1 as a 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 supposed to use an exception to get out of this loop; Cedar provides a number of convenient ways to do this, such as EXIT and RETURN.

2.2.4 Exceptions

An exception is treated as a special value returned from an application. The exception value contains an exception *code* and an *args* value which may be of any type. When an application sees an exception value, it immediately abandons the application and returns the exception value; thus application is *strict*. There has to be some way to stop this, or the first exception would be the value of the program. The HIDE primitive takes any value and returns a variant record of type HEX. It 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 arguments in 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 the args; it is the *domain* of the exception, and (∇EXCEPTION[*T*]).DOMAIN = *T*. An exception value is 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 IN **butChoice** ELSE ... ELSE UNHIDE[v'] ELSE ERROR)
butChoice ::= e_1 => e_2 |	IF selector' = e_1 THEN LET MKBINDD[∇e_1.DOMAIN, h'.args] IN e_2 |
e_1 , e_1, !.. => e_2 |	IF (selector' = e_1) OR ... THEN e_2 |
ANY => e_2	IF TRUE THEN e_2

A BUT expression evaluates *e*. If it is a normal value, that is the value of the BUT. If it is an exception, each butChoice in turn gets a look at it. If one of them likes it, then it supplies the value 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 matches *code* in the exception, then *args* in the exception is bound to the domain of the code, and e_2 is evaluated in that environment. If there is more than one, then e_2 is just evaluated in the current environment. An ANY butChoice matches any exception, but of course doesn't bind the arguments.

2.3 Values and computations

A computation in Cedar is the evaluation of an expression in some environment. This section describes the kinds of values which can be computed by Cedar programs, and the basic mechanisms for doing computations.

2.3.1 Application

The basic mechanism for computing in Cedar is applying a proc to argument values. A proc is a 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 to inspection, but which is defined in § 4).
>
> By a *closure*, which is the value of a λ-expression whose body in turn consists of an expression, which may contain further applications of procs to arguments, e.g., λ [*x*: INT] IN *x*+3. When a closure is applied, the *parameters* declared after the λ are bound to the arguments, and then the *body* after IN is evaluated in the new environment thus obtained.

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

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

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

> evaluate the λ-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 (with different results in general, though not in this case).

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

Actually there may be one more step in an application. If an argument doesn't have the type expected by the proc, the argument is *coerced* to the proc's domain type if possible. If no coercion 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 named. E.g., *OpenFile*: PROC[*name*: ROPE, *mode*: *Files.Mode*] takes a binding with two values named *name* and *mode*. It might be applied like this: *OpenFile*[*name*~"Budget.memo", *mode*~$*read*]. If the names are missing, there is a *positional* coercion which supplies them left-to-right, see § 2.3.6. There is also a *defaulting* 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 APPLY proc for 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 primitives cannot be the values of expressions, and can only be applied by writing some other construct. The desugaring rules show how large parts of the Cedar syntax denote various special kinds of application. In each case, the meaning is defined by the standard meaning of application and the 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. Future 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 names, 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 value.

Note that a functional expression must not depend on values contained in variables bound to its free names. Thus, *v*.VALUEOF is *not* functional.

A proc is a *function* if every application of it is functional. It doesn't matter when or how many times a function is applied; the order of evaluation doesn't matter for functions. Thus Cedar functions can be thought of as mathematical functions for many purposes. Note that a constant can 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 formal distinction, either in syntax or in the type system, between functions and procs. However, it does 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 λ-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 statement unless they don't affect each other; if this is done the effect of the program is unpredictable.

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

Because the values of variables constitute the state, it is only the existence of variables that allows non-functional procs to exist. In particular, the VALUEOF proc which returns the value of a variable is non-functional (because its result depends on the state), and the ASSIGN proc which changes the 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 expression in the program is a value. Thus numbers, arrays, variables, procedures, interfaces, and types are all values. In the kernel language, all values are treated uniformly, in the sense that each can be:

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 for computation and λ-expressions as the basis for program structure. In addition, each particular kind or *type* of value has its own primitive operations. Some of these (like assignment and equality) are defined for most types. Others (like addition or subscripting) exist only for certain specific types (numbers or arrays). None of these operations, however, is fundamental to the language. Formally, assignment or equality has the same status as any operation on an abstract type supplied by its implementor; thus INTEGER.ASSIGN has the same status as *IO.GetInt*. In practice, of course, special syntax is usually used to invoke these operations, and the implementations are not Cedar programs open to inspection by the editor or debugger. A complete description of the primitives supplied by the language can be found in Chapter 4, organized by the type of the main operand. Table 4−5 is an alphabetized index of these descriptions.

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

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

2.3.3 *Variables*

Certain values, called variables, can contain other values. A variable containing a value of type *T* has type VAR *T*. If the variable doesn't allow the value to be changed, the type is READONLY *T*; this is not the same as *T*, because there may be a VAR *T* value which is the same container. The value contained by a variable (usually called the value *of* the variable) can be changed by assigning a new value to the variable. The set of all variables accessible from the *process array* constitutes the state of the computation; these are all the variables which can be reached from any process, and a variable which cannot be reached cannot affect the computation. Note that a variable value is a *container*, which like all values is immutable; it may help to think of it as (the address of) a block of storage. The *contents* of a variable can be changed by assignment. Thus the value *of* a variable 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*→[]]. This representation is not used in Cedar, but it clarifies the way in which variables fit into the type system: VAR *T*⇒VAR *U* only if *T* and *U* have the same predicate, because the *Get* proc requires *T*⇒*U* and the *Set* proc requires *U*⇒*T*. 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*, so that a variable can be passed without fuss as an argument to a proc which expects a value.

Restriction on variables: In current Cedar, variables generally cannot be passed as arguments or results. The only exception is that an interface can declare a variable (called an exported variable) for which an implementation supplies a value; this is normally written *x*: VAR INT in the interface, but for historical reasons it is also possible to write just *x*: INT. Certain primitives (e.g., dereferencing a REF or POINTER) return variables, a variable can (indeed, must) be passed as the first argument to ASSIGN, and a variable can be bound to a name by a declaration in a LET or block (LET *x*~INT.NEW IN ... binds a VAR INT value to *x*). For the most part, however, a program which wants to handle variables 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 representation of its value. All the built-in VAR types are represented in this way. A variable *u* *overlaps* another variable *v* if assigning to *u* can change the value of *v*. The primitive ASSIGN procs have the property 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 unchecked program has given overlapping blocks of storage to the two variables (if *u* and *v* have different 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. Such a composite value is called a group, and the simpler values are its *components* or *elements*. Thus [3, *x*+1, "Hello"] denotes a group, with components 3, the value of *x*+1, and "Hello". The main use of explicit groups is for passing arguments to procs without naming them (these are sometimes called *positional* arguments). This is done by binding the group to the declaration which is the domain type of the proc; the result is a binding which is the argument the proc expects. Thus, with *P*: [*x*: INT, *y*: REAL]→[. . .], the application *P*[2, 3.14] is sugar for *P*[[*x*: INT, *y*: REAL]~[2, 3.14]], 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* has type *U*, then [*x*, *y*] has type $T \times U$. Thus for syntactic types, $\nabla[e_1, e_2, ...] = \nabla e_1 \times \nabla e_2 \times ...$ The × type constructor is associative, and type implication (§ 2.4.2) extends to cross types elementwise. If the T_i are types, there is a coercion called MKCROSS from $[T_1, T_2, ...]$ to $T_1 \times T_2 \times ...$; because of this, the explicit cross type is usually not needed.

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

2.3.5 Bindings

A binding is a group in which each component has a name. Thus, it is an ordered set of [name, value] 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*: BOOL], its argument must be a binding such as [*i*~3, *b*~TRUE]. The application then looks like this: *P*[*i*~3, *b*~TRUE]. A binding argument is sometimes called a *keyword argument list*. See the 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 beginning 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 extracted from the set using dot notation. Thus if *b* is the binding [*i*~3, *b*~TRUE], the value of *b.i* is 3. In 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 argument of 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 called *coercions* which may be applied to the values. This usually happens when the arguments of a proc 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 the names from *d* 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. There is an error unless the names match exactly. The remaining coercions are done on individual items, *n*: *t* from the declaration and the corresponding *n*~*v* from the binding. If *v* has a type implying *t*, all is well. Otherwise, if there is a sequence of coercions from the type of *v* to *t*, these are applied to *v*. If no such sequence exists, there is an error. In particular, there is a coercion from OMITTED to the *default* 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 in § 4.14.

An important special case is constructors for record and array values. A record type has a construction proc; e.g.,
  R: TYPE~RECORD[*a*: INT, *b*: REAL←0.0]
has a proc *R*.CONS of type PROC[*a*: INT, *b*: REAL←0.]→[*R*]. Thus *R*.CONS[*a*~2, *b*~3.1416] constructs a record value. There is also a coercion from BINDING to the particular declaration *RB* which is the domain type of *R*.CONS, so that
  rl: *R*←[*a*~2, *b*~3.1416]
is short for
  rl: *R*←*R*.CONS[*a*~2, *b*~3.1416].
Composing the positional coercion from GROUP to *RB* with *R*.CONS makes
  rl: *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 don't 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 program which the compiler can verify. It also discusses the role of types in organizing the names of the 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 whole number between 0 and 10 represented in a single machine word. A value which has the property is 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 accepts a single value (of any type) and returns TRUE iff the value satisfies the assertion. In principle the predicate can be applied to any value at runtime, but in practice a lot of optimization is done by the compiler.

- A type contains a collection of named procs (and perhaps other values) related in some useful way. Most often, the procs of type *T* take a value of type *T* as their first argument. 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 easy to 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 rules are enforced about what kind of values are in the binding. However, the idea is that the cluster 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 important.

The predicate and the cluster serve rather different purposes:

> The predicate provides the basis for type-checking (§ 2.4.2). The most important function of type-checking is to guarantee the integrity of abstract data types; this is done with basic predicates called *marks* (§ 2.4.3).
>
> The cluster provides the basis for convenient naming of a large collection of procs and other values (§ 2.4.4). Clusters are organized into a hierarchy of *classes* (§ 2.4.5).

Like everything else which can be named, a type is a value. Hence there is nothing special about 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 *T* is not rebound). Furthermore, with two exceptions, all type expressions are functional: identical type expressions in the same scope denote the same type value. The exceptions are the record and enumeration type constructors, which make a distinct type each time they are used (by constructing a new mark; see § 2.4.3).

Restriction on uses of types: Current Cedar has a number of restrictions on the use of TYPE values, given in § 4.8.

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 of procs. In 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 $i \geq 0$ and $i \leq 10$ just before the expression *a*[*i*] is evaluated. This is called a *bounds check*; if it fails there is an exception called *Runtime.BoundsFault*. Where did this check

come from? Note that $d[i]$ is short for ∇a.APPLY[a, i], and ∇a.APPLY is SUBSCRIPT, the subscript procedure for ARRAY [0..10] OF INT. The type of SUBSCRIPT is PROC[*array*: VAR ARRAY [0..10] OF INT, *index*: [0..10]]→[VAR INT]. So when i is passed as the *index* argument, the declaration of SUBSCRIPT says it must have the type [0..10]. The predicate for this type is

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

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

The type system is designed, however, so that most assertions can be checked *statically* (i.e., *proved*), by examining the text of the program without running it. Static checking has three obvious advantages:

> It reports any errors after a single examination of the program, leaving none (of this kind) to be discovered later in Peoria.
>
> It introduces 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 declarations must be simple enough that the compiler can reliably prove or disprove them.

The proofs done for type-checking have exactly the same form as program correctness proofs based on preconditions and postconditions. Consider a proc whose value is the λ-expression

λ [x: T]=>[y: U] IN e.

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

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

argument⇒domain⇒range⇒result

The result in turn will be the argument of another application. In this way the proof is extended to 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 done by observing that every expression has a unique *syntactic type* U, which is the type of every evaluation of that expression; e.g., an application always has the range type of its proc (see below for a more detailed discussion of syntactic type). If every value of type U has type T, we are done. Hence the usefulness of *type implication*. One type implies another, $T \Rightarrow U$, iff $(\forall x)\ T[x] \Rightarrow U[x]$; sometimes we say that T is a *sub-type* of U. If two types are equal, each implies the other. However, there are many other useful cases of implication. For 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 possible forms of 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, 'A has 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 declaration

of a λ-expression, and we have already seen how to ensure that the *n*'s value has 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 that *e* has type *T*.

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

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

There is one more link in the chain. An application *f*[*x*] has an arbitrary expression for *f*, not necessarily a λ-expression. The requirement is that *f* must have a proc type, say *D*→*R*; *D* is the domain type and *R* the range type. Since the type of λ *D*=>*R* IN *e* is *D*→*R*, satisfying the precondition *D* for the application is the same as satisfying the precondition *D* for the λ-expression, and similarly in reverse for the postcondition. The value of *f* may be a primitive rather than a closure obtained from a λ-expression. In this case, the implementation of the primitive can still depend on the precondition and must still establish the postcondition, but since the implementation 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 machine's 32-bit add instruction.

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 an application then depends on the argument *values*.

Restriction on dependent proc types: In current Cedar only a module has a type whose range depends on its argument values; the type returned by an interface, or the interfaces exported by an implementation, may depend on the interface and implementation parameters.

As a by-product of the type-checking proof rules just given, a *syntactic type* is derived for every expression *e* in the program. It is denoted by ∇*e*, 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 λ the obvious proc type;

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

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

In order to carry out the proofs described above, the compiler must either compute the values of all types, including those denoted by complex expressions such as ARRAY [*i..j*] OF INT, or it must be able to prove the equality of unevaluated type expressions. For the most part, current Cedar requires the former approach; hence a type expression must have value which the compiler can 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 predicates are worth all this fuss. The reason is simple: they are the basis for authenticating values of an abstract type, so the implementation can be sure that it is working on properly formed values. Suppose you are the implementer of an abstraction, e.g., *Table*. You provide operations to *Lookup* a key in the table, to *Insert* a [key, value] pair, and to *Enumerate* the items in the table. A *Table* is implemented as a REF to a record containing a sorted array *a* of items and an INT *n* which gives the number of

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 size of the array. They will not work properly if these assumptions are not satisfied, and indeed they may try to subscript the array with an out-of-bounds index or to violate other requirements of the abstractions they depend on.

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

```
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 REAL, and the assignment to *r*↑ stores the four bytes which represent 3.14159 into that block. If somehow a REF BOOL finds its way into *r*, the assignment will still store four bytes, since it doesn't know any better. But the REF BOOL points to a two-byte block; the other two bytes that will be modified belong to some unrelated variable, which will be clobbered without warning.

The second example is scarier because the consequences of the bug seem more unpredictable. In both cases, however, the fundamental problem is the same: even if the implementation is correct, the wrong thing happens because it is given an improper value to work on. Or to make the same point in different words, the implementation cannot be held responsible for bad results from one of 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 operation, there must be a way to *authenticate* a value of the abstract type. In Cedar this is done by placing a *mark* on the value; think of it as a little flag stuck into the value. The mark uniquely identifies the abstract type, and authority to affix it is under the control of the implementation. A correct implementation will mark only values which have the properties needed for a representation of an abstract value, and if no one else can affix the mark, the implementation can be sure that every value with the mark has the desired properties.

A mark can be thought of as an abbreviation for an assertion or *type invariant* which characterizes a proper abstract value, such as *Table* or REF REAL. Such an assertion can be quite complex. In the *Table* example, it would say that the representation is a record of the proper form, that n is less than the array size, and that the first n array elements are sorted. In the REF REAL example, it would say that the address points to a block of storage such that at least the first four bytes don't overlap any other blocks. Such assertions are not easy to write down formally, and proving them is certainly beyond the power of any existing program. So the abbreviations are not a mere convenience, but a 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 for the presence of the mark *m*. *Affix* adds the mark to a *Rep* value.

Restriction on marks: MARK, UNIQUEID, CREATEMARK, HASMARK and TYPEFROMMARK are not accessible in current Cedar. Record and enumeration 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 representation type, and obtain a new mark *m*. TYPEFROMMARK[*m*] with an appropriate cluster added is the new abstract type. The implementation must use *Affix* to mark only values which satisfy the properties it demands.

The type returned by TYPEFROMMARK[*m*] has the predicate

λ [*x*: ANY]=>[BOOL] IN HASMARK[*x*, *m*]

and an empty cluster. Except for subranges and bound unions, all types in current Cedar have a predicate of this form. The built-in types (INT, BOOL etc.) come with such predicates, and the built-

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 UNIQUEID, and 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 CREATEMARK directly, we need not explain how to prevent forgery of UNIQUEIDS. Future versions of Cedar will address this problem.

In current Cedar you make a new abstract type by declaring it as an opaque type in an interface:
  T: TYPE[ANY]
This generates a new mark, and declares *T* to be a type which has that mark. You get such a type by explicitly painting some other type, normally in an implementation which exports *T* to the 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 variable can be referenced by a REF, and in particular by a REF ANY value. The type of a REF ANY value can be tested at runtime using the primitive
    ISTYPE: PROC[*x*: ANY, *U*: TYPE]→[BOOL]
If ∇*e* is REF ANY and *RT*=REF *T*, then the value of ISTYPE[*e*, *RT*] is TRUE iff the predicate for *T* just tests for mark *m*, and *x*↑ has the mark VAR *m*. ISTYPE is described in detail in § 4.3.1, along with the WITH ... SELECT construct and the NARROW primitive, which are more powerful operations built up from ISTYPE.

For other values, there is no mark actually stored; instead, types must be computable statically using the methods described in the last section. The *AMTypes* interface, however, gives a way to 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 the value actually carries. We arrange for this by imposing a partial order on the marks, and requiring that:

> 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 constructors, or 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 *T*⇒READONLY *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 dealing with values of the type. This is done by putting these procs into the type's cluster. The cluster is simply a binding which is part of the type value (the predicate is the other part). There are no rules enforced about what goes into the cluster. However, there is a special dot notation which makes it desirable to populate *T*'s cluster with procs which take a *T* as their first argument. The usual effect is like this: *t.n* is sugar for ∇*t.n*[*t*], and *t.n*[other args] is sugar for ∇*t.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 *P*, then

the proc can be applied by an expression like $\iota.P[3]$, which is sugar for $\nabla\iota.P[\iota, 3]$. The name P is looked up only in T's cluster, not in the current scope. If Q: [T]→[INT] is also in the cluster, it can be applied with $\iota.Q$, which is sugar for $\nabla\iota.Q[\iota]$.

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

Dot notation can also be used to obtain values from a binding or from the cluster of a type without any application: *T.P* would be the proc named *P* in the previous example. The possible cases of dot notation in current Cedar are described in detail in § 4.14.

Restriction on constructing clusters: There is currently no way to explicitly construct clusters. The built-in types and type constructors have clusters; they are described in detail in § 4. In addition, there is a clumsy way to provide a cluster for an opaque or record type in an interface: *every* proc name in the interface is put into the type's cluster. For a record, the procedures supplied by the record constructor are also in the cluster, and they win if there are name conflicts. There is one of these clusters for each type in *each* imported interface value; if a module imports more than one value of the same interface, 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 [x: INT, y: REAL]. All the relationships among types, and between types and values, are carried over elementwise to decls and bindings; the elements are matched up by name rather than by position. 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 for making decls, MKDECL, takes a pattern and a type. Thus MKDECL[PATT[x, y], INT×REAL]=[x: INT, y: REAL]. In general, a pattern is one of NIL, a simple name, or a pair of patterns, just like a Lisp S-expression. Similarly, a type argument to MKDECL is one of NIL, a type, or a *cross type*. The type must decompose in a way which matches the pattern. Normally, as in Lisp, we deal only in flat patterns, where the first element of a pattern is always a name. Such flat patterns are conveniently denoted by constructors of the form [x, y, ...]. The reason for defining things in terms of pairs is that it makes it much simple to write down precise rules for the semantics, using structural induction on the values.

The main use of a decl is to type-check a binding. The basic binding constructor is MKBINDD[d, e], where d is a decl and e is matching group or binding. If e is a binding, then its structure and names must match the structure and names of d, and each element of e must have the type demanded by the corresponding component of d, after a possible coercion. Thus MKBINDD[[x: INT, y: REAL], [x~3, 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. Even though the resulting binding is the same as the argument, the type-checking is essential.
>
> There may be coercions involved, so that the resulting binding is not the same. Coercions on the component values are discussed in § 2.6.1. There are also coercions on the binding 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, as discussed in § 2.3.6. Thus in MKBINDD[[x: INT, y: REAL], [3, 3.14]] the second argument is coerced to [x~3, 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 is better to use the MKBINDP primitive to bind the value directly to a pattern. The syntactic type of the result is the decl whose type is the syntactic type of the value. Thus [x~3, y~3.14] is short for

MKBINDP[PATT[x, y], [3, 3.14]]; its syntactic type is MKDECL[[x, y], ∇[3, 3.14]]=MKDECL[[x, y], INT×REAL]=[x: INT, y: REAL].

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

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

{x: INT; y: REAL; z: BOOL~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: BOOL~TRUE IN S

Anomaly about uninitialized names or variables: In Cedar the names in a block are introduced recursively, so that the d's and b's can refer to each other. It is possible for a binding or type to refer to a value which has not yet been 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 the cluster is just a binding, it is characterized by its type, which is a decl. When used for this purpose, a decl is called a *class*. See § 4.1 for further discussion of classes, and an enumeration of the primitive classes 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 application which involves sub-expressions: the proc being applied, and the arguments. The concrete syntax treats certain kinds of expressions specially: modules, blocks (which introduce new variables and return no value), and statements (which return no value). All desugar into simple expressions, 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 contexts in 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 in which 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 for *n* 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 (∇*e*).*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 value 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 the type of *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 in the constructor denotes the value bound to *n* in that constructor; see § 2.5.6 for details. In the kernel 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 atom has a *print name* which is a rope (an immutable sequence of CHARs). A name following a $ is an *atom literal*: $*n* denotes the atom with print name *n*. Other properties of atoms are described in § 4.5.1A.

Caution on names: 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 distinguishes 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*: *M*, and the second *M* denotes its binding in the surrounding scope (i.e., the binding supplied 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 § 4.7.1A for details.

2.5.3 Scope

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

2.5.4 Constructors

The kernel has constructors, denoted [...], to make expressions which denote group, decl and 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 the form *p*~*e* or *d*~*e*. The presence of the ~ distinguishes it from the others. Here *d* is a decl element (*not* a declaration), and *p* is a pattern, in which the names are being defined rather than evaluated.

A decl constructor is a list of *decl elements* (d in the syntax) of the form *p*: *t*. The presence 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 are *not* expressions, although constructors *are* expressions.

Constructors are useful for making decls and bindings where the names are literal. This is the normal case, and in fact the only case in current Cedar. If you want to make them out of other decls, for instance to bind an expression to a decl which is the value of a name *dn*, you cannot use a constructor; [*dn*~*e*] would bind the value of *e* to the name *dn*, not to the decl which is its value. 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] in the syntax).
>
> Binding constructors for arguments in an application, or as an expression alone if a record or array value is needed (argBinding[27] in the syntax).

2.5.5 Recursion

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

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

2.6 Conveniences

The facilities described here are not fundamental, but they are of great practical importance.

2.6.1 Coercion

A coercion is a proc which is automatically applied under some circumstances to map a value of one type T (called the *source*) to a value of another type U (called the *dest*), e.g. from [0..5) to INT. Coercions are obtained from the clusters of the types involved. The coercion mechanism adds no new functionality, since the programmer could always write the applications himself, but it is important in concealing some of the distinctions made by the type system when they are distracting rather than helpful.

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

If $T \Rightarrow U$ it is sometimes natural to think in terms of a coercion from T to U that is implemented by the identity function. In fact, implication is stronger than that, since it propagates through many type constructors, including PROC, while coercion does not. Implication is discussed in § 2.4.2 and § 4.12.

There is a rather general rule for finding coercions from the clusters of types, though it is not of much practical importance in current Cedar, since there is no way for the user to define coercions. The rule goes like this. Each cluster may have a *From* item and a *To* item. *T.From* should consist of pairs with type [*tag*: ATOM, *proc*: $T \rightarrow U$], and *T.To* of pairs with type [*tag*: ATOM, *proc*: $U \rightarrow T$]. Ignore the *tags* for the moment. Consider the binding *n*: $U \sim e$, where $\nabla e = T$, and $T \Rightarrow U$ is false. For each proc *P* in *T.From* or *U.To* we try *n*: $U \sim P[e]$.

> If *P*: $T \rightarrow V$ is in *T.From*, it maps *e* to a value of type *V*, and we have to bind *n*: $U \sim P[e]$. If $V \Rightarrow U$ we are done: otherwise we can recurse on this sub-problem.
>
> If *P*: $V \rightarrow U$ is in *U.To*, we have to bind *m*: $V \sim e$. If $T \Rightarrow V$ 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 terminate when all paths have been explored, and a particular path can be abandoned when a type appears on it for the second time. Since the search is done statically (by the compiler), and since the results of an attempt to coerce *T* to *U* can be cached, the time required for the search is not a problem.

There are two obvious difficulties with this scheme. First, it may transform erroneous applications into legal ones, by coercing an argument in ways not intended by the programmer. Second, more than one path of coercion procs may exist, and different paths may give different results. The second difficulty can be avoided, and the first minimized, if every coercion proc *P* is chosen so that it has a (partial) inverse, and $P^{-1}[P[x]] = x$ for all *x* in *P*.DOMAIN. This says that a coercion does not lose information, and that different paths give the same answer. Sometimes this is not feasible, e.g. for the *narrowing* coercion from INT to [0..5). The following rule gives the builder of clusters control 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 have different 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 groups and bindings.

2.6.2 Exceptions

The basic idea behind exceptions is to extend the value space, so that it includes not only ordinary values, but also a set of *exception values*. An exception value has the special property that whenever it appears in an application, it becomes the value of the application, so that it propagates up through the control stack of the program until it finally becomes the value of the whole program. Of course this isn't always what is wanted, so there is a special HIDE construct which is not an ordinary application, but takes its argument value, ordinary or exception, and bundles it in a variant record which is a normal value. Then ordinary code can be used to test for the exception and take appropriate action. This construct is sugared to give distinctive ways of *catching* an exception: in the kernel with BUT (§ 2.2.4), and in Cedar with ENABLE, EXITS and REPEAT (§ 3.4.3). Cedar has two kinds of exception: GOTO labels and ERRORS, which must be raised and caught separately, and have slightly different semantics.

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

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

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

Cedar also has *signals*, which historically were viewed as a kind of exception but now have a very different interpretation, as a way of obtaining dynamic rather than static scoping for names. They are discussed in § 3.4.3A.

2.6.3 Finalization

This subject is discussed in § 3.4.3A.

2.6.4 Concurrency

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

2.7 Miscellaneous

The different kinds of allocation are discussed in § 4.5. Static values are defined in § 3.9.1.

2.7.1 Pragmas

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

> INLINE, which causes a proc body to be expanded inline when it is applied. See § 3.5.1 for details.
>
> PACKED, which causes array components that fit in 8 or fewer bits to be packed, at the expense of more expensive code to access them (§ 4.4.2).
>
> CHECKED, which forbids application of unsafe procs in a block, and adds runtime checking for some primitive procs which are otherwise unsafe—in particular, narrowing to a subrange, and assigning a proc (§ 3.4.4).
>
> PRIVATE, which forbids access to items in an interface or instance except to modules which EXPORT (or SHARE) it (§ 3.3.6).
>
> MACHINE DEPENDENT, which allows positions of record fields (§ 4.6.1) and representation values for enumeration elements (§ 4.7.1A) to be specified (strictly, it is the absence of MACHINE DEPENDENT that is the pragma, since the positions or representation values are legal only when it is present.)

2.8 Relations among groups, types, bindings and declarations

Cedar has are four closely related basic ways of building product values from simple values (all are 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 ×-*type* is the type of a group (if x: T and y: U then $[x, y]$: $T \times U$) (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 § 2.4.5).

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

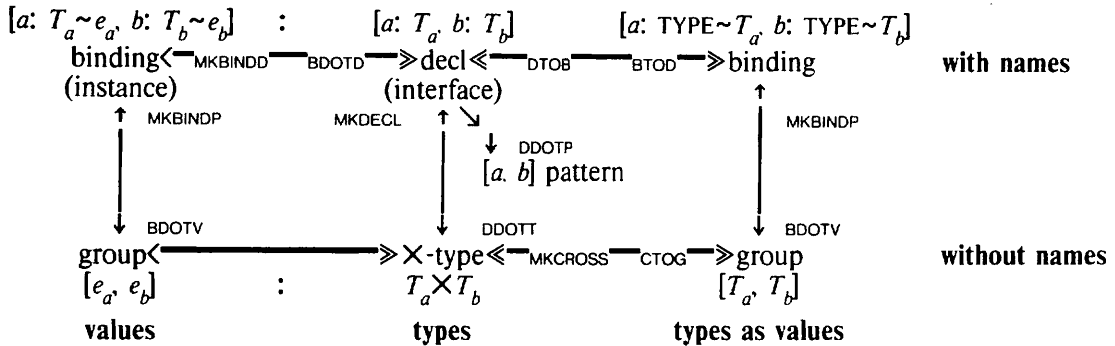

Figure 2−1: Relations among groups, types, bindings and decls

2.9 Incompatibilities with current Cedar

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

> Type expressions in Cedar do not have the same syntax as ordinary expressions and cannot appear in the same contexts, for the following reasons:
>
> > The use of ← for specifying a default value for a type *vs* its use for assignment.
> >
> > The use of {} for enumeration types *vs* its use for a block.
> >
> > The use of parentheses and brackets to specify subranges
> >
> > •The use of adjectives for variants (*red Node*).
>
> Target type overloading for union constructors ([*rator*~$*plus*, *rands*~*binary*[...]]), and •enumeration literals (*red* instead of *Color.red* or $*red*) is incompatible with the kernel's simple rules for the meaning of names.
>
> •In addition to writing *n*: *t*~*e* or *n*~*e* for a binding, you can also write *n*: *t*=*e* (in a module header or block) and *n*: *e* (in an argBinding). The most unfortunate consequence is that a Cedar argBinding can look like a kernel decl constructor!

It is now possible to avoid all the conflicting constructs except the relatively harmless ones: ← for 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 informal explanation of the meaning of each construct, and a precise desugaring of each construct into the kernel language defined in § 2. The desugaring, together with the definitions of the kernel primitives used in it, are the authority for the meaning; the informal explanation is just for your reading pleasure. However, paragraphs beginning *Anomaly* or *Restriction* document properties of Cedar not captured in the desugaring. The primitive procs and types of Cedar are specified in § 4.

In addition to the grammar rules and desugaring, there are examples for each construct. These are intended to illustrate the constructs and do not form a meaningful program. The Cedar Manual has longer examples which do something interesting, and also illustrate the use of the standard Cedar 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 omits a 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 systematically with the rules of the grammar, explaining peculiarities of the syntax and giving the semantics:

§ 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 types.

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, desugaring, and commentary of 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 also be ",", ELSE, IN, or OR, or it may be absent. If the separator is ";", a trailing ";" is optional.
item; !.. is just like item; ... but there is at least one occurrence.
A terminal is a punctuation character other than bold **()?|**, or any character underlined, or a word in SMALL CAPS. Note that [] and {} are terminals, and do *not* denote optional occurrence and repetition as they do in many other variants of BNF.
The rules are numbered sequentially.
Special symbols mark constructs with special properties:

- † = unsafe;
- • = obsolete;
- ‡ = machine-dependent;
- ★ = efficiency hack.

The grammar is written so that a non-terminal never expands to the empty string. When an element 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 they are 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 order to read the grammar.

With the exception of these abbreviated non-terminals, each use of a non-terminal is cross-referenced with a small superscript number[59], unless the non-terminal is defined in one of the next few rules. If a non-terminal (other than e, t or n) is used in more than one rule, then all the rules 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 duplications do not lead to syntactic ambiguity. In most cases they are harmless, since the symbol has essentially the same meaning in each case, and the rules are separate only for greater readability, to highlight an unusual use of a construct, or for historical reasons. In some cases, however, the symbol has quite different meanings in different rules. These are marked on the left as follows:

- ⊗ In the rules whose numbers are marked with * the symbol has a different meaning than in the others, and confusion is quite possible. The programmer should beware.
- ○ In the rules whose numbers are marked with * the symbol has a different meaning than in the others, but the context is sufficiently clear that confusion is unlikely.
- • The rules whose numbers are marked with • are obsolete and should be avoided.

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

	Symbols	Rules	Explanation
○	( )	19, 25, *51.1, *54	expr, subrange, *position in record or enumeration
○	[ ]	19, 25, 26, 37, 43, 51	constructor/built-in/funnyAppl, subrange, application, typeName, fields, mdFields
○	{ }	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, 27.1, 30, 33, 35	See note in § 3.2.
○ •	:	1, 2, 3, 5, •7, 11, 13, 18, •27, 33, •34, 51, *51.1, 53	introducing names with types, except *51.1=position, •7=open, •27=argBinding •34=withSelect
	.	19, 37	dot notation for e is repeated for types
○	..	25^{x4}, *51.1	subrange, *position
○	*	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 choices^{x4}, unionTC
⊗	←	14, 16, 18, 21, *55	s, e←STATE, iterator, e, *defaultTC
○	~	2, 3, 13, 20, *22, *27	interface,implementation,b,argBinding,*unaryOp,*relOp
	~~	7, 34	open, withSelect
⊗	ANY	*9, 40, 43	*enable, variableTC, fields
⊗	CODE	*13, 23	*new exception, convert t to e
	ENDCASE	31, 52	select endChoice, unionTC
○	ERROR	*19, *24, 41.1	*expression, *funnyAppl, transferTC
	IN	18, 22	iterator, relOp
	LONG	38^{x2}, 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	select, safeSelect, withSelect, unionTC
	SHARES	2, 3	interface and implementation
○	SIGNAL	*24, 41.1	*funnyAppl, transferTC
	TRASH	27^{x2}, 55^{x2}	argBinding, defaultTC
	TRUSTED	6, 13	block, machine code
⊗	USING	1, *5	directory, *locks
○	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 into comments describing the meaning in English. This is a purely *textual* transformation; i.e., it is done on the *text* of the program, not on the *values*. The rewriting is done one rule at a time; a single step of rewriting involves elements from exactly one rule. The desugaring is specified by slightly informal but straightforward rewriting rules, in which:

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

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

Every occurrence of **e** (expression) and **t** (type) in the desugaring is implicitly parenthesized, so that the desugared program parses as the rewriting rule indicates. To reduce clutter, these parentheses are not written in the desugaring rules.

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

Examples: the following rule for subranges:

subrange ::= **(** typeName **|)** **(**	
([e_1 .. e_2] **|** [e_1 .. e_2) **) |**	**(typeName | INT).MKSUBRANGE ([e_1, (e_2 | e_2.PRED)]) |**
((e_1 .. e_2] **|** (e_1 .. e_2) **))**	[e_1.SUCC, **(** e_2 **|** e_2.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 ("′") characters. Such names cannot be written in a Cedar program, and hence they are safe from name conflicts. The desugaring is constructed so that the Cedar scope rules prevent multiple uses of these names from being confused.

3.1.3 Notation for the commentary

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

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

> *Anomaly*: the meaning of this Cedar construct is not explained by desugaring into the 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 like this.
>
> *Restriction*: a construct is not fully general, and will cause a static error unless the 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 in the examples, usually to point to a rule whose example introduces a name.

3.2 Lexical structure

```
56 name ::= letter (letter | digit)...                -- But not one of the reserved words in Table 3-2.
57 literal ::= num ?( ( D|d | B|b ) ?num ) |          -- INT literal, decimal if radix omitted or D, octal if B. |
   digit (digit |A|B|C|D|E|F) ... ( H|h ) ?num |      -- INT literal in hex; must start with digit. |
   ?num . num ?exponent |                             -- REAL as a scaled decimal fraction; note no trailing dot. |
   num exponent |                                     -- With an exponent, the decimal point may be omitted. |
   ' ( extendedChar | ' | " ) | • digit !.. (C|c) |   -- CHAR literal; the C form specifies the code in octal. |
   " ( extendedChar | ' ) ... " ?•(L|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|N | r|R) | (t|T) | (b|B) |                  -- CR, '\015 | TAB, '\011 | BACKSPACE, '\010 |
       (f|F) | (l|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*; these are the terminal symbols of the grammar. Rules 56-61 give the syntax of most tokens. A token is:

- A literal[57]. More information about literals of type *T* is in the section of § 4 devoted to *T*.
- A name[56], not one of the reserved words in Table 3 − 2. Note that case matters in names.
- A reserved word, which is a string of uppercase letters that appears 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 of one of the two-character sequences below. The legal punctuation symbols in programs are:

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

  The following ASCII characters are not legal punctuation symbols (and must not appear in a program except in an extendedChar[60]):

  % & \ ?

- 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]

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

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

Table 3 – 2: Reserved words and predefined names

The program is parsed into tokens by starting at the beginning and successively taking from the front the longest sequence of characters which forms a token according to the rules above, after first discarding any number of initial *whitespace* characters or *comments.*

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

A comment is one of:

A sequence of characters beginning with --, not containing -- or a carriage return, 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 token; they are token delimiters, and hence cannot appear in the middle of a token. Whitespace and comments thus do not affect the meaning of the program except:

When they delimit a token.

Within a CHAR literal or a ROPE literal, where they are taken literally. Thus ` is equal to '\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 made up entirely of upper case letters. All are at least three characters long except the following:

DO GO IF IN OF OR TO.

Caution on use of reserved words and predefined names: They should not be rebound by the program; in some but not all cases the compiler forbids their rebinding.

A note on lists of items and their separators. In general, semicolons are used to separate statements, or slightly larger constructs that contain statements. Commas are used to separate the items in all other kinds of lists. Precisely:

> *Semi-colons* are used to separate declarations, bindings and statements in a body[10], and to 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 range, a recordTC or a unionTC), bindings in an application[27] or an open[7], choices in a select expression[29, 32, 34] or in a unionTC[52], expressions in a choice[6, 9, 17, 30, 35, 52], items in imports, exports or shares lists[2, 3].

In general these lists may be empty, and an extra separator at the end is harmless when there 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 body[13] may be replaced by BEGIN and END reserved words. BEGIN replaces "{" and END replaces "}". If one brace is replaced, its matching partner must also be replaced. The braces delimiting an enumTC[54] may not be replaced by BEGIN and END.

3.3 Modules

```
1 module ::= DIRECTORY (n_d(: TYPE (n_t | ) | )          Λ [ (n_d: ( (TYPE n_t | TYPE n_d) | TYPE n_d) ), ... ] IN
                 ?(USING [ n_u, ... ] ) ), ... ;            LET (n_d~RESTRICT[n_d, [$n_u, ... ] ] ), ...
          ( interface | implementation )                 IN ( interface | implementation )
2 interface ::= n_m !.. : ?CEDAR DEFINITIONS            LET r'~[ n_m: INTERFACETYPE[[ $n_m, ...]] ] IN (imports | λ=>r') IN
  ?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 n_m

3 implementation ::= n_m : ?CEDAR                        LET r'~ [(n_e: n_e) , ... , FRAME: TYPE n_m ,
                                                                 n_m: FRAME, CONTROL: PROGRAM]
  ?safety ( PROGRAM ?drType[42] |                        IN (imports | λ=>r') IN
           MONITOR ?drType[42] ( | locks) )              ( | LET l'~( LET LOCK~NEWLOCK IN (λ IN LOCK) | locks) IN)
  (imports | )                                           LET b'~NEWPROGINSTANCE[block].UNCONS IN
  ?(EXPORTS n_e, ...)                                      [ (n_e~BINDDFROM[n_e, b' PLUS n_m~b'.n_m] ), ... ,
                                                             FRAME~MKINTTYPE[block],
  ?•(SHARES n_s, ...)                                       n_m~b' , CONTROL~b'.n_m] where the block body is desugared:
  ~ ?•access[12] block .                                   [ (d | b), ... , n_m: PROGRAM drType~{s; ...}]
3.1 imports ::= IMPORTS ( (n_v : | ) n_t ), ... --In 2, 3.   λ [(n_t:n_t), ...]=>r' IN LET [((n_v | n_t)~ (n_t PLUS n_t.BINDING) ), ... ]
4 safety ::= SAFE | UNSAFE --In 3, 41.
5 locks ::= LOCKS e ?( USING n_u: t)                     λ ?( [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,read],   --  unless most of the interface is used
   IO: TYPE IOStream,                             --  or it is a standard one like Rope or IO,
   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 not):

> A file of source 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 accessible to each other, hiding secrets or irrelevant information from other modules.
>
> A procedure which can accept interface types and bindings as arguments, and returns interface instances as results.
>
> The procedures of a monitor, perhaps with its protected data.

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

There are two kinds of modules: interface modules (written with DEFINITIONS) and implementation modules (written with PROGRAM or MONITOR). They have the same header (except that interfaces have no EXPORTS list); it defines the parameters and results of the module viewed as a proc (§ 3.3.1) and specifies the name n_m of the module. The bodies (following the ~) are different. Table 3−3 summarizes the structure of modules and their types; it omits a number of details which are given 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*, *Rope*: TYPE *Rope, IO*: TYPE *IO*, *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 they return.
>
> § 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 model (or on the compiler's command line), and used during compilation.

Instances of interfaces, declared in the IMPORTS list. These arguments are also supplied by the model (or in a *config* file passed to the binder, or implicitly by the loader), and used during loading.

§ 3.3.3 discusses the types of these arguments and how they are declared. In addition, an implementation may take PROGRAM arguments declared in the drType following PROGRAM or MONITOR. These are ordinary values; they are discussed in § 3.3.2A.

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 (§ 3.3.2A), called CONTROL.

This application cannot be written in the program, only in the model; it is described in § 3.3.2.

An *interface* (sometimes called an *interface type*) is a type, as the latter name suggests. This type is a declaration (obtained from the declarations which constitute the module body), with an extended cluster that includes all the bindings in the module body that don't use declared names (§ 3.3.4). In the example, the *Buffer* interface (obtained by applying the *Buffer* module to the arguments declared in its DIRECTORY) has declarations for *New*, *Get*, and *Put*, and its cluster includes values for *Handle* and *BufferObject*.

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

A *program instance* or a *global frame* is a frame, as the latter name suggests, i.e., a binding obtained 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 the instance directly, but only with the exported interface instances. In the example, *BufferImpl* exports a 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 instance.

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 is redundant to write the arguments explicitly.

When these conditions hold, there is a close correspondence among the following four objects:
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 simply consider the interface and implementation modules, and identify them with the files which hold their text. In 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 seven 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 *IO*), and *Rope* (of type *Rope*). Since the instance parameters are declared in an inner scope, the instance *Rope* is the one visible in the module body; the interface *Rope* is visible only in the header. The same is true for *IO*, but both the interface *CIFS* and the instance *Files* are visible in the body.

When *BufferImpl* is compiled, the four interface parameters must be supplied, in the form of (compiled) interface modules named *Rope*, *CIFS*, *IO* and *Buffer*. When *BufferImpl* is instantiated (normally by loading it), the three instance parameters must be supplied, i.e. there must be other instantiated implementation modules which export the *Rope*, *CIFS*, and *IO* interfaces. Normally 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 thus be used as a parameter by some other module.

3.3.2 Applying modules

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

A module is applied to its interface (DIRECTORY) arguments by compiling it; the result is a BCD (represented by a *.bcd* file). The *bcd* is still a proc, with instance parameters. Like any proc, a module can be applied to different arguments (i.e., different interfaces) to yield different results (BCDs).

A BCD is applied to its instance (IMPORT) arguments by loading (or binding) it; the result is a program instance, together with any interface instances exported by the module. Again, the BCD can be applied to different arguments (i.e., different interface instances) to yield different instances. Indeed, because an instance may include variables, even two applications to the same arguments will yield different results (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 compiler. The only type error possible in the second stage is supplying an unsuitable argument.

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

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

We've always done it that way.

3.3.2A Initializing a program instance

The statements in the body of an implementation module form the body of a proc called the *program procedure.* The function of this proc is to initialize an instance of the module. When program instance *PI* is made, no code in the module is executed; hence *PI* may be uninitialized. It is the job of the program proc *PP'* to initialize *PI*, perhaps using the PROGRAM arguments if there are any. Until *PP'* has been called, *PI* is not in a good state. It would be better to supply the PROGRAM arguments along with the imported instances, and call *PP'* as part of making *PI*, so that *PI* is never accessible in its uninitialized state. But it isn't done that way; hence the programmer must ensure that *PP'* is called before any use is made of *PI*. The preferred way to get hold of *PP'* is from an interface to which it is exported; see § 3.3.5.

To confuse things, *PP'* is not an ordinary procedure but a PROGRAM, and it must be called using the START construct (see § 4.4.1). Note that in addition to the statements of the module body, *PP'* also contains the type-specific initialization code for any variables or non-static values in the instance; e.g., if *x*: INT←3, the 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 called before the instance has been initialized by START, a *start trap* occurs. At this point, if *PP'* takes no arguments it is called automatically, and the original call then proceeds normally; if *PP'* does take arguments, there is a *Runtime.StartFault* ERROR.

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

Caution on referencing module variables before initialization: If a variable in the instance is referenced before the instance has been initialized, no error is detected, and the uninitialized value will be obtained. *PP'* can still be called to initialize the 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 if its 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 which 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 type TYPE *n*, where *n* is any one of the names given before DEFINITIONS in the header of the interface module that produced *I*. The INTERFACETYPE primitive in the desugaring takes a list of atoms and returns a type which implies TYPE *n* for each $*n* in the list. The reason for allowing several names is to aid conversion of an interface from one name to another; both names can continue in use for a while.

The use of these names provides a clumsy check that the proper interface is supplied as an argument. DIRECTORY *n*: TYPE and DIRECTORY *n* are both short for DIRECTORY *n*: TYPE *n*.

The compiler must be able to find the interface arguments, which in general are stored as files. When the modeller is used, it supplies these arguments from the specifications in the model. Otherwise, they may be specified explicitly on the compiler's command line, or failing that, the compiler gets the interface *I* from the file *I.bcd*.

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 of the bindings in the interface module body (§ 3.3.4).
>
> In an IMPORTS list as the type of an instance parameter to a module.
>
> After POINTER TO FRAME (§ 4.5.3)

The USING clause in the DIRECTORY, if present, restricts the cluster of the interface to contain only items with the names n_u, ... Thus in the example, only ROPE and *Compare* are in the cluster of *Rope* in the *BufferImpl* module. This means that *Rope*.ROPE and *Rope.Compare* are legal, but *Rope.n* for any other *n* will be an error. Note that USING affects only the cluster of the parameter; it does not affect the clusters of any types or the bodies of any INLINE procs obtained from the interface. Thus within *Rope*, *Compare* might be bound by

> *Compare:* PROC[*r*1, *r*2: ROPE] RETURNS [BOOL]~INLINE { IF *Length*[*r*1]~ = *Length*[*r*2] THEN ... }

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

In the example, *CIFS*, *IO*, and *Rope* are interfaces. They are the types of three IMPORTS parameters named *Files*, *IO*, and *Rope* (if the IMPORTS clause gives no name for the parameter, the name of the interface is recycled). An actual argument for an IMPORT parameter must be an interface instance, i.e., a value whose type is an interface type. Such a value is obtained from one or more modules which *export* the interface (§ 3.3.5). An instance is a binding; in it, the value of a name declared in the interface is provided by the exporter; the value of a name bound in the interface (e.g., *x*~3) is just the value the interface binds to the name (in this case, 3). This rule has two effects:

> The client can ignore the distinction between names bound and declared in the interface, since both appear in the instance binding and are referenced uniformly with dot notation. This means that the client is not affected, for example, when a proc is moved from an INLINE in the interface to an ordinary definition in an implementation.
>
> The client can often ignore the distinction between the interface and the instance, since all the values in the interface are also in the instance, with the same names. This is the motivation for the shorthand which allows the name of an IMPORT parameter to default to the name of the interface; the interface is no longer accessible, but *I.x* has the same meaning (namely 3) whether *I* is the interface or the instance.

Caution on inlines in interfaces: Names bound to inline procs in an interface do not appear in the interface binding, but only in an instance. This somewhat dubious rule ensures that clients won't have to add to their imports lists if a proc stops being an inline.

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

Restriction on importing a principal instance into imported interfaces: Often an interface module has no IMPORTS, because it only needs access to the static values (types and constants) bound in its interface parameters, and does not need values for any names declared there (procs and interface variables). If an interface module does have IMPORTS, however, and there is more than one instance of any imported interface around, then there is a restriction on the argument values. Suppose that *Int*1 imports *Int*2, and that a program module *P* imports *Int*1. Then *Int*1 may only import one instance of *Int*2, and if *P* also imports *Int*2, the principal instance of *Int*2 in *P* must be the same as the value of *Int*2 imported by the *Int*1 imported by *P*. For example, with

> DIRECTORY *Int*2: *Int*1: DEFINITIONS IMPORTS *Int*2*V*: *Int*2 ...
>
> DIRECTORY *Int*1, *Int*2: *P*: PROGRAM IMPORTS *Int*1*V*: *Int*1, *Int*2*V*: *Int*2 ...

we must have in *P* that *Int*1*V*.*Int*2*V* = *Int*2*V*.

3.3.4 *Interface module bodies*

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

Restriction on bindings in interfaces: The construct that follows the ~ in one of the bindings[13] is restricted:

> If it is an expression, it must be static (§ 3.9.1). Thus, no imported names. As a result,
> *P*: PROC~*I.P*
> *E*: ERROR~*I.E*
> are not allowed.
>
> If it is a block (providing the body of a proc), it must be INLINE (because there isn't any place to put the compiled code).
>
> It may not be CODE. This is an unfortunate accident of the implementation.

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

Restriction on referring to names introduced in an interface: The types and expressions in the declarations and bindings of an interface may refer to other names in the bindings as usual, but they *may not* refer to names introduced in the declarations, 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 type of a 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 which the interface is accessible. Thus if the value of the previous interface module is bound to *J*, e.g., because

J: TYPE *I* appeared in the DIRECTORY, then *J.x* is equal to 3. The declarations cannot be accessed directly (*J.y* is an error).

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

A transfer type (§ 4.4.1); this is just like a declaration of a transfer parameter to an ordinary proc, except that it is readonly.

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

VAR *T*, or READONLY *T* for any type *T* except TYPE; this is an *interface variable*; discussed in § 3.3.4.1 below. 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, there is an item *n*~*v* in the instance, and *v* has type *T*. This is the same rule which determines that a binding has the type of a declaration; e.g., that a proc argument has the domain type. In this respect there is nothing special about an interface.

Note that a name can be declared PRIVATE in an interface, even though it must be declared PUBLIC in the exporter (§ 3.3.6). This can be useful if the name is used in a type constructor or inline proc in the interface, but its value should not be accessible to the client.

3.3.4A Interface variables

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

•If you use the obsolete shorthand of *T* for VAR *T* in an interface variable declaration, you cannot declare a transfer type variable as an interface variable, since that already means passing the transfer value.

Caution on uninitialized interface variables: the variable which is exported to provide the value for an interface variable is not initialized until its module is initialized (§ 3.3.2A). However, there is nothing to stop it from being accessed sooner, with possibly undesired results.

Performance of interface variables: An interface variable can be read and (if not READONLY) set directly, which is significantly faster than *Get* and *Set* procs. Of course, the implementor gives up some control. These operations are not quite as fast as access to an ordinary variable, since there is an extra level of indirection which costs one or two extra instructions each time. There is also one pointer per interface variable per module which refers to it. If you use a private interface variable and inline *Get* and *Set* procs, you 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 type (§ 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. On the one hand, it is an ordinary block, the body of an almost ordinary proc *PP′* called the PROGRAM proc, which has parameters and results like any other. *PP′* is special in one way: it has a PROGRAM type rather than a PROC type. When *PP′* is applied (using the special construct START; see § 4.4.1), its declarations and bindings are evaluated, its statements are executed, and its results are returned

as with any proc. The only difference is that the values bound to the names introduced in the block (i.e., the frame of *PP′*) are retained after the proc returns; in fact, forever (unless *Runtime.Unnew* is used to free the frame). Procs local to the block can access these values in the usual way, and values of exported names can also be accessed through interfaces, as explained below; see § 3.3.2A.

As with any proc (§ 3.5.1), the frame of *PP′* includes the parameters and results from *Imp*'s drType[42] as well as the names introduced in the block's d's and b's. It also includes an additional item:

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 interfaces exported by *Imp*. For each interface *Ex* which *Imp* exports, an interface value *ExI* of type *Ex* is 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 a slightly peculiar kind of binding; as in an ordinary binding, *x* must be coerceable to *T* (§ 4.13). Note that *n* must have PUBLIC access (§ 3.3.6) in the body.

If *n* is *Imp* and *n*: *T* is in *Ex*, then *n*~*PP′* is in *ExI*; the type of *PP′* (which is PROGRAM *D* RETURNS *R*, where *D* RETURNS *R* is *Imp*'s drType) must be coerceable to *T*. This method of exporting *PP′* is the usual way of giving another module access to the program proc, so that it can be called to initialize the module at the proper time.

If *n* is declared in *Ex*, not bound in the body of *Imp*, and not the same as *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 *P*.

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 same way in any interface value.

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

On the other hand, it is quite usual to have several modules exporting to the same interface. The modeller, loader and binder provide facilities for merging the interface instances produced by the several modules into a single instance that contains all the items bound by any of the modules.

The result of instantiating *Imp* is a binding with:

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

One item CONTROL: PROGRAM[] RETURNS [], whose value is the program proc *PP′* if that has no arguments and no results, 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. The instance exists before *PP′* is called (though it is uninitialized). In fact, its *Imp* item can be applied to call *PP′*.

This binding is accessible in a model, where it can be used to get access to the interface instances, 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; *I* provides access to constants bound in the module, and allows you to declare an IMPORTS parameter whose argument will be a program instance of the module. From *I* you can also obtain a first-class Cedar type POINTER TO FRAME[*I*]; see § 4.3.5. *I*'s cluster includes a coercion from *I* to POINTER TO 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 it are now 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 use by clients, should be declared PRIVATE.

Access to a name is declared by writing PUBLIC or PRIVATE right after the colon in a declaration:
    x: PUBLIC *T*
In the Cedar syntax these colons occur in the declarations[11] and bindings[13] in bodies[10], fields[43,51], and interface modules[2], and in the tag[53] of a unionTC. You can set a default access for all the names in a module[2, 3] or record[50] by writing PUBLIC or PRIVATE just before the { or RECORD; this is overridden by an explicit PUBLIC or PRIVATE inside. By default, an interface is PUBLIC and an 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 if, e.g., *M* is 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.x*] regardless of whether *x* is PUBLIC or PRIVATE. It is illegal if *x* is PRIVATE, but it never refers to the *x* declared by the *x*: INT.

Furthermore, if a record has any PRIVATE components, a constructor or extractor for the record is legal only in a module where use of the PRIVATE names is legal (even if the private components are not mentioned and have defaults).

3.4 Blocks, OPEN and ENABLE

```
6 block ::= ?(CHECKED | UNCHECKED | TRUSTED)
  { ?open ?enable ?body                         open LET n'', ... : EXCEPTION~NEWLABEL[] , ...
    ?(EXITS (n, !..=>s); ...) }                  IN ( ( body enable ) BUT { (n'', ... => s ); ... } )
  --In 3, 13, 15.                               -- But n'' is not visible in s.
7 open ::= OPEN ( n ~~ e | e ), !.. ;           ( LET n~λ_open IN e.UNREF |  --The IN before !.. is a separator.
  In 2, 5, 17. •The ~~ may be written as :.      LET BINDP[(∇(e.UNREF)).P,
                                                  OPENPROCS[(∇(e.UNREF)).P, λ IN e.UNREF] ] ) IN !.. IN
8 enable::= ENABLE ( enChoice |                 BUT ( { enChoice } |
                   {enChoice; ...});                 { enChoice; ... } )
  In 5, 17.
9 enChoice ::= ( e, !.. | ANY ) => s            ( e | ANY ), ... => { s; REJECT; EXITS
  In 7, 27.1.                                      Retry' =>GOTO Retry''[14]; Cont' =>GOTO Cont''[14] }
10 body ::= (d | b); !.. ; s; ... | s; !..      LET NEWFRAME[ REC [(d | b), ...] ].UNCONS IN { s; ... }
  In 5, 17.
```

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["X"];   -- 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]]; CONTINUE; -- 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 allocation of variables declared in the block, as in Algol or Pascal. A Cedar block has four other features:

> 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 initialization expressions in the d's and b's are also evaluated in the order they are written; this may be important if they 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 ~) are known throughout the body with the values supplied by the d's and b's, except in inner scopes where they are reintroduced; they are not known elsewhere in the block. The *frame* of the block can be coerced to a binding with 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 desugaring for body indicates, the frame is constructed (by NEWFRAME), and then a LET makes the names in the binding known in the statements of the body.

Anomaly on order of evaluating bindings: A name introduced by a binding, *n*: *T*~*e*, has the value of *e* throughout the body if *e* is static. If *e* is not static, it is evaluated after all preceding d's and b's, but before any following ones. This means that *n* is trash in all the d's and b's before its binding. Symmetrically, if *e* refers to a name introduced in a following decl or non-static binding, it will get a trash value. Compiling with the "u" switch will yield a warning in this case. Note that only attempts to use the value of *n* get trash; *n* may appear anywhere in a λ-expression, and all will be well as long as the λ-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 *n* is allocated, and its INIT proc is executed, before any statements in the block is executed (this is done by the NEWFRAME proc in the desugaring).

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

Caution on uninitialized RC variables: The failure to initialize RC variables is a safety loophole, since the trash can be picked up and used as an address.

Style of expressions in bindings and initializations: The expression in a binding or defaultTC should be functional, or at least it should have only benign side-effects. There is no enforcement of this recommendation, unfortunately. In current Cedar such an expression is evaluated exactly once, at the time described above. This may change in the future, however.

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

Obtain a pointer to *v* with @; this pointer value can survive the block.

Obtain a proc value for a local proc which refers to *v*; this proc value can survive the block.

In the checked language both these *dangling references* are impossible: the @ operator, being unsafe, is forbidden, and ASSIGN for proc values gives an error unless the proc is local to a program instance (which has a retained frame).

Caution on dangling references to frames: An unchecked program can get into trouble.

Performance of block entry and exit: There is no overhead associated with block entry or exit, even if the block has an open, enable or EXITS. The only cost is for initializing the variables bound to its names. It is good 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 λ_{open} IN *e*.UNREF. This is just like λ IN *e*.UNREF, except that there is a coercion from *n* to *n*[]. In other words, every time *n* appears, its value is obtained by evaluating *e*.UNREF. The effect is exactly like call by name in Algol; the ~~ is to remind you that this is not ordinary value binding. The value of *e*.UNREF is

e if the cluster of ∇*e* does not include DEREFERENCE;

e↑.UNREF if it includes DEREFERENCE;

In other words, a reference value is dereferenced (and a single-component record or binding replaced by the component), repeatedly if necessary, to obtain a non-reference value. In an open, *e*.UNREF must be a record, interface or instance.

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

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

A record value has a corresponding binding which has the names of the record fields bound to the field values (or variables, for a VAR record).

•An application returns a binding, though the call-by-name feature makes it unwise to use an application in an open.

The nameless open converts *b* into another binding *bp* in which each value is a λ_{open} proc (see 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, with the same meaning as *r*.*a* and *r*.*b*.

Style for nameless open: Nameless open should be used with discretion, with the smallest practicable scope, and only if the value being opened is very familiar, or heavily used, or both. Nameless open can cause great confusion, since it is not obvious from the text of the program where to find the bindings for the 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 single open may have several bindings or expressions. These are applied sequentially, so that the names 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, § 2.6.2). ENABLE catches signals and errors raised in the body (but not the open, enable, or exits). EXITS catches GOTOs in the body or enable (but not the open or exits). Both are in the scope of the open, if any. Neither is in the scope of any names introduced in the body.

3.4.3A ENABLE

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

Each enable choice (enChoice[9]) has a list of expressions with exception values (•or ANY) before the =>. If ANY appears, it must be the last enChoice. If the exception is equal to one of these values, or if ANY appears, the statement after the => is executed. Control leaves this statement in one of the following ways:

- A REJECT statement causes the exception to be the value of the block; it will then be propagated within the enclosing block, or if the block is a proc body it will be propagated 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 is a proc 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 the block) over again at the beginning. You cannot write RETRY in a body's d's or b's.

  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 names in ∇e.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 with a matching *name*. Normally an error *E* will be declared in some interface, its value will be supplied by a binding of the form *E*: PUBLIC ERROR ... ~ CODE, and both the signaller and the enChoice will refer to this value by the name *E*. In this case, it is natural to think of the binding as being by name. However, it is possible to have a different name for this exception value, e.g. by writing *E1*: ERROR ... ~ *E*. It is also possible to bind some other exception value to *E* in a scope which includes 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 any application; this value immediately stops the evaluation of the containing application, which likewise returns *ev* as its value. This propagation is stopped only by an enable choice which catches the ERROR. As each application is stopped, it is *finalized.* Aside from invisible housekeeping, finalization confusingly consists of executing an enChoice which catches the ERROR UNWIND. The programmer can write any cleanup actions he likes in this statement.

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

Anomaly on order of finalization: In fact, things are a bit more complicated. When a signal or error is propagated, the enChoice statement is called as a proc from the SIGNAL or ERROR which raises the exception. When 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 enChoice statement is executed *before* any finalization. This is useful for signals, which often resume. In some cases, however, notably if finalization would release monitor locks, this can cause trouble. Avoid the problem by exiting from the enChoice immediately with a GOTO.

Caution on exceptions in enable choices: An enChoice can raise a second exception *ex*2 and fail to catch it. This will probably result in confusion, and should be avoided. If it happens, *ex*2 is propagated just like the first exception *ex*1; all the enChoices which saw *ex*1 will see *ex*2. This is because the enChoice statement for *ex*1 was called as a proc. Unless *ex*2 is a signal which is resumed, the enChoice which caught *ex*1 will be finalized and abandoned.

Caution on ANY and UNWIND: 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: don't.

Signals

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

> The proc to be called is an enChoice which is found exactly as though the signal were an error. The effect of this is that SIGNAL *P*[*args*] binds the proc name *P* to the proc body *dynamically*, by searching up the call stack for a binding of *P*. This is just the way Lisp binds free variables, except that a binding for *P* can only be found in an enChoice, not in the frame of a proc.
>
> Actually this is not quite right. Like an error handler, the signal proc is not found by matching *names*, but by 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. The GOTO exception is treated exactly like a GOTO out of an enChoice for an error; it causes all the intervening frames to be finalized.

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

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

3.4.3B EXITS

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

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

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

Anomaly on GOTO and UNWIND: 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 avoid 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 also hold afterwards. Roughly, these invariants ensure that the value of every expression has the syntactic type of the expression, and that addresses refer only to storage of the proper type (§ 4.5.1). An UNSAFE 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; this means that all the primitives it applies must be safe, as well as all the user-defined procs.

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

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 it preserves the safety invariants; the compiler assumes that the block is safe. This is a 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 transfer type constructor anywhere in the module defaults the SAFE option to TRUE. Hence the resulting 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 anywhere in the module defaults the SAFE option to FALSE. Hence the resulting type will be unsafe, and 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). The 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 with a safe type. This applies to enable choices for signals as well as to procs. In both cases, the body must be checked or trusted if the type is safe. ERRORs are treated differently, however, because of the view that an ERROR is a value returned from an application, unlike a signal which *calls* the enChoice expression. Hence the enChoice for an ERROR is treated just like any statement in its 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 (but not a REF), and all pointer arithmetic.

APPLY of

- a descriptor (because it involves dereferencing a pointer);
- a computed sequence, or 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

Syntax	Meaning
11 **declaration** ::= n, !.. : ?access[12] varTC[40]	(n: varTC), ...
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 |
t_2 -- if t=TYPE |	t_2 -- Same as e except for conflicting syntax. |
CODE |	NEWEXCEPTIONCODE[] --t⇒SIGNAL or ERROR |
?INLINE (ENTRY | INTERNAL |) block[6] |	λ [d′: t.DOMAIN] IN LET r′~NEWFRAME[t.RANGE].UNCONS IN (LET r′ IN ({t.DOMAIN~d′; (l′.ENTER; | |) block; RETURN} (FINALLY l′.EXIT | |)) 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~Histogram[11]; Size: INT~10;               -- Implementation:  Binds a TYPE and INT.
HistValue: PUBLIC TYPE~HV[40.1];                   --                  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: Handle[3], 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 contexts where 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. See also § 3.7 on argument bindings. Note that the **e** in a binding is evaluated just once, even if several names are bound. There are four special forms of binding given in rule 13, however, which are defined here:

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

A name with a signal or error type can be bound to CODE; this use of CODE is not allowed 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 are separated by semicolons. Each instruction is assembled from a list of expressions separated by commas. An expression in the list is evaluated to yield a [0..256) static value which forms one byte of the instruction; successive expressions form successive bytes.

A λ-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 subsection.

3.5.1 PROC bindings

A binding of the form *n*: *T*~{...} is the only way to construct a proc value and bind it to a name, since you cannot write a λ-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 type's 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 the exception type (§ 3.4.3A).

The expression following LOCKS in a module heading is turned into a proc according to a peculiar rule (§ 4.10).

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

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

Anomaly about parameter and result names: It is an error to introduce the same name twice in the domain, range or block.

Performance of proc calls: A proc call and return is about 30% faster if the proc is *local*, i.e., denoted by a name which was bound to a proc body in the same module as the call. A proc which is local to another proc, rather than bound in the body of an implementation, is about 20% slower to call. It also introduces some overhead when its parent proc is called, and its access to non-static names introduced in its parent proc is slower than access to other names. A call and return for an ordinary, non-local proc takes about 10 times as long as the statement $x \leftarrow y+z$, not counting the time for passing arguments or results. Each argument or result value costs as much as an assignment of that value. If the total size of the arugments is more than 11 words (in the current implementation), the cost of passing them is doubled, and likewise for results.

The attributes ENTRY and INTERNAL can be used only in a MONITOR; they are discussed in § 4.10.

The attribute INLINE has no effect on the meaning of the program, but it causes the proc body to be expanded inline whenever it is applied. This saves the cost of a proc call and return and sometimes the cost of argument passing, and it may allow static arguments to participate in static evaluation within the proc.

Restrictions on inlines: An INLINE proc may not be:

- Recursive.
- Exported.
- Used as a proc value except in an application; thus you cannot assign it to a proc variable.
- The argument of FORK.
- Accessed from the cluster of a POINTER TO FRAME type.

Caution on inlines in interfaces: An inline proc binding in an interface is not accessible from the interface (i.e., from a DIRECTORY argument); you must get it from an instance (i.e., import the interface). See § 3.3.4.

Performance of inlines: Excessive application of inline procs will result in much larger compiled code. Excessive definition of inline procs will result in much larger data structures in the compiler, and hence in larger symbol table files, and a greater chance of overflowing the compiler's capacity. The 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 accesses to the formals after potential side effects.

3.6 Statements

<table>
<tr><td>14 Statement ::= sS
In 6, 10, 17, 19.</td><td>{ SIMPLELOOP {sS; GOTO Cont″; EXITS Retry″=>NULL};
EXITS Cont″=>NULL }</td></tr>
<tr><td>15 sS ::= e_1←e_2 | e | block[6] | escape | loop | NULL</td><td>[e_1←e_2].TOVOID | e --must yield VOID-- | --all four yield VOID--</td></tr>
<tr><td>16 escape ::= GOTO n | GO TO n |
EXIT | CONTINUE | •LOOP | •RETRY |
(RETURN | RESUME) ?e |
•REJECT | †‡e ← STATE</td><td>HEX[exception[code~ n″, args~NIL]] |
GOTO (Exit′[17] | Cont′[9] | Loop′[17] | Retry′[9]) |
{ ?(r′[13]←e;) GOTO (Return′[13] | Resume′) } |
THISEXCEPTION[] | DUMPSTATE[e]</td></tr>
<tr><td>17 loop ::= (iterator |)
(WHILE e | UNTIL e |)
DO ?•open[7] ?•enable[8] ?body[10]

?(REPEAT (n, !..=>s); ...) ENDLOOP</td><td>{ (<u>iterator</u> ; | done′~FALSE; Next′: PROC~{};)
{ Test′~λ IN (NOT e | e | FALSE);
{ <u>open</u> SIMPLELOOP {
IF Test′[] OR done′ THEN GOTO FINISHED;
{ enable body EXITS Loop′=>NULL }; Next′[] }
EXITS Exit′⇒NULL; (n, !..⇒s); ...; FINISHED⇒NULL}}}</td></tr>
<tr><td>18 iterator ::= THROUGH e |
FOR (n : t | ★n)
((| DECREASING) IN e |

← e_1 , e_2)</td><td><u>FOR x′: e IN e</u> |
(n: t; |)
(Range′: TYPE~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); |
done′: BOOL~FALSE; Next′: PROC~{n←e_2}; n←e_1) ;</td></tr>
</table>

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..4] 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 easily defined in terms of a special type called VOID, which is equivalent to the empty declaration []. This is the range type of a PROC [...]→[], and it is also the result type of a block, control, loop or NULL statement. An expression whose value is a VOID can be used as a statement, and cannot be used as an ordinary value in a binding (since it wouldn't have the right type). If you want to call a proc 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 value is the value of the right operand. This is explained in the desugaring[15] using a special proc TOVOID in the cluster of every assignable type; it takes a value of the type and returns a VOID. Note that the grammar is ambiguous here, since there are two parsings of e_1←e_2 as a statement; the one written in the rule for statement is preferred.

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

Anomaly about applying a parameterless proc: •If you write an expression whose value is a proc taking no arguments as a statement, the 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 § 3.4.2 for open procs).

A statement[14] is actually a rather complicated construct, as the desugaring shows. This is because of 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 in various obscure cases. CONTINUE and RETRY are legal only in an enable choice (§ 3.4.2), and they may not appear in a declaration at all. •RETRY should be avoided everywhere, since it introduces a 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 § 3.4.3B. REJECT is explained in § 3.4.3A.

Anomaly about GOTO and procs: You cannot use a GOTO to escape from a proc body, even though the body is within the scope of the label. Only normal completion, or a RETURN or ERROR exception (or a SIGNAL which 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 the WHILE or UNTIL test. It has a body, bracketted by DO and ENDLOOP, which is almost like a block, but with some confusing differences:

> You catch GOTO exceptions with REPEAT, which is exactly like EXITS in a block immediately around the loop, except for the different delimiting reserved word. Note that the scope of the labels does not include the iterator or the test, even though these are evaluated repeatedly during execution of the loop. This feature is best avoided if possible, but unfortunately is 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 confusing. It is better to write a block explicitly inside the DO if you need these facilities.

There are three special exceptions associated with loops:

> EXIT is equivalent to GOTO *Exit'*, where *Exit'* is a label automatically declared in the REPEAT of every loop. Its enable choice does nothing. Thus EXIT simply terminates the smallest loop that encloses it.
>
> FINISHED is raised when the iterator or the WHILE/UNTIL test terminates the loop. It can be declared in the REPEAT like any label, but it must come last. If it is not declared, a null enable choice is supplied for it.
>
> •LOOP causes the next repetition of the loop to start immediately.

Anomaly about GOTO FINISHED: You cannot write GOTO FINISHED.

An iterator[18] declares a *control variable* *v* which is initialized by the iterator and updated after each execution of the loop; the scope of *v* is the entire loop, and it is constant in the loop. After the loop is terminated by the iterator (i.e., in the FINISHED clause), the value of *v* is undefined. •If you omit the declaration and simply name an already declared variable, it will be used as the control variable, and will not be constant; it will still be undefined after the loop is terminated by the iterator. Avoid this feature.

There are three flavors of iterator:

> THROUGH, which has no explicit control variable; THROUGH [0..*k*) or THROUGH [1..*k*] is convenient when you just want to loop *k* times.
>
> FOR *v*: *T* IN [*first* .. *last*] ...; *v* is initialized to *first*, and set to *v*.SUCC after each repetition. The iterator finishes the loop after a repetition which leaves $v \geq last$. The > case can only occur in FOR *v* IN ..., when an out-of-range value is assigned to *v* in the loop body. DECREASING reverses the order in which the elements of the subrange are used. The subrange need not be static. Note that the subrange is evaluated only once, before execution of the loop begins.
>
> FOR *v*: *T*←*first*, *next* ...; *v* is initialized to *first*, and set to *next* after each repetition. This iterator never finishes the loop. Note that the expression *next* is reevaluated each time 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* repetition, and that both tests are made before the first repetition, so that zero repetitions are possible.

3.7 Expressions

Syntax	Meaning
19 expression ::= n | literal[57] | (e) | application[26] |	
(e | typeName[37]) . (9) n |	
prefixOp e | e_1 infixOp e_2 |	e . **prefixOp** | e_1 . **infixOp**[e_2] |
e_1 relOp (4) e_2 |	(λ [x′: ∇e_1, y′ :∇e_2]=>[BOOL] IN **relop**) [e_1, e_2] |
e_1 AND (2) e_2 | e_1 OR (1) e_2 |	IF e_1 THEN e_2 ELSE FALSE | IF e_1 THEN TRUE ELSE e_2 |
e ↑ (9) | •STOP | ERROR |	e . DEREFERENCE | STOP[] | ERROR NAMELESSERROR |
builtIn [e_1 ?(, e_2, !..) ?applEn[27]] |	e_1 . **builtIn** ?([e_2, ... ?applEn]) |
funnyAppl e ?([?argBinding[27] ?applEn[27]]) |	e . **funnyAppl** ?([?argBinding ?applEn]) |
[argBinding[27]] |	--Binding must coerce to a record, array, or •local string-- |
subrange[25] |	
if[28] | select[29] | safeSelect[32] | •withSelect[34] |	
s	

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 select choices[30 33 35].

Syntax	Meaning
20 **prefixOp** ::= @ (8) | − (7) | (~ | NOT) (3)	VARTOPOINTER | UMINUS | NOT
21 **infixOp** ::= * | / | MOD (6) | + | − (5) | ← (0)	TIMES | DIVIDE | REM | PLUS | MINUS | ASSIGN
22 **relOp** ::= ?NOT (?~ (= | < | >) | # | (<= | >=) | IN)	?NOT (?NOT x′.(EQUAL | LESS | GREATER)[y′] | x′~=y′ | x′=y′ OR x′ (< | >) y′ | x′>=y′.FIRST AND (x′<=y′.LAST BUT {BoundsFault=>FALSE}))
--In 19, 30.	
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] |) ([| ( ) e_1 .. e_2 ( ] |))	LET t′~(**typeName** | INT) , first′~(e_1 | e_1.SUCC) IN t′.MKSUBRANGE[first′, (e_2 | e_2.PRED)] BUT {BoundsFault=>t′.MKEMPTYSUBRANGE[e_1]}
--In 19, 39, 48.	
26 **application** ::= e [?argBinding ?applEn]	LET m′~e, a′~[**argBinding**] IN ((m′. APPLY ▸a′) ?applEn)
27 **argBinding** ::= (n ~ (e | | ★TRASH)), !.. | (e | | ★TRASH), ...	(n ~ (e | OMITTED | TRASH)), !.. | (e | OMITTED | TRASH), ...
In 19, 26. •TRASH may be written as NULL, ~ as :.	
27.1 **applEn** ::= ! enChoice[9]; ...-- In 19, 26.	BUT { **enChoice**; ... }

Examples

Example	Comment
lv: LabelValue[13]←[i, 3, "Hello", 31.4E—1, (i+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[LabelValue[13]];	-- must be statements.
h[−3, NOT(i>j), i*j, i←3, i NOT >j, p OR q, lb.r↑];	-- An application with sample expressions.
lv[19]←[first~0,last~5,x~3.2,g~2,f~5,r~NIL,s~"1"];	-- Short for *lv← LabelValue*[13][...].
[first~i, last~j]←lv[19];	-- Assignment to VAR binding -- (extractor).
b: BOOL←i IN [1..10]; FOR x: INT IN (0..11) DO ...;	-- Subrange only in types or with IN.
b←(c IN Color[54](red..green] OR x IN INT[0..10));	-- The INT 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];                           -- ≡File.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 and postfix operators, explicit application of a primitive proc[23], or the funnyAppl[24] in which the first argument follows the proc name without any brackets. All of these constructs desugar into dot notation (§ 2.4.4, § 4.14); this means that the procs come from the cluster of the first argument. The exceptions to this rule are ALL, CONS for variant records and lists, LIST, and the single-argument 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 if expression, since the second operand is evaluated only if the first one is TRUE or FALSE respectively.

The order of evaluation for arguments of an application, and therefore for operands in an expression, is not defined (unless the operator is AND or OR). However, the arguments are evaluated one at a time, and all arguments are evaluated before the proc is applied. In particular, an 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 such as [*a*~*x*, *b*~*y.f*].

Rules 19-21 give the precedence for operators: ↑ and . are highest (bind most tightly) and ← is lowest. All are left-associative except ←, which is right-associative. Application has still higher precedence.

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

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

Pair: TYPE~RECORD[INT, INT]

you can write

[*i*, *j*]←*Pair*[*j*, *i*]

to transpose *i* and *j*.

The expresssion ERROR is short for raising a nameless ERROR exception. You should think of it as a 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 brackets in the usual way; e.g., START *P*[3, "Help"]. RETURN WITH ERROR is explained in § 4.10.

Anomaly about NEW: The funnyAppl NEW *e* actually stands for *e*.COPYIMPLINST. See § 4.4.1 and § 4.5.3.

Anomaly about enables in funnyAppls: 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 open and closed intervals is used to indicate whether the endpoints are included in the subrange. A subrange can also be used after IN in an expression or iterator; in these contexts it need not be static.

You can write enable choices[9] after a ! inside the brackets of an application[26], built-in[23], or funnyAppl[24]. See § 3.4.3A for the semantics of this. Note that only an exception returned by the application is caught by these choices, not one resulting from evaluating the proc or arguments.

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

3.8 IF and SELECT

```
28 if ::= IF e1 THEN e2 (ELSE e3 | )                IF e1 THEN e2 ELSE (e3 | NULL)
29 select ::= SELECT e FROM                         LET selector'~e IN
      choice; ... endChoice                             choice ELSE ... endChoice
   The ":" is "." in an expression; also in 32 and 34.   -- ELSE is a separator for repetitions of the choice.
30 choice ::= (( | relOp22 ) e1 ), !.. =>e2         IF ((selector' (= | relOp ) e1) OR ... ) THEN e2
31 endChoice ::= ENDCASE ( => e3 | )                ELSE (e3 | NULL)
   In 29, 32, 34.

32 safeSelect ::= WITH e SELECT FROM                LET v'~e IN
      safeChoice; ... endChoice31                       safeChoice ELSE ... endChoice
33 safeChoice ::= n : t => e2                       IF ISTYPE[v', t] THEN LET n : t←NARROW[v', t] IN e2
34 •withSelect ::= WITH (n1 ~~ e1 | • e1 )          OPEN v'~~e1 IN LET n'~($n1 | NIL), type'~∇v',
   SELECT ( | †e11) FROM                              selector'~(e1.TAG | e11) IN withChoice ELSE ... endChoice
      withChoice; ... endChoice31                   -- e11 must be defaulted except for a COMPUTED variant.
   •The ~~ may be written as :.
35 •withChoice ::= n2 => e2 |                       IF selector'=$n2 THEN OPEN
   n2, n2, !.. => e2                                  (BINDP[n', LOOPHOLE[v'.type'.n2] ] | BINDP[n', v'] ) IN e2
```

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=>{...};                                      -- ≡r:INT~f[j]; IF r<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 evaluates e_2 if *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 chosen based on a single value. The selector expression e is evaluated once to yield a value *selector'*, and then each of the choices is tested in turn. Within each choice, each expression e_1 preceding the => is compared in turn with *selector'*; the comparison is *selector' relop* e_1 if e_1 is preceded by a *relop*; otherwise it is *selector'* $= e_1$. If any comparison succeeds, the expression e_2 following the => is evaluated to yield the value of the select. If no comparison succeeds, the next choice is tried. If no choice succeeds, the expression e_3 following the ENDCASE is evaluated to yield the value of the select; e_3 defaults to NULL, and hence must be present when the select is not a statement to prevent a type error.

Style for SELECT: It is good practice to arrange the tests so that they are disjoint and exhaust the possible values of the selector. ENDCASE should be used to mean "in all other cases"; often the appropriate e_2 raises an error. Don't use ENDCASE to mean another specific selector value which you don't bother to mention. Another acceptable form is SELECT TRUE FROM ...., which selects the first choice that succeeds, and is sometimes easier to read than a long sequence of ELSE IF's.

Performance of SELECT: If the e_2 are static and select subsets of the selector values, the average size of these subsets is not too large, and the density of unselected values is not too high, a select compiles into 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 must be a 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 one possible type that the selector might have, and declares a name which is initialized to the selector value if it has that type. Thus, the example tests for *r* having the types REF INT and REF REAL. If it has REF INT, the first choice's *e* is evaluated; within *e*, *rInt* is a variable initialized to the selector, and has type REF INT. Likewise for REF REAL and the second choice. As with an ordinary select, the ENDCASE expression is evaluated (with no new names known) if none of the other choices succeeds. Note that safeSelect does ordinary binding by value, not the binding by name done in open and withSelect.

•†A withSelect[34] is an unsafe and rather tricky construction for discriminating cases of unions. Its use should be avoided unless a safeSelect can't do the job; this is the case for a COMPUTED tag, or 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 is dereferenced to yield a variant record value. It also means that this value is *not* copied, and hence it can change its type during execution of a choice, either by assignment to the variant part of a variant record (an unsafe operation), or by a change in the value of e_1.
>
> If the union has a COMPUTED tag, the selector value to be used for the discrimination must be given as e_{11} in the withSelect. It is entirely up to the programmer to supply a meaningful value. If the tag is not COMPUTED, e_{11} must be omitted and the selector value is e_1.TAG.
>
> The n_2 preceding => in a choice are literals of the (enumerated) type (§ 4.7.1A) which is the tag type of the union (§ 4.6.3). They are compared with the selector, and if one matches, the e_2 following => is evaluated as with an ordinary select. If exactly one is given, then the e_2 following => is in the scope of
>
> OPEN n_1~~LOOPHOLE[e_1.UNREF, $V.n_2$], or simply OPEN LOOPHOLE[e_1.UNREF, $V.n_2$]
>
> if no n_1~~ followed the WITH. If several n_2 are given, then there is no discrimination, and the e_2 following => is in the scope of
>
> OPEN n_1~~e_1.UNREF or OPEN e_1.UNREF

3.9 Miscellaneous

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

3.9.1 Static values

An expression has a *static* value if the compiler can compute the value. Static values are required in various contexts, notable in type expressions, and as the right hand side of a binding in an interface module. In Cedar, an expression has 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.

Note that values obtained from an interface are static, but imported values are not.

Performance of static expressions: The compiler evaluates *all* static expressions, not just type expressions. This is often 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$\geq 2^{16}$ lacks the following procs:
  ALL
  ASSIGN
  CONS
  DESCRIPTOR
  INIT
  NEW
- A subrange type T must have
  $0 \leq T$.LAST $-$ T.FIRST$<2^{16}$
  $-2^{15} \leq T$.FIRST$\leq 2^{15}$
  T.LAST$<$(IF T.FIRST<0 THEN $2^{15}+T$.FIRST ELSE 2^{16})

3.9.3 Checking

Possible errors arising from certain primitive operations are checked, and cause ERROR exceptions if 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).

In an UNCHECKED block these errors are *not* checked for unless the program is compiled with the "u" switch.

Chapter 4. Primitives

This chapter gives detailed information about the primitive types, type-returning procs (*type constructors*), and other procs. It should be read after § 2.4, which defines a Cedar type and explains the basic ideas underlying the type system.

§ 4.1 gives the partial ordering called the *class hierarchy* that is used to classify the primitive types. § 4.2 lists all the primitives of Cedar. §§ 4.3-4.11 give the declarations and semantics for all the primitive classes and types. These descriptions are ordered according to the class hierarchy in Table 4−1. Each one specifies:

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

The constructor for types in the class.

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

Anomalies and facts about performance.

The implies relations on primitive types are summarized in § 4.12, and the coercions in § 4.13. The various cases of dot notation are described in § 4.14.

4.1 The class hierarchy

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

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

By convention, the name *T* in a cluster denotes the type to which the cluster belongs. We call each <name, type> pair in the class an *item*.

Sometimes when a type *U* is derived from another type *T* (e.g., REF *T* from *T*), some of *U*'s items are obtained from *T*'s items with the same names in some simple way (e.g., REF RECORD[*a*, *b*: INT] has procs *a* and *b* which dereference the REF and then apply the record's *a* and *b* procs). We say that *U inherits* the items from *T*.

A type *T* is *in* a class *C* if *T.Cluster* has the type *C*; we also say that *T is* a *C* type, e.g., INT is in class *Numeric*, or is a numeric type.

To make this explicit, we give the type CLASS a cluster proc called *Type*, such that every type *T* in class *C* has type *C.Type*. For example, INT has type *Numeric.Type*. Thus,

T is a *C* type ≡ *T* in *C* ≡ *T* has type *C.Type* ≡ (*C.Type*).*Predicate*[*T*] = TRUE

A value satisfies the predicate for *C.Type* if it is a type, and its cluster satisfies the declaration which defines *C*. E.g., INT satisfies the predicate for *Numeric.Type* because it is a type, and its cluster contains procs for PLUS, MINUS, LESS etc. with the right types. Precisely, (*C.Type*).*Predicate* is

λ [*T*: ANY] IN TYPE.*Predicate*[*T*] ∧ *C*.*Predicate*[*T.cluster*]

Class	Subclasses or types
all	general* | TYPE◇ § 4.8 | fully opaque § 4.3.4 | TYPE *n* § 4.3.5 | interface § 4.3.5 | SEQUENCE→row
general § 4.3.1	assignable* | hasNIL* | variable § 4.3.3 | PORT→transfer | MONITORLOCK◇ § 4.10 | CONDITION◇ § 4.10 | composite with a non-assignable component, and not a SEQUENCE
assignable § 4.3.2	-- everything not mentioned separately under all or general, i.e.:-- *n*-opaque § 4.3.4 | transfer→map | descriptor→map | address | RELATIVE | ordered | unspecified | composite with no non-assignable components.
has NIL § 4.3.7	variable | address | transfer
composite § 4.3.3	row | RECORD | union
map § 4.4	transfer* | row* | descriptor*/address | BASE POINTER/pointer § 4.4.3
transfer § 4.4.1	PROC | PROGRAM | PORT | PROCESS | SIGNAL | ERROR
row § 4.4.2/composite	ARRAY § 4.4.2A | SEQUENCE--second class-- § 4.4.2B⊃(TEXT◇ | *StringBody*◇)
descriptor § 4.4.2.3 /address	LONG DESCRIPTOR | DESCRIPTOR
address § 4.5	reference* | descriptor→map | ZONE § 4.5.2 | POINTER TO FRAME § 4.5.3
reference § 4.5.1	REF⊃(LIST | ATOM◇) § 4.5.1A | pointer*
pointer § 4.5.1B /ordered	LONG POINTER⊃LONG STRING◇ | POINTER⊃STRING◇ | BASE POINTER→map
RELATIVE § 4.5.4	RELATIVE POINTER | RELATIVE DESCRIPTOR
record --painted-- § 4.6 /composite	RECORD § 4.6.1 | variant § 4.6.2
union --second class, painted-- § 4.6.3 /composite	
ordered § 4.7	discrete* | numeric* | pointer→address | subrange § 4.7.3
discrete § 4.7.1	whole number→numeric | enumeration --painted-- § 4.7.1A⊃(BOOL≡BOOLEAN◇ | CHAR≡CHARACTER◇)
numeric § 4.7.2	whole number*/discrete | REAL◇ § 4.7.2B
whole number § 4.7.2A	long number* | short number*
long number	INT≡LONG INTEGER◇ | •LONG CARDINAL◇
short number	INTEGER◇⊃NAT◇ | CARDINAL◇⊃NAT◇
•unspecified § 4.9.1	•UNSPECIFIED◇ | •LONG UNSPECIFIED◇
exception | DECL | BINDING § 4.9.2 --kernel only--	
process § 4.10	MONITORLOCK◇ | CONDITION◇

Notation:

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

Table 4 – 1: The class hierarchy

A class C is a *subclass* of another class D if $C \Rightarrow D$. Recall the implies relation for declarations (§ 2.2.1F) means that

Each name n in C is also in D.

n's type in C implies n's type in D.

Precisely,

$(\forall n \in C.names)\ n \in D.names \wedge (C.\mathrm{DTOB}.n \Rightarrow D.\mathrm{DTOB}.n)$

For example, the class ORDERED includes

LESS: PROC[*T*, *T*]→BOOL

Every subclass of ORDERED must also declare a LESS proc which takes two Ts to a BOOL. If we had a richer assertion language, there would also be axioms defining LESS to be an ordering relation. Similarly, every ORDERED type (e.g., INT) must have such a LESS proc in its cluster.

The subclass relation defines a class hierarchy, i.e., it gives a partial ordering on classes. Table 4–1 gives the class hierarchy for the primitive classes of Cedar. It is presented as a tree: a node N with sons $N_1, N_2, ..., N_k$ is written

N $\qquad\qquad\qquad$ $N_1 \mid N_2, \mid ... \mid N_k$

and any of the N_i that are not leaves are marked with a * and defined on following indented lines:

N_i $\qquad\qquad\qquad$ $N_{i1} \mid N_{i2}, \mid ...$

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

4.2 Summary of primitives

36 **type** ::= typeName | builtInType | typeCons

37 **typeName** ::= n_1 | typeName . n_2 |
typeName [e] | •n_2 typeName $\qquad$ **typeName**.SPECIALIZE[e] | **typeName** . $\mathbf{n_2}$
In 19, 25, 36, 40.1, 49. $\qquad$ --n_2 names a variant.

38 **builtInType** ::= INT | REAL | TYPE | ATOM | MONITORLOCK | CONDITION |
★ ?†UNCOUNTED ZONE | •†MDSZone | •LONG CARDINAL | •† ?LONG UNSPECIFIED -- See Table 4–2.
TYPE only as t in a b or an interface's d. INTEGER, CARDINAL, NAT, TEXT, STRING, BOOL, CHAR are predefined.

39 **typeCons** ::= subrange[25] | paintedTC[40.1] | transferTC[41] | arrayTC[44] | seqTC[45] | †descriptorTC[45.1] |
refTC[46] | listTC[47] | †pointerTC[48] | •†relativeTC[49] | recordTC[50] | unionTC[52] | enumTC[54] | defaultTC[55]

Examples

```
P: PROC[ b: Buffer¹.Handle,          -- A type from an interface.
         i: INT←TEXT[20].SIZE ];     -- A bound sequence; only in SIZE, NEW.

TypeIndex: TYPE~[0..256);            -- A subrange type.
BinaryNode: TYPE~Node⁵².binary;      -- A bound variant type.
```

The tables in this section summarize the primitive and predeclared types, type constructors and procs of Cedar. There are also a number of interfaces which contain useful procs or values of primitive types; in some cases, the distinction between a primitive in the language and one in such an interface is rather arbitrary. These interfaces are *Process, Inline, CedarReals, AMTypes, Rope, SafeStorage, UnsafeStorage, ListsAndAtoms, PrincOps, Runtime.*

4.2.1 Primitive types and constructors

Table 4–2 lists the primitive or predeclared types of Cedar, giving the name for each in the current language, and either a definition or, for the primitive types, a comment suggesting the meaning of the type. Later sections describe the items in the clusters of these types, and give their representations.

Name	Meaning
INT, LONG INTEGER § 4.7.2.1	$=[-2^{31}..2^{31})$
REAL § 4.7.2.2	-- 32-bit IEEE floating point
BOOL, BOOLEAN § 4.7.1.1	={FALSE, TRUE}
CHAR, CHARACTER § 4.7.1.1	={'\000, ..., '\377}
TYPE § 4.8	
ATOM § 4.5.1.1	-- for unique strings, global property lists
CONDITION § 4.10	-- for process synchronization
-- *The following are appropriate when performance tuning is needed.*	
★INTEGER § 4.7.2.1	$=[-2^{15}..2^{15})$; INTEGER.SIZE=1
★NAT § 4.7.2.1	$=$ INTEGER$[0..2^{15})$; NAT.SIZE=1
★TEXT § 4.4.2.2	=MACHINE DEPENDENT RECORD [*length* (0): [0..INTEGER.LAST] ← 0, *text* (1): PACKED SEQUENCE *maxLength* (1): [0..INTEGER.LAST] OF CHAR]
★ZONE § 4.5.2	-- controls safe storage allocation
-- *The following are not recommended for general use.*	
★MONITORLOCK § 4.10	-- use MONITOR or MONITORED RECORD
†UNCOUNTED ZONE § 4.5.2	-- controls unsafe storage allocation
LONG CARDINAL § 4.7.2.1	$=[0..2^{32})$, mixes poorly with INT.
CARDINAL § 4.7.2.1	$=[0..2^{16})$; CARDINAL.SIZE=1
-- *The following are obsolescent.*	
●†*MDSZone*	-- controls unsafe storage allocation in the MDS.
●†?LONG STRING § 4.4.2.2	=?LONG POINTER TO *StringBody*
●*StringBody* § 4.4.2.2	=MACHINE DEPENDENT RECORD [*--see text for anomalies--* *length* (0): CARDINAL ← 0, *maxLength* (1): --READONLY-- CARDINAL, *text* (2): PACKED ARRAY [0..0) OF CHAR]
●†UNSPECIFIED § 4.9	-- unsafe, matches any one-word type
●†LONG UNSPECIFIED § 4.9	-- unsafe, matches LONG INTEGER, LONG CARDINAL, REAL, LONG POINTER, or REF.

Table 4 – 2: Primitive and predeclared types

4.2.2 Type constructors

Table 4–3 gives the declarations of all the primitive Cedar type constructors. Since type-returning procs cannot be written in the current language, these are in fact all the Cedar type constructors. The concrete syntax for type constructors is in rules 40-55, and in § 4.2.2.1 on options. Rule 39 above lists all the cases.

All the arguments of a type constructor must be static (§ 3.9.1), except for:

> MKSUBRANGE, which can have non-static arguments when it appears in an expression or iterator as the second operand of IN.
>
> CHANGEDEFAULT, which takes a proc derived from the e in the defaultTC. This e may be non-static in an implementation, or in the fields of a transferTC in an interface.
>
> *T*[*n*], where *T* is a sequence-containing record.

All the type constructors are functional (produce the same type when given the same arguments) except TYPE[ANY], TYPE[*n*], MKUNION, and MKRECORD, MKENUMERATION and their MD friends in an

interface. MKRECORD and MKENUMERATION are functional in an implementation so that module replacement is more convenient. A non-functional type constructor produces a *different* type each time it is applied. By a slight misuse of language, such types are sometimes called *painted.*

In current Cedar, type expressions and ordinary expressions do not have the same syntax. The severe restrictions on where types can be used ensure that the parser can distinguish the cases where a type is expected. There are a few cases where this is not true, and type names (rule 37) must be written instead of general expressions: subrangeTC, specializations of variant records, relativeTC and paintedTC.

Name	*Domain*	*Class of result*	*Rule*	§
MKVAR	[*readOnly, short*: BOOL←FALSE]	variable	40	§ 4.3.3
-- This proc in the cluster of each type *T* produces the type VAR *T* or READONLY *T*.				
REPLACEPAINT	[*in*: TYPE, *from*: OPAQUE.*Type*]	general	40.1	§ 4.3.4
MKINTTYPE	[LIST OF DECLORBINDING]	interface	2	§ 4.3.5
INTERFACETYPE	[LIST OF ATOM]	TYPE *n*	2	§ 4.3.5
MKXFERTYPE	[*flavor*:{PROC,PORT,PROCESS,SIGNAL,ERROR,PROGRAM}, *domain, range*: DECL←NIL, *safe*: BOOL←ISCEDAR]	transfer	41	§ 4.4.1
MKPROC	[*domain, range*: DECL←NIL, *safe*: BOOL←ISCEDAR]	PROC	41	§ 4.4.1
~MKXFERTYPE[PROC, *domain, range, safe*]				
MKARRAY	[*domain*: DISCRETE.*Type*←CARDINAL, *range*: TYPE, *packed*: BOOL←FALSE]	ARRAY	44	§ 4.4.2.1
MKSEQUENCE	[*domain*: TAG, *range*: TYPE, *packed*: BOOL←FALSE]	SEQUENCE	45	§ 4.4.2.2
•MKARRAYDESCR	[*arrayType*: ARRAY.*Type*, *long*: BOOL←FALSE, *readOnly*: BOOL←FALSE]	DESCRIPTOR	45	§ 4.4.3
MKREF	[*target*: TYPE, *base*: BASE←WORLD, *readOnly, ordered, uncounted*: BOOL←FALSE]	reference	46	§ 4.5.1
MKLIST	[*range*: TYPE, *readOnly*: BOOL←FALSE]	LIST	47	§ 4.5.1.1
†MKPOINTER	[*target*: TYPE←UNSPECIFIED, *long, readOnly, ordered, base*: BOOL←FALSE]	pointer	48	§ 4.5.1.2
~MKREF[*target*~*target*, *readOnly*~*readOnly*, *ordered*~*ordered*, *uncounted*~TRUE, *base*~(IF *long* THEN WORLD ELSE MDS)].				
★†MKRELATIVE	[*range*: TYPE, *baseType*: BASE.*Type*]	RELATIVE	49	§ 4.5.4
MKRECORD or MKMDRECORD	[*fields*: DECL, *access*: {PUBLIC, PRIVATE}←CURRENTACCESS, *monitored*: BOOL←FALSE]	RECORD	50	§ 4.6.1
MKPOSITION	[*firstWord*: NAT, *firstBit*: INT←0, *lastBit*: INT←−1]	–	51	§ 4.6.1
MKUNION	[*selector*: TAG, *variants*: LIST OF FIELD]	union	52	§ 4.6.3
MKENUMERATION	[LIST OF ATOM]	enumeration	54	§ 4.7.1.1
MKMDENUMERATION	[LIST OF RECORD[ATOM, NAT]]	enumeration	54	§ 4.7.1.1
MKSUBRANGE	[FIRST: *T*, LAST: *T*]	subrange	25	§ 4.7.3
-- *T* is the *Discrete* base type, which has a MKSUBRANGE type constructor in its cluster.				
CHANGEDEFAULT	[*type*: TYPE, *proc*: (PROC[]→*type*), *allowTrash*: BOOL]	assignable	55	§ 4.11

Table 4 – 3: Primitive type constructors

4.2.2A Options

The built-in type constructors take an assortment of optional BOOL arguments, as indicated in their declarations. In the current syntax these are specified by writing *options* in the type constructor. When an option appears in a type constructor, the argument of the same name has the value TRUE; if it is missing, the argument has the value FALSE (except for SAFE, which defaults to TRUE if the module header says CEDAR, to FALSE otherwise). The effect of these arguments on the type produced by the constructor is given as part of the description of its result class. Table 4–4 lists the options and the constructors for which each is appropriate.

Option	*Constructors*
★BASE	MKPOINTER
LONG	MKPOINTER, MKARRAYDESCR
MONITORED	MKRECORD
•ORDERED	MKPOINTER
★PACKED	MKARRAY, MKSEQUENCE
PUBLIC, PRIVATE	MKDECL, MKRECORD
READONLY	MKVAR, MKREF, MKLIST, MKPOINTER, MKARRAYDESCR, MKDECL (interface vars only)
SAFE	MKXFERTYPE
UNSAFE	MKXFERTYPE

Table 4–4: Type options and their constructors

4.2.3 Primitive procs

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

The *Name* column gives the name of the value in the cluster. For a proc, the following symbol summarizes the handling of exceptions:

> A "!" means that application can cause an exception, and you can write an applEnable[27.1].
>
> An italic "*!*" means that an exception is possible, but you cannot write an applEnable. If you are desperate, enclose the application in a block with an enable.
>
> An italic "*!!*" means that an exception should be possible, but the implementation does not make the necessary check (e.g., for overflow on adding INTs).
>
> If nothing follows the name, no exception is possible.

The *Classes* column gives the classes in which the name appears; see Table 4–1.

The *Type* column gives the type with which it is declared in those classes. The type usually refers to other names of the class. Since it is taken from the class declaration, it can use these names without explicit qualification; see the detailed class descriptions in § 4.3-4.11 for their meanings.

The *Notes* column gives information about how a proc is applied or a non-proc value is denoted in current Cedar. In the kernel a proc named *P* from the cluster of type *T* is applied to a value *x* of type *T* by the expression *x.P* if there is only one argument or *x.P*[*y*, ...] if there are several. In current Cedar, however, not all primitives can be applied or denoted by dot notation. There are three other ways of applying a primitive proc:

> It may be an *operator* with a symbol listed in the *Notes* column. If it takes two arguments, the operator is infix. Thus for a proc named *P* with operator symbol ⊕, you write *x*⊕*y* instead of *x.P*[*y*]. If it takes one argument the operator is usually prefix; you write ⊕*x* instead of *x.P*. The ↑ operator is postfix; you write *x*↑ instead of *x*.DEREFERENCE.

It may be a *built-in* proc named *P*, in which case you usually write *P*[*x*] or *P*[*x*, *y*, ...] as an alternative to *x*.*P* or *x*.*P*[*y*, ...]. For each built-in which cannot be applied using either of these notations, the ways of applying it are indicated explicitly in the *Notes* column; any ways not mentioned *cannot* be used.

It may be a *funny application* proc named *P*, in which case you write *P x* or *P x*[*y*, ...].

The three kinds of primitive proc are listed in that order, alphabetically within each kind. Values which are not procs (ABORTED, FALSE, FIRST, LAST, NIL, SIZE, TRUE) are listed with the built-in procs. Except for ABORTED, FALSE and TRUE, which are globally known and must be written alone, the cluster must be specified by dot notation (INT.SIZE) or optionally as an argument (SIZE[INT]).

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

Some PROC [*T*]→[*U*] are coercions: CONS, FROMGROUND, LONG, TOGROUND, VALUEOF. This means that they may be invoked automatically when typechecking demands a *U* and an expression has syntactic type *T*; see § 4.13 for details.

Some involve *target typing*: ALL, CONS, LIST, VAL, union constructor; they are marked TT. For these the proc does not come from the cluster of the type of the first argument. Instead, it comes from the cluster of the so-called *target type*. An application of one of these procs must appear as an argument in another application (e.g., *f*[*y*, NARROW[*x*]] or *z*←NARROW[*x*]), and not before a dot. In this context the target type is known from the declaration of the outer proc being applied (*f* or *z*.ASSIGN in the example; if the type of *f* is PROC [*U*, *T*]→[*V*], the target type for the NARROW application is *T*). Target typing is also used for enumeration literals (§ 4.7.1A), and is optional to default the type argument of DESCRIPTOR, NARROW or LOOPHOLE.

One is ambiguous: MINUS for CHAR and pointer. The type of the second argument decides.

Name	*Notes*	*Classes*	*Type*
Operators (infix except as noted)			
VARTOPOINTER	@(prefix)	general	UNSAFE PROC[*T*]→[MKPOINTER[*target*~*T*.TARGET, *long*~LONG]]
EQUAL	=	general	PROC[*x*: *T*, *y*: *T*]→[BOOL]
ASSIGN	←	assignable	UNSAFE --*sometimes*-- PROC[*x*: VAR *T*, *y*: *T*]→[*T*]
PLUS*!!*	+	numeric	PROC[*T*, *T*]→[*T*]
	+	•CHAR, •pointer	PROC[*T*, INTEGER]→[*T*]
MINUS*!!*	−	numeric	PROC[*T*, *T*]→[*T*]
	−	•CHAR, •pointer	PROC[*T*, *T*]→[INTEGER]
	ambiguous		PROC[*T*, INTEGER]→[*T*]
UMINUS*!!*	−(prefix)	numeric	PROC[*T*]→[*T*]
TIMES*!!*	*	numeric	PROC[*T*, *T*]→[*T*]
DIVIDE*!*	/	numeric	PROC[*T*, *T*]→[*T*]
LESS	<	ordered	PROC[*T*, *T*]→[BOOL]
GREATER	>	ordered	PROC[*T*, *T*]→[BOOL]
	~	same as NOT	
DEREF-ERENCE*!*	↑(postfix)	ref, pointer	PROC[*r*: *T*]→[TARGET] UNSAFE PROC[*r*: *T*]→[TARGET]
REM*!*	MOD	whole number	PROC[*T*, *T*]→[*T*]
NOT	NOT(prefix)	BOOL	PROC[BOOL]→[BOOL]

continued

continued

Name	Notes	Classes	Type
Built-ins (•in addition to x.P[...], procs can be applied with P[x, ...], except as noted)			
ABORTED	ABORTED	ERROR	ERROR
ABS*!*		numeric	PROC[*T*]→[*T*]
ALL	ALL[...]: TT	ARRAY	PROC[*x*: RANGE]→[*T*]
APPLY!	APPLY[*f*, *a*]	transfer	PROC[*map*: *T*, *arg*: DOMAIN]→[RANGE]
BASE		ARRAY	UNSAFE PROC[*a*: VAR *T*]→[LONG POINTER TO UNSPECIFIED]
		descriptor	UNSAFE PROC[*a*: *T*]→[LONG POINTER TO UNSPECIFIED]
CODE		TYPE	PROC [*T*: TYPE]→[*AMTypes.Type*]
CONS	*T*[...]; coercion	ARRAY	PROC[*g*: RANGE × ...]→[*T*]
	T[...]; coercion	RECORD	PROC[*b*: FIELDS]→[*T*]
	a[...]; TT for *a*	union	PROC[*b*: FIELDS]→[*a*.]]
!	CONS[...], *z*.CONS[...]: TT	LIST	PROC[*z*: ZONE←*SafeStorage.GetSystemZone*[], *x*: RANGE, *y*: *T*]→[*T*]
DESCRIPTOR	DESCRIPTOR[*v*, ...]	row, sequence-containing record	UNSAFE PROC[*v*: VAR *T*]→ [LONG DESCRIPTOR FOR ARRAY *T*.DOMAIN OF *T*.RANGE]
	TT for *t*	descriptor	UNSAFE PROC[*base*: LONG POINTER TO UNSPECIFIED, *length*: CARDINAL, *t*: TYPE] →[LONG DESCRIPTOR FOR ARRAY CARDINAL OF *t*]
FALSE	FALSE	BOOL	bool
FIRST*!!*		discrete	*T*
first	*l.first*	LIST	PROC[*l*: *T*]→[RANGE]
FREE!	FREE[...] or *z*.FREE[...]	ZONE	PROC[*z*: *T*, *p*: NEWTYPE[NEWTYPE[*U*]]]→[]
	FREE[...] or *z*.FREE[...]	UNCOUNTED ZONE	UNSAFE PROC[*z*: *T*, *p*: NEWTYPE[NEWTYPE[*U*]]]→[]
FROMGROUND*!*	*T*[...]; coercion	subrange	PROC[*x*: GROUND]→[*T*]
ISTYPE!	ISTYPE[*x*, *U*]	general	PROC[*x*: *T*, *U*: TYPE]→[BOOL]
LAST*!!*		discrete	*T*
LENGTH		ARRAY,descriptor	PROC[*a*: *T*]→[CARDINAL]
LIST*!*	LIST[...] or *z*.LIST[...]: TT	LIST	PROC[*z*: ZONE *g*: RANGE × ...]→[*T*]
LONG	also coercion	short number	PROC[*x*:*T*]→[LONG *T*]
	also coercion	POINTER	PROC[*p*:*T*]→[LONG POINTER TO *T*.TARGET]
	also coercion	DESCRIPTOR	PROC[*p*:*T*]→[LONG DESCRIPTOR FOR ARRAY OF *T*.RANGE]
LOOPHOLE	L'E[...] only; TT for *U*	general	UNSAFE --*if u is RC*-- PROC[*t*: *T*, *U*: TYPE]→[*U*]
MAX	MAX[!..]	ordered	PROC[*T*, ...]→[*T*]
MIN	MIN[!..]	ordered	PROC[*T*, ...]→[*T*]
NARROW!	NARROW[...] : TT for *U*	general	PROC[*x*: *T*, *U*: TYPE]→[*U*]
NEW!	NEW[...] or *z*.NEW[...]	ZONE	PROC[*z*: *T*←*SafeStorage.GetSystemZone*[], *U*: TYPE] →[*r*: NEWTYPE[*U*]]
NIL	also NIL alone	variable, address, transfer	*T* or NILTYPE
ORD		enumeration	PROC[*T*]→[INT]
PRED*!!*		discrete	PROC[*x*: *T*]→[*T*]
rest	*l.rest*	LIST	PROC[*T*]→[*T*]
SIZE		general	CARDINAL
SIZE	SIZE[*T*, *n*]	general	PROC[*T*, CARDINAL]→[CARDINAL]
SUCC*!!*		discrete	PROC[*x*: *T*]→[*T*]
TOGROUND	coercion	subrange	PROC[*x*: *T*]→[GROUND]
TRUE	TRUE	BOOL	BOOL
VAL	VAL[...]; TT	enumeration	PROC[INT]→[*T*]

continued

continued

Name	Notes	Classes	Type
Funny applications			
BROADCAST	no args	CONDITION	PROC[*T*]→[]
ERROR		SIGNAL, ERROR	like APPLY
FORK!	FORK *P*[*args*]	PROC	PROC[PROC[DOMAIN]→[RANGE]]→ [PROC[DOMAIN]→[PROCESS []→[RANGE]]]
JOIN!	no args	PROCESS	UNSAFE PROC[*T*]→[RANGE]
NEW	no args	PROGRAM, POINTER TO FRAME	PROC[*p*: *T*]→[*T*]
		I: TYPE *Imp*	PROC[*p*: POINTER TO FRAME[*I*]]→[POINTER TO FRAME[*I*]]
NOTIFY	no args	CONDITION	PROC[*T*]→[]
RESTART!	no args	PROGRAM	PROC[*T*]→[]
RETURN WITH		*PrincOps.StateVector*	
RETURN WITH ERROR		ERROR	like APPLY
SIGNAL		SIGNAL	like APPLY
START!		PROGRAM	like APPLY
STOP!	no args	PROGRAM	PROC[]→[]
TRANSFER WITH		*PrincOps.StateVector*	
WAIT!	no args	CONDITION	PROC[*T*]→[]
Not in current Cedar			
APPLY!		map	UNSAFE --*sometimes*-- PROC[*map*: *T*, *arg*: DOMAIN]→[RANGE]
BINDING			
BYTESTOINSTRUCTIONS			
Cluster		general	
Default		general	PROC[]→[*T*]
DOMAIN		map	TYPE
DUMPSTATE			
HIDEEXCEPTION			
INIT		variable	
ISLONG		variable	BOOL
ISREADONLY		variable	BOOL
LOCALSTRING	coercion	STRING	PROC[[CARDINAL]]→[STRING]
MACHINEINSTRUCTIONS			
NAMES		binding, decl	
NEWEXCEPTIONCODE		exception	
NEWFRAME		decl	PROC []→FRAMETYPE[*T*]
NEWLABEL		exception	PROC []→EXCEPTION
OMITTED			
OPENPROCS			
Predicate		general	PROC[ANY]→BOOL
RANGE		map	TYPE
TARGET		reference	TYPE
TOVOID		assignable	PROC [*T*]→[VOID]
Trash		general	PROC[]→[*T*]
UNBOUND			
UNCONS	coercion	record	PROC[*T*]→[FIELDS]
UNREF		general	
VALUE		variable	TYPE
VALUEOF		variable	PROC[*T*]→[VALUE]

Table 4 – 5: Primitive procs

4.3 General types

Nearly all types belong to the *General* class (with the items enumerated below). and most belong to its subclass *Assignable* (with assignment and some related items).

4.3.1 General types

The general class has the items

T: TYPE	-- The type itself.
SIZE: CARDINAL	-- The number of words to represent a *T* value.
ISTYPE: PROC[*x*: *T*, *U*: TYPE]→[BOOL]	-- Roughly, TRUE if *x* has type *U*. See below.
NARROW: PROC[*x*: *T*, *U*: TYPE]→[*U*]	-- Converts *x* into a *U* if possible, or raises the error *Runtime.NarrowFault*. Never works for a variable type.
†LOOPHOLE: UNSAFE PROC[*x*: *T*, *U*: TYPE]→[*U*]	-- Returns the bits representing *x* as a *U*. Requires *T*.SIZE=*U*.SIZE
Predicate: PROC[ANY]→[BOOL]	-- The predicate of the type.
Cluster: BINDING	-- The cluster of the type.
MKVAR: PROC[*readOnly*, *short*: BOOL←FALSE] →[TYPE]	-- Returns the type of VAR *T* or READONLY *T*.
INIT: PROC[STORAGEBLOCK[SIZE]]→[VAR *T*]	-- Can't be called directly. See § 4.3.3.
NEW: PROC[*z*: ZONE←*SafeStorage.GetSystemZone*[], *T*: TYPE]→[*r*: REF *T*]	-- Denoted NEW[*T*] or *z*.NEW[*T*]. See § 4.3.3.
EQUAL: PROC[*x*: *T*, *y*: *T*]→[BOOL]	-- TRUE iff the bits representing *x* and *y* are identical. Missing for most unions.

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

•*Anomaly about SIZE*: There is another SIZE item in each cluster:.

SIZE: PROC[*n*: DOMAIN]→[CARDINAL]	-- Returns the size of a PACKED ARRAY [0..*n*) OF *T*. Apply with SIZE[*T*, *n*].

This proc can be useful in calculating the space required for the target of a descriptor for a packed array. You can only apply it with SIZE[*T*, *n*]; the second argument is what selects this proc. It is usually better to use a sequence.

In current Cedar the value of ISTYPE[*x*, *T*] is determined as follows. Here $T \approx U$ means that *T.Predicate*=*U.Predicate*. Two types may be unequal and yet have the same predicate if they have different clusters. Currently, the cluster can only be changed by CHANGEDEFAULT.

1) It is TRUE statically if:

   $\nabla x \approx T$, or

   one of ∇x and *T* is an opaque type, and the other is the corresponding concrete type (only in an implementation that exports the opaque type).

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

   $\nabla x \approx$ REF ANY and $T \approx$ REF *U* for any *U* except ANY, or

   $\nabla x \approx$ PROC *U* RETURNS ANY and $T \approx$ PROC *U* RETURNS *V* for any *U*, and any *V* except ANY, or

   $\nabla x \approx$ PROC ANY RETURNS *V* and $T \approx$ PROC *U* RETURNS *V* for any *V*, and any *U* except ANY, or

   $\nabla x \approx$ REF *V*, and $T \approx$ REF *V.a*, or

   $\nabla x \approx$ (LONG) POINTER TO *V* and $T \approx$ (LONG) POINTER TO *V.a*.

   $\nabla x \approx V$, and $T \approx V.a$, or

Note that the result is TRUE if *x*=NIL (except in the last case).

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

In current Cedar, NARROW[*x*, *T*] is
  IF ISTYPE[*x*, *T*] THEN *x* ELSE ERROR *e*
where *e* is
  AMTypes.NarrowRefFault[*x*, *T*.RANGE.CODE] if ISTYPE[*x*, REF ANY];
  AMTypes.NarrowFault[] otherwise.
Note that NARROW[*x*, *T*] gives a static error if ISTYPE[*x*, *T*] does (case (3) above). Note also that ISTYPE and NARROW are conveniently packaged in the safeSelect construct (§ 3.8).

Performance of ISTYPE for PROC ANY: The ISTYPE (and therefore NARROW) of a PROC type with ANY domain or range are very slow, since they use *AMTypes* to do the test, and it consults the symbol tables.

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

Anomaly for LOOPHOLE on variable types: For a variable type *T*, if *T*.LOOPHOLE is applied to a second argument *U* which is not a variable type, *U* is coerced to *U*.MKVAR[]. Thus
  {*x*: INT; LOOPHOLE[*x*, BOOL]←TRUE}
leaves $x=1$.

Every general type *T* with *T*.SIZE<2^{16} has an EQUAL proc except a variant record or union type. A variant record type has EQUAL only if its variant part is a union in which all the cases are the same size. Note that a bound variant does have EQUAL, unless it is itself a variant record. EQUAL is denoted by the infix operator =.

Anomaly for equality of variants: If *v* is a variant record and *bv* is one of its bound variants, the expression *bv*=*v* applies the EQUAL proc of the bound variant. This works even though *v* is not of the same type as *bv*.

Representation and address equality: EQUAL compares addresses in the representation of a value; it does not dereference them. Thus types like ROPE and ZONE which are represented by addresses are compared by comparing the addresses.

Restriction on EQUAL procs: A type has an EQUAL proc only if *T*.SIZE<2^{16}.

4.3.2 Assignable types

Most types (see Table 4−1 for exceptions) are in this class, which is a subclass of general (§ 4.3.1) and has items:

```
ASSIGN: UNSAFE --sometimes-- PROC       -- Returns y after storing it in x. Denoted by the
  [x: VAR T, y: T]→[T]                     right-associative infix operator ←.
TOVOID: PROC[T]→[]                      -- Discards the value. See § 3.6.
Default: PROC[]→[T]                     -- See § 4.11.
Trash: PROC[]→[T]                       -- See § 4.11.
```

As explained in § 3.7, groups and bindings are assignable if their components are. Since you cannot write these types in declarations, you have to write the constructors explicitly on the left of the ←; they are called *extractors*. E.g.,
  {*x*: INT; *y*: REAL; [*x*, *y*]←*Pair*[2, 3.4] }

Note that if *T* is not assignable, it cannot be used as the type of a proc argument or result, since arguments and results are passed by assignment to variables.

ASSIGN is unsafe for

unions and variant records;

assigning an unspecified type to anything except itself.

In a CHECKED block, a proc value cannot be assigned if it is local to another proc rather than to an implementation (since this could lead to a dangling reference). This is checked at runtime.

Restriction on ASSIGN procs: A type *T* has ASSIGN only if $T.SIZE<2^{16}$.

Representation of ASSIGN: Since it involves a VAR parameter, an ASSIGN proc cannot be written in current Cedar. The primitive ASSIGN procs simply copy the bits of *y*'s representation into the variable *x*, unless some of them represent REFs. In this case the assignment involves reference-counting if *x* is in counted storage; see § 4.5 for details.

CHANGEDEFAULT can take any type and produce a new one which is identical except for the cluster items named *Default* and *Trash* which determine how default values are supplied when a binding value is coerced to a decl type; see § 4.11 for details.

4.3.3 Variable types

40 **varTC** ::= (| READONLY | VAR) t | ANY　　　　(VAR | READONLY | VAR) t | ANY
In ll. 45 – 48. ANY only in refTC. VAR only in interface decl.

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

VAR *T*
READONLY *T*
SHORT VAR *T*
SHORT READONLY *T*

You cannot denote these types in current Cedar except in a few contexts, but they are fundamental to an understanding of how it works nonetheless. The basic facts about variables in Cedar are given in § 2.3.3. A variable type is made by the MKVAR proc in the cluster of the non-variable type (§ 4.3.1).

The variable class is a sub-class of general (§ 4.3.1) and hasNIL (§ 4.3.7), and has items:

```
VALUE: TYPE;                          -- (VAR U).VALUE=U; T.VALUE.MKVAR=T.
VALUEOF: PROC[T]→[VALUE];             -- A coercion.
ISLONG: BOOL;                         -- FALSE for short vars.
ISREADONLY: BOOL;                     -- TRUE for readonly vars.
VARTOPOINTER: UNSAFE PROC[T]→         -- Apply by prefix @.
      [MKPOINTER[range~T.VALUE, long~ISLONG, readOnly~ISREADONLY]];
```

Furthermore, *T* inherits the cluster of *T*.VALUE. The procs are not modified, since the VALUEOF coercion provides them with *T*.VALUE arguments where needed. There is one exception: the *component* procs described below are replaced by procs which return variables instead of values.

The INIT proc (§ 4.3.1) converts a block of storage into a legal variable of type *T*, at least in theory. In fact, it is currently a no-op except for

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

Bound variants; the tag field is set appropriately.

INIT cannot be supplied or called directly by the user; it can only be called indirectly, from NEW.

The NEW proc (§ 4.3.1) calls on the zone *z* to obtain a block of storage of size *T*.SIZE (§ 4.5.2), and applies *T*.INIT to convert the block into a VAR *T*; call it *x*. Then if *T.Default* exists, NEW calls it and assigns the result to *x*.

Caution on finalization: A variable type may have a finalization proc, which is called when no client references to a variable remain; see *SafeStorage*. This proc is executed *concurrently*, and must therefore provide proper synchronization.

Restriction on NEW: A type has a NEW proc only if T.SIZE<2^{16}.

•The @ operator (VARTOPOINTER) does not work on a variable *v* which is a component of a packed array (no matter what its type is), or component of a record if *v* is represented in less than 16 bits. These restrictions are machine-dependent, and @ is unsafe; avoid it if at all possible.

Composite variables and component procs

MKVAR commutes with a composite type, cross type or declaration constructor. For example,
  VAR [*a*: INT, *b*: REF ANY]
is equal to
  [*a*: VAR INT, *b*: VAR REF ANY]
and likewise for READONLY. Similarly, VALUEOF commutes with the component procs for values of these types, so that *v.a*.VALUEOF = *v*.VALUEOF.*a* if *v* has the variable type just mentioned.

Another way to think of this is that one of these variables is the *composite* of a set of variables, one for each component. If *T* is a record type, row type, cross type or declaration, then a *component proc* in *T*'s cluster which extracts a component of a *T* value (e.g., a field proc, APPLY which subscripts the array, etc.) has a counterpart in the cluster of VAR *T* which extracts a variable. Thus if *a* and *r* are array and record variables, then *a*[*i*] and *r.f* are also variables which can be modified by ASSIGN.

4.3.4 Opaque types

40.1**paintedTC** ::= typeName PAINTED t   REPLACEPAINT[in~t, from~**typeName**]
  typeName must be an opaque type, t a recordTC or enumTC.

Example

```
HV: TYPE~Interface.HistValue PAINTED          -- See 13 for use.
  RECORD[...]
```

An opaque type declaration in an interface is the only way to declare a type parameter (except for the interface parameters declared in the DIRECTORY). Such a type parameter is called *opaque*. The type of an opaque type must be TYPE[ANY] or TYPE[*n*]; thus you can write
  T: TYPE[ANY]
or
  T: TYPE[*n*]
in an interface. These expressions are non-functional; each generates a new mark, and a type can be exported to *T* (i.e., has the type denoted by the TYPE[ANY] or TYPE[*n*] expression which declares *T*, and hence is an acceptable argument value for this formal parameter) only if it carries that mark. A type exported to *T*: TYPE[*n*] must have additional properties described below.

You attach one of these marks to a type using a paintedTC. The type being painted (t in the rule) must be a recordTC or enumTC. The paint comes from the typeName, which must be an opaque type; it replaces the new paint which the constructor would have supplied.

Any record or enumeration type can be painted from a type declared TYPE[ANY]; only a type so painted can be supplied as the argument for the declaration *T*: TYPE. *T* is called *fully opaque.*

A type *V* can be painted from a type *U* declared TYPE[*n*] if:

  V.SIZE = *n*.

  V is a recordTC or enumTC and has standard NEW, INIT, ASSIGN, EQUAL and ISTYPE procs. All the assignable primitive types do except

the RC types (§ 4.5.1);

bound variant types (§ 4.6.2);

types produced by a defaultTC[55];

composite types with a component that has a non-standard NEW, INIT, ASSIGN, or EQUAL proc.

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

Only a type painted with *U* can be supplied as the argument for the declaration *U*: TYPE[*n*]. *U* is called *n-opaque.*

Example: For the interface:

```
I: DEFINITIONS~{
   FO: TYPE[ANY];
   nO: TYPE[SIZE [INT]] }
```

this implementation is suitable:

```
IImpl: PROGRAM EXPORTS I~{
   FO: PUBLIC TYPE~I.FO PAINTED RECORD[a: INT, b: ROPE];
   nO: PUBLIC TYPE~I.nO PAINTED RECORD[INT];
```

Note that replacing INT by REF ANY in *nO* would not work, since this does not have standard ASSIGN and INIT procs.

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

The cluster of an *n*-opaque type *U* has VAR, NEW, INIT, ASSIGN, EQUAL and ISTYPE procs (the last not yet implemented). Thus these operations can be done on a *U* value. As a consequence, a *U* value can be passed as a parameter and declared.

Restriction on values of opaque types: All instances of any interface produced by applying an interface module which declares an opaque type *T* must supply a type value with the same predicate for *T* if they supply any value at all; this value is called the *standard implementation* of *T*. Because of this restriction, clients can safely interassign values of type *T*, no matter how obtained. In addition, it is safe for any exporter of *T* to convert a value of type *T* to a value of the corresponding concrete type, and conversely. The restriction arises from the fact that the type is identified by its mark; hence the same mark must not be assigned to two different types.

Anomaly on referencing opaque types: It is not necessary to import an interface to refer to an opaque type declared in that interface (because of the above restriction).

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

Performance of converting between opaque and concrete types: The conversion between an opaque type and the corresponding concrete one costs nothing at runtime.

4.3.5 Interface types

The type of an interface module is $d\rightarrow[n_m$: TYPE $n_m]$, where *d* is the declaration given in the DIRECTORY; when the module is applied, the result is an interface, with type TYPE n_m. The interface is itself a type. A value of that type is an instance exported by an implementation module that exports the interface. These classes have no standard items (except an implementation instance, which has COPYIMPLINST), but the clusters of these types do have the items bound or declared in the interface. Thus you cannot do anything with these types except

use them in a DIRECTORY, IMPORTS, SHARES, or EXPORTS;

select items from the cluster using dot notation;

use an interface type in an open.

See § 3.3.4-5 for complete information.

4.3.6 ANY

The type ANY is implied by every type. ANY cannot be the type of a d or b item, and an expression never has syntactic type ANY unless it is an ERROR application. ANY can only be used as the target of a REF or as the domain or range of a transfer type. A value whose type involves ANY cannot be dereferenced or applied, since these operations would yield an expression with syntactic type ANY. However, it can be narrowed (§ 4.3.1).

4.3.7 HasNIL

Variable, address and transfer types are in this class, which is a subclass of general (§ 4.3.1), and gives them one thing in common:

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

There is a universal value NIL (with type NILTYPE) which can be coerced into any particular *T*.NIL.

4.4 Map types

The map class is a subclass of assignable (§ 4.3.2) and has the items:

DOMAIN: TYPE;	-- Domain type for the mapping.
RANGE: TYPE;	-- Range type for the mapping.
APPLY: PROC[*map*: *T*, *arg*: DOMAIN]→[RANGE]	-- *map*[*arg*] is sugar for *map*.APPLY▸*arg*. In current Cedar, you can write this explicitly only for transfer types.

Usually DOMAIN and RANGE are declarations, so that bindings can be used for the arguments and results. Application is denoted by brackets (*map*[*arg*]), or explicitly (APPLY[*map*, *arg*]) for transfer types only.

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

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

Transfer types: procs, and their close friends processes, signals, errors, ports and programs; applying a transfer value executes the body of some λ-expression (§ 4.4.1). § 2.2.1 and § 2.6 tell all about applying procs.

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

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

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

In addition, many subclasses of TYPE have APPLY procs with assorted meanings (§ 4.8).

4.4.1 Transfer types

41 **transferTC** ::= ?safety4 xfer ?drType — MKXFERTYPE[drType, flavor~xfer]

41.1 **xfer** ::= PROCEDURE | PROC | PROGRAM | PORT | PROCESS | SIGNAL | ERROR

42 **drType** ::= ?fields$_1$ RETURNS fields$_2$ | fields$_1$ — domain~**fields**$_1$, range~**fields**$_2$

No domain for PROCESS. In 3, 41.

43 **fields** ::= [d11, ...] | [t, ...] | ANY

ANY only in drType. In 42, 50, 52.

Examples

```
Enumerate: PROC[
    l: RL,
    p: PROC[x: REF ANY] RETURNS [stop: BOOL]]
  RETURNS [stopped: BOOL];
p2:PROCESS RETURNS[i:INT]←FORK stream.Get;
failed: ERROR [reason: ROPE]~CODE;
```

Transfer is a subclass of map (§ 4.4) and of hasNIL (§ 4.3.7). The subclasses of transfer are PROC, PORT, PROGRAM, PROCESS, SIGNAL, and ERROR. These types are constructed by transfer type constructors which begin with those words, or in the kernel by the MKXFERTYPE constructor. What they have in common is that application executes the body of some λ-expression, but the transfer class adds no items to the map class.

One transfer type *T* implies another *U* if

> The subclass is the same.
>
> *T*.RANGE implies *U*.RANGE.
>
> *U*.DOMAIN implies *T*.DOMAIN.

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

> They have the same names, or each has only one name, and
>
> The corresponding types imply each other.

I.e.

> If *n*: *T* is in *D* and *n*: *U* is in *E*, then $T \Rightarrow U$.
>
> If *D*=[*m*: *T*] and *E*=[*n*: *U*], then $T \Rightarrow U$.

See § 2.2.1F. *D* implies a cross type *T* if *D*.T implies *T*; in this case *T* also implies *D*.

Either the domain or the range of a transfer type (or both) can be ANY. A value of these types cannot be applied, but it can be narrowed to a specific transfer type (§ 4.3.1).

Representations for transfer types are given in the *PrincOps* interface. They tend to change when the machine architecture changes.

An attempt to apply a NIL transfer value results in the error *Runtime.UnboundProc*.

PROC types

The PROC class is a subclass of transfer (§ 4.4.1) with no additional items. In the kernel, a new proc value is made by evaluating a λ-expression. In current Cedar, it is made by a binding of the form

    P: T~{...}

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

Assignment of a proc may lead to a dangling reference, if the proc value is for a local proc *P* and it survives the return of *P*'s enclosing proc. In a checked block any assignment of a local proc value is disallowed (except the assignment of a parameter value to a parameter variable).

PROGRAM types

The program class is a subclass of transfer (§ 4.4.1), and also has items:

•STOP: PROC[]→[]	-- Apply by STOP. Legal only if RANGE=[]. Denoted by STOP, since it takes no arguments.
•RESTART: PROC[*T*]→[]	-- Apply by RESTART *P*. Legal only if RANGE=[].
•COPYIMPLINST : PROC[*p*: *T*]→[*T*]	-- Apply by NEW *P*;

Their use is not recommended; for details, consult a wizard. For more on implementations, see § 3.3.2.1 and § 3.3.5. COPYIMPLINST makes a copy of the implementation module for which *p* is the program proc, and returns the program proc of the copy. See § 4.5.3 for more details.

The syntax for applying a program *P* is
START *P*[*args*]
•The START may be omitted, so that it looks like an ordinary application; avoid this feature. This expression's type is ∇*P*.RANGE.

A program value is obtained from the frame of an implementation, which always includes the item:
Imp: PROGRAM *T*~*PP*;
where *Imp* is the name of the module, *T* its drType, and *PP* its program proc; see § 3.3.5. This value can be accessed:

from an interface exported by *Imp* which declares *Imp* as a PROGRAM *T*;

as *F.Imp*, where *F* is a POINTER TO FRAME of the implementation;

as the CONTROL item returned by the module.

•*PORT types*

Use of ports is complex, unsafe and not recommended. See chapter 9 of the Mesa manual if necessary.

PROCESS types

The process class is a subclass of transfer (§ 4.4.1) with no other items, but *Process.Abort*[*P*] raises the ERROR ABORTED in *P*. § 4.10 describes Cedar's facilities for concurrent programming.

A process always has DOMAIN=[]. The syntax for applying a process *P* is
JOIN *P*
This expression's type is ∇*P*.RANGE. •The JOIN may be omitted, so that it looks like an ordinary application; avoid this feature.

A process value is obtained from:
FORK: PROC[PROC[DOMAIN]→[RANGE]]→[PROC[DOMAIN]→[PROCESS []→[RANGE]]]
The syntax for using this is
FORK *P*[*args*].
The FORK *P* returns a proc which when applied to *args* creates a new process, starts it running, and returns it.

Anomaly for FORK: Note the peculiar parsing (FORK *P*)[*args*]. You cannot write FORK *P* alone to get hold of the process-creating proc.

SIGNAL and ERROR types

These are subclasses of transfer (§ 4.4.1) with no other items.

In the kernel, a new signal or error value is made by applying NEWEXCEPTIONVALUE. In current Cedar, it is made by a binding of the form
E: *T*~CODE
in a d or b, where *T* is a signal or error type. The effect is to construct a unique exception value, not equal to any other. An enable choice which catches this value will only catch an exception raised with this value; it cannot catch some other expression with the same name.

Anomaly for CODE: Unfortunately, CODE does not yield a unique value at each execution. The value is only unique to the textual occurrence of CODE and the module instance; if CODE appears inside a proc, the same value is produced each time the proc is applied. Thus care may be needed if the proc is recursive.

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

§ 2.6.2 and § 3.4.3 explain errors in detail. A signal is exactly like a proc, except that the closure that is executed is obtained from the statement of an enable choice; see § 3.4.3A for details.

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

4.4.2 Row and descriptor types

44 **arrayTC** ::= ?★PACKED ARRAY ?t_1 OF t_2	MKARRAY[domain~t_1, range~t_2]
45 **seqTC** ::= ?★PACKED SEQUENCE tag[53] OF t Legal only as last type in a recordTC or unionTC.	MKSEQUENCE[domain~**tag**, range~t]
45.1†**descriptorTC** ::= ?LONG DESCRIPTOR FOR varTC[40] varTC must be an array type.	MKARRAYDESCR[arrayType~**varTC**]

Examples

Vec: TYPE~ARRAY [0..maxVecLen) OF INT;	
Chars: TYPE~RECORD [text: PACKED SEQUENCE	-- A record with just a sequence in it.
len: [0..INTEGER.LAST] OF CHAR]; ch: Chars;	-- *ch.text*[*i*] or *ch*[*i*] refers to an element.
v: Vec~ALL[0];	
dV: DESCRIPTOR FOR ARRAY OF INT~ DESCRIPTOR[v];	

A row value provides an indexed set of values of an arbitrary type, called the *components* of the row: application maps an index into the corresponding value. Usually the values are variables, so that assignment to a component is possible. A descriptor is an unsafe pointer to a row which includes a subrange of the domain or index type in the descriptor value; thus values of the same descriptor type can point to rows of different sizes. Because all the row types use the same representation for the set of values, it is possible to make a descriptor from any row value.

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

The PACKED argument of the row type constructors governs the representation of a row whose range type is represented in ≤8 bits. See the discussion of representation below. It also disallows the use of @ on an element of the row.

The row class is a subclass of map (§ 4.4) and also has the item:

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

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

Representation of rows: A VAR row value is represented by a contiguous block of words. If PACKED=FALSE, each element VAR occupies *T*.RANGE.SIZE words, and the successive elements occupy consecutive blocks of storage, beginning with the one indexed by *T*.DOMAIN.FIRST. If PACKED=TRUE and a *T*.RANGE value is represented in $n \leq 8$ bits, each element occupies $2^{\text{CEILING}[\text{LOG}_2[n]]}$ bits, i.e. 1, 2, 4 or 8 bits depending on its size; PACKED has no effect on the representation for ranges with bigger values. Note that the entire representation of a packed array may be smaller than a word, and need not be word-aligned in another packed array or in a record. This is the entire representation of an array value; a sequence value also has a tag field, which is represented like a component of the containing record.

Restriction on row sizes: A row type must have $T.\text{SIZE} < 2^{28}$ and $T.\text{RANGE.SIZE} < 2^{16}$.

It is not possible to obtain a REF to a row component; this is because the implementation of both reference counting and REF ANY discrimination requires more information about each VAR than is available for an array element. If the row is PACKED, it is not possible to apply @ to obtain a pointer to an element either.

Performance of row arguments and results: Passing a row as an argument or result entails copying the representation. Unless the row is quite small, this is expensive. It is usually better to pass a REF. Very large rows (say, more than 100 words) should not be declared in a block, since this results in large frames which consume the 64k words of frame space. Instead, they should be allocated with NEW.

4.4.2A ARRAY types

An array is a row with an element for each value in the domain; its APPLY proc is a total function. The advantages of this are that no space is needed to store the length of an array, and any bounds checking on a subscript is done against constant values (as part of narrowing the subscript to the domain type, which is usually a subrange). The disadvantages are that a given proc, written to deal with a given array type, cannot be used on other arrays of different lengths, since there is no way in current Cedar to parameterize the proc with a type. In this case it is better to use a sequence (§ 4.4.2B).

The array class is a subclass of row (§ 4.4.2) and of assignable (§ 4.3.2) if RANGE is assignable. It also has the items:

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

CONS takes a group of values, one for each element of the array, into an array value. Note that the argument of CONS may have omitted values, which are filled in if possible by the defaulting coercion for *T*.RANGE. If the index type is enumerated, CONS takes a binding, with one element named *n* of type *T*.RANGE for each index value *n*. In current Cedar you can't write *T*.CONS. Instead you write *T* itself; i.e., *T*[...] for *T*.CONS[...]. Because CONS is a coercion from group or binding to array, you can omit the *T* whenever the group or binding appears as an argument or in a binding; see § 4.13. Examples:

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

Anomaly about ALL: ALL replicates its argument in all the elements of an array. In current Cedar you can't write *T*.ALL. Instead you just write ALL; it must be in an argument or binding. Unlike most built-ins, ALL is not sugar for dot notation. If the range type permits it, you can write ALL[TRASH] to trash all the elements.

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

BASE returns the address of its VAR array argument. It is mostly useful for writing storage allocators. The resulting LONG POINTER TO UNSPECIFIED can also be passed to DESCRIPTOR to yield a descriptor for a different type of array; obviously this is dangerous.

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

4.4.2B SEQUENCE types

A sequence is like an array, but each sequence value includes a *tag* value which specifies the number of elements in that sequence, i.e. the values of the domain type for which APPLY is defined. Note that APPLY for a sequence is usually *not* total. If the domain type is *T* and the tag value is *v*, then APPLY is defined for [*T*.FIRST..*v*). Usually *T* is NAT, so that *v* is the number of elements in the sequence, and the elements are indexed by 0, 1, ..., *v*−1.

In current Cedar there are many restrictions on the use of sequences. A sequence type is defined by a sequenceTC[45]; it is not a first-class type, and can *only* appear as the type of the last field of a variant record or union (§ 4.6.2). The items in the cluster of a sequence type are just those for a row; they are inherited by the containing variant record, which is the type a program normally deals with.

A record type *T* containing a sequence field is a variant record. *T* is a first-class type which can be bound to a name, but unlike a union-containing record it cannot be used where type[36] appears in the grammar, except in a refTC[46] (or pointerTC[48]). The only items in the cluster of *T* are the ones of the variant record class, and those inherited from the row class of the contained sequence:

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

The tag of a sequence is readonly.

Hence the only uses of *T* are:

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

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

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

```
NEW: PROC[z: ZONE←SafeStorage.GetSystemZone[] -- Denoted NEW[T[n]] or z.NEW[T[n]]
          T: TYPE]→[r: REF GENERAL]
SIZE: CARDINAL
GENERAL: TYPE                                 -- The type of the unspecialized sequence.
```

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

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

Examples:

```
StackRep: TYPE~RECORD[
   top: INT← −1,
   item: SEQUENCE size: NAT OF T];
Number: TYPE~RECORD[
   sign: {plus, minus},
   magnitude: SELECT kind: * FROM
      short=>[val: [0..1000)],
      long=>[val: LONG CARDINAL],
      extended=>[val: SEQUENCE length: NAT OF CARDINAL]
      ENDCASE];
rs1: REF StackRep←NEW[StackRep[100]];                       -- rs1.top= −1, rs1[i] is trash.
rs2: REF StackRep←NEW[StackRep[100]←[top~3, item~TRASH]];  -- rs2.top=3, rs2[i] is trash.
rn1: REF Number.extended←NEW[Number.extended[2*k]];
-- rn1[2]=rn1↑[2]=rn1.item[2]=rn1↑.item[2], but all start out trashed.
```

•A sequence may have a COMPUTED tag, with the same meaning as for unions: no tag field exists, no bounds checking is possible so that application is unsafe, and the cluster has no DESCRIPTOR proc. You can still compute the address of the sequence with @ and use the unsafe three-argument form of DESCRIPTOR (§ 4.4.2.3). Example:

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

A sequence may not have an OVERLAID tag, and * cannot be used for the tag type.

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

There is a predefined sequence TEXT; see Table 4−2 for its declaration. There are literals of type REF TEXT, denoted as in rule 57 by the characters of the literal enclosed in doublequotes. Such a literal is shorthand for a constructor (which you couldn't actually write in current Cedar, since it lacks constructors for sequences). REF TEXT can be used where efficiency is critical; for general purposes use *Rope*.ROPE.

•There are also unsafe predefined types LONG STRING and STRING; see Table 4–2 for their declaration. They are described here for completeness, but should not be used. These types are pointers to a *StringBody* type also given in Table 4–2.

Anomaly for StringBody: In spite of the declaration, *StringBody* behaves like a sequence with tag *maxlength* and sequence *text*. Thus *z*.NEW[*StringBody*[*n*]] returns a STRING or LONG STRING with *maxlength*=*n*; if *s* is a STRING or LONG STRING, *s*[*i*] indexes its *text*, etc. You can also use *s.text*, as with sequences, but this is not recommended: because of the definition, *s.text*[*i*] is never bounds-checked (use *s*[*i*]), and DESCRIPTOR[*s.text*] describes an array of length 0 (use DESCRIPTOR[*s*↑].

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

LOCALSTRING: PROC[[*length*: CARDINAL]]→[STRING] -- A coercion.

Because this is a coercion, you can write

s: STRING~[20]

to obtain a local string of length 20. Of course, the storage will be freed when the proc frame is freed, and a dangling reference may remain. This construct is legal only in declarations as the e of a defaultTC.

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

†4.4.2C Descriptor types

A descriptor is a pointer to a row value which includes a subrange of the row's domain as part of the descriptor value. A proc which takes descriptors rather than rows or REFs to rows can deal with rows of different sizes. Because a descriptor is like a pointer, there are short, long and relative descriptors which are exactly analogous to short, long and relative pointers; see § 4.5.1 and § 4.5.4 for details.

Style for rows of variable length: Applying a descriptor is unsafe. It is generally better to use a REF to a sequence-containing record.

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

Descriptor is a subclass of row (§ 4.4.2) and address (§ 4.5). Like array, it has the items:

```
LENGTH: PROC[a: T]→[CARDINAL]          -- Returns the cardinality of the subrange in a.
BASE: UNSAFE PROC[a: T]                -- Returns the address of a's first element.
        →[LONG POINTER TO UNSPECIFIED]
```

Like pointer, it has:

```
TARGET: TYPE                           -- The type of the arrayType used to
                                          make the descriptor.
```

In addition, there is an unsafe and untypesafe proc for making a descriptor with RANGE=CARDINAL from a LONG POINTER:

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

d.LENGTH=*length* and *d*.BASE=*base*.

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

There is a compile-time coercion from LONG DESCRIPTOR to DESCRIPTOR, which works exactly like the similar coercion from LONG POINTER to POINTER (§ 4.5.1B).

•†4.4.3 *BASE POINTER types*

A base pointer *bp* is like an ordinary pointer, except that it has an APPLY operation which maps a relative pointer *rp* (see § 4.5.4) into an ordinary pointer *p*. Its class is a subclass of pointer (§ 4.5.1B) and approximately a subclass of map (§ 4.4), but with the items:

APPLY: UNSAFE PROC[*bp*: *T*, *rp*: DOMAIN]→[*p*: *rp*.TARGET]
DOMAIN: *T* RELATIVE POINTER

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

In other respects, a base pointer is like an ordinary pointer; indeed, it is a subclass of pointer. Thus, it has a target type of its own, and can be dereferenced to yield a value of that type. This allows it to point to a record or other variable at the start of the region. Note that the base pointer's target has nothing to do with the range of its APPLY, which is the target of the relative pointer it is applied to; unlike other map types, a base pointer has no RANGE of its own.

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

Representation of base pointers: The APPLY proc is

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

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

Anomaly for relative array descriptors: A relative array descriptor (obtained by using a descriptor type as the *range* argument of the type constructor) doesn't quite work this way, since it uses the bounds in the descriptor, rather than in TARGET.DOMAIN, to check the subscript.

4.5 Address types

46 **refTC** ::= REF (varTC[40] |)	MKREF[target~(**varTC** | ANY)]
47 **listTC** ::= LIST (OF varTC[40] |)	MKLIST[range~(**varTC** | REF ANY)]
48 †**pointerTC**::= ?LONG ?ORDERED ?•BASE POINTER ?•subrange[25] (TO varTC[40] |) |	MKPOINTER[target~(**varTC** | UNSPECIFIED), subrange~**subrange**] |
•POINTER TO FRAME [n]	**n**
Subrange only in a relativeTC; no typeName[37] on it.	
49 •†**relativeTC** ::= typeName[37] RELATIVE t	MKRELATIVE[range~t, baseType~**typeName**]
t must be a pointer or descriptor type, typeName a base pointer type.	

Examples

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

UnsafeHandle: TYPE~LONG POINTER TO Vec[44];

Address is a subclass of assignable (§ 4.3.2) and of hasNIL (§ 4.3.7). It has no items of its own. An address value is the address of a variable, i.e., of a block of storage.

Storage is a precious resource which must be reclaimed when it is no longer needed, i.e., when the variable it represents will no longer be touched by the program. Cedar provides *safe storage* which does this reclamation automatically, and *unsafe storage* which must be reclaimed explicitly by the program. A checked program (§ 3.4.4) deals only with safe storage, and need not be concerned with how storage is reclaimed, or how things can go wrong, except for one point discussed in the next paragraph. If you write only checked programs, you can skip to § 4.5.1. An unchecked program must maintain the safety invariants, in order to ensure that the Cedar system continues to function. These invariants are given in the remainder of this sub-section.

Cedar has two *garbage collectors* for reclaiming safe storage. The *incremental collector* runs continuously and reclaims storage without stopping other computations for more than a few milliseconds at a time. The *trace-and-sweep* collector runs only when invoked, and stops other computations for many seconds. The disadvantage of the incremental collector is that it cannot reclaim a cyclic structure, even if that structure can no longer be reached by the program. Therefore, a production program, especially a real-time or interactive one, should break the cycles in its structures when they are no longer needed. The *package finalization* mechanism is often helpful in doing this. It and other features of Cedar safe storage are described in the *SafeStorage* interface.

Anomaly in garbage collection: It is possible that an unreachable variable will not be reclaimed because it appears to be pointed to by some double-word quantity in a frame which is not actually a REF. This can happen because the collectors cannot tell which double-words in a frame are REFs, and hence proceed conservatively.

Definitions

To state the safety invariants, we need some definitions.

A *safe variable* (SV for short) is a frame or a *counted* variable, i.e., one allocated by *z*.NEW, where *z* is a ZONE (§ 4.5.2). A *safe reference* (SR for short) is a transfer or REF value. A SR is the only legitimate way of addressing a SV, and furthermore, a SR can legitimately only be stored in a SV. A *reference-containing* type (RC for short) is a SR type, or a composite type with a RC component.

A SV is *reachable* if:

> it is the *process array* or, in current Cedar, the global frame of a module,
>
> or a SR which points to it is stored in some reachable SV.

The collector tries to reclaim safe storage when it is no longer reachable.

A SV *v* is *good* if:

> It overlaps no variable of another type.
>
> If its VALUE type is RC, then *v*.VALUEOF is good.

A SR is *good* if it points to a good SV of the proper type, or is NIL. A composite RC value is good if each of its RC components is good.

The idea is that if:

> new address values are generated only by NEW or frame allocation, and
>
> these allocators always return an SR which is the address of a SV that doesn't overlap any other SV, and
>
> SR values never get damaged or mistyped,

then by keeping track of the SRs the collector can know about all possible ways of reaching an SV. If there are no ways, the SV can be freed.

For the purpose of this analysis, we assume that every value is held in some variable; the fact that some values are constant is not important here. Storage can be modified only by an ASSIGN proc for some variable. Hence the behavior of ASSIGN determines how values can change. A composite variable (§ 4.3.4) is made up of other variables; in Cedar record, union and row variables are composite. ASSIGN for a composite variable is simply a sequence of ASSIGNs for the components. Therefore the remaining analysis considers only non-composite variables.

Safe storage main invariants

Cedar safe storage depends on three invariants. These in turn depend on some local invariants (L1-L4), and some properties of the Cedar primitives (P1-P3) given below. The proofs of the main invariants follow these definitions.

S1) Every SV is good.

S2) Every SR is good.

S3) A SV is not freed if there is an SR for it in some other SV.

Local invariants

L1) No variable of another type overlaps an existing SV. The allocator ensures that no SV will do so, because NEW[*T*] returns the address of a block of at least *T*.SIZE words, none of which is part of an existing SV. Similarly, applying a closure allocates a block of unused words at least as large as the frame. *Unchecked code must ensure this for other variables.*

L2) Assignment to a SV works, and is type-correct: the value being assigned has the VALUE type of the SV, and the assignment leaves it as the value of the SV (P1). *Unchecked code must ensure that only a SR of the proper type is assigned to a SV.* In particular, it must not produce a SR value out of thin air, unless it is known that there is an equal existing reachable SR value.

L3) A counted SV is reached only through a REF: the allocator which creates a counted SV returns only a REF to it. There is no safe operation for obtaining a counted SV except from a REF. *Unchecked code must not produce a counted SV except from a* REF.

L4) An SR which points to a frame (i.e., a transfer SR) is stored only in a frame which is freed first. A checked assignment cannot assign a transfer SR unless it points to a global frame, which is never freed (except by an unsafe operation); when a SR is bound to a name, it must be from the same or a larger scope. *Unchecked code must not preserve a transfer SR after its frame has been freed.*

Primitive properties

P1) ASSIGN(*to*~*SV*, *from*~*SR*) leaves *SV*.VALUEOF=*SR* and affects no other non-overlapping variable. If *SV* is counted (i.e., came from dereferencing a REF) it updates the count correctly.

P2) The collector does not free a counted SV holding a SR until the value of the SR is NIL.

P3) The collector does not free a SV until no SR on the stack points to it and it has a zero reference count.

P4) A SR is stored only in a SV.

Proof of main invariants

S1) Every SV *v* is good. Proof by induction.
  Basis: A SV is good when created by NEW.
  Induction: There are three ways *v* might cease to be good:
    Another variable might come to overlap it, but this doesn't happen (L1).

If *v*.VALUEOF is SR, it might change:

By assignment to it, but ASSIGN replaces the value in *v* with another SR value (L2), and this other value is good (S2).

By an assignment to some other variable which clobbers *v*, but no variable of another type overlaps *v* (S1), and no assignment to a non-overlapping variable can clobber *v* (P1).

If *v*.VALUEOF is SR, it might cease to be good, but it points to a good SV (S2) which remains good (S1).

S2) Every SR is good. Proof by induction.

Basis: the values produced by NEW and by applying a closure are good.

Induction: the other source of SRs is SVs (P4), and these are good (S1). Furthermore, an SV for which an SR exists is not freed (S3), so the SR remains good.

S3) A SV *v* is not freed if there is an SR for it. Proof by case analysis.

A) Not by the reference counting garbage collector, because:

This collector frees *v* only if no SR on the stack points to it, and it has a zero reference count (P3).

An SR can be stored only in a SV, i.e., on the stack or in a counted SV (P4).

The number of SRs pointing to *v* in counted SVs is equal to the reference count for *v*, by induction:

Basis: Both start at zero.

Induction: There are three ways the number of counted SRs pointing to *v* can change:

ASSIGN to a counted SV, which updates the count correctly, because a counted SV is reached only through a REF (L3), and any assignment through a REF updates the count correctly (P1).

ASSIGN to some variable *w* of another type, but *v* is good, hence overlaps no variable of another type (S1), hence is not affected by ASSIGN to *w* (P1).

Freeing a counted SV, but it is not freed until its value is NIL (P2).

B) Not by the trace-and-sweep garbage collector, because:

It implements the definition of reachability. Note that the collector sets SRs for unreachable SVs to NIL, thus breaking circular structures.

C) Not by the frame deallocator, because:

A frame is either permanent (a global frame), or an SR which points to it is stored only in a frame which is freed first (L4).

It is possible to obtain a variable without going through a SR value by using an unsafe *pointer-containing* type (PC for short). The non-composite PC types are:

pointer (which includes POINTER TO FRAME, string and uncounted zone);

descriptor;

A program which obtains a variable from a PC value (by dereferencing a pointer, applying a string or descriptor, or using NEW or FREE for an uncounted zone) must maintain the safety invariants L1-L4.

4.5.1 Reference types

This class is a subclass of address (§ 4.5) and has the items:

```
TARGET: VARIABLE.Type                          -- Always a variable type.
DEREFERENCE: PROC[r: T]→[T.TARGET]             -- Denoted by r↑
APPLY: PROC[r: T, arg: T.TARGET.DOMAIN]→       -- Inherited from the target type if it has APPLY.
        [T.TARGET.RANGE]
f: PROC[r: T, arg: T.TARGET.f.DOMAIN]→         -- Inherited from the target type for each proc f
        [T.TARGET.f.RANGE]                        in its cluster; see below.
```

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

If the target has an APPLY, DESCRIPTOR, WAIT, NOTIFY or BROADCAST proc, or any record field procs in its cluster, these are inherited by the reference type (except that APPLY is not inherited by a BASE POINTER, which has its own APPLY: see § 4.4.3). The value of an inherited *f* is

λ [*r*: *T*, *arg*: *T*.TARGET.*f*.DOMAIN] IN *r*↑.*f*[*arg*]

In other words, the address is dereferenced, and then the target's *f* is applied. The effect is that a reference to an array or proc can be applied without explicit dereferencing, a reference to an array can be turned into a descriptor, a reference to a condition can be used to do a WAIT or whatever, and a reference to a record can be used to select a field.

Procs which get into a cluster by being in an interface instance are also inherited in this way, but this is not useful, since they are *not* modified to dereference their reference argument: this is a deficiency. To compensate for this, you can define such procs to take a REF *T*, so they will be useful when inherited from *T.Cluster* to (REF *T*).*Cluster*.

4.5.1A REF types

The REF class is a subclass of reference (§ 4.5.1) and has no additional items. A REF value can be safely created only by a NEW proc. Every general type except union has one of these (§ 4.3.1).

The type VAR ANY may be the target of a REF; it cannot appear anywhere else. This REF type is denoted REF ANY, or simply REF. It is implied by every REF type. ISTYPE can be used to test the particular REF type of a REF ANY value, and NARROW can be used to convert a REF ANY value into a REF *T* value (§ 4.3.1). These two operations are combined in a convenient way by safeSelect[32] (§ 3.8). REF ANY does not have a DEREFERENCE proc, and of course there are no procs for it to inherit from the target.

LIST types

The LIST class is a subclass of REF, and has items:

```
RANGE: VARIABLE.Type;                            -- Always a variable type.
first: PROC[l: T]→[RANGE];                       -- Denoted l.first, not first[l]
rest: PROC[l: T]→[T];                            -- Denoted l.rest, not rest[l]
CONS: PROC[z: ZONE←SafeStorage.GetSystemZone[], -- Denoted z.CONS[x, y] or CONS[x, y].
          x: RANGE, y: T]→[T]
LIST: PROC[z: ZONE←SafeStorage.GetSystemZone[], -- Denoted z.LIST g or LIST g.
          g: RANGE X ...]→[T]
```

The TARGET type *R* of a list type *T* is opaque, but it may be thought of as an unpainted record [*first*: RANGE, *rest*: *T*]; thus a list value is a REF to an *R*. The *first* and *rest* procs return the fields of an *R*. LIST is short for LIST OF REF ANY.

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

LIST[x_0, ..., x_n].$rest^i$.*first*=x_i

Note that the *g* argument of LIST may have omitted values, which are filled in if possible by the defaulting coercion for RANGE. Examples:

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

The type ATOM

An ATOM is a REF to an opaque type which is exported from *AtomsPrivate* as
AtomRec: TYPE~RECORD[
printName: *Rope*.ROPE.
propertyList: REF ANY←NIL.
link: ATOM←NIL]
There are no additional items in ATOM's cluster; the useful operations on ATOMs are provided by the *ListsAndAtoms* interface. However, the language does provide ATOM literals for atoms which have Cedar names as their printnames, with the syntax $*n*. Examples:
$red
$VeryLongAtomMadeUpOfSeveralWords

There is a coercion from ATOM to any enumerated type: see § 4.7.1A.

Anomaly for space in ATOM literals: You cannot put white space between the $ and the name in an ATOM literal. In return, the name may be a reserved word.

4.5.1B †Pointer types

Pointer is a subclass of reference (§ 4.5.1). There are two flavors of pointer: short and long. Short pointers occupy one word, and point only within the 64k word main data space where frames are allocated. Long pointers occupy two words and point anywhere.

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

Long pointer types have the following dubious items:
- •PLUS: PROC[*T*, LONG INTEGER]→[*T*] -- Denoted by infix +.
- •MINUS: PROC[*T*, LONG INTEGER]→[*T*] -- Denoted by infix −.
- •DIFF: PROC[*T*, *T*]→[LONG INTEGER] -- Also denoted by infix −.

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

Short pointer types have the same procs without the LONG. They also have the following coercion, called *lengthening*:
LONG: PROC[*p*:*T*]→[LONG POINTER TO TARGET]

Note that VAR types have a VARTOPOINTER proc (denoted by prefix @); this turns a VAR *T* into a LONG POINTER TO *T*.

Anomaly for narrowing to a short pointer: The VARTOPOINTER and BASE primitives turn a variable into a LONG POINTER. If the compiler can determine that the variable is in the main data space, then an application of one of these primitives can be narrowed into a POINTER. This is done statically; if an error is possible it is reported by the compiler, even though the actual narrowing might have been successful.

The subrange in pointerTC[48] is only for a pointer type used as the *range* argument of RELATIVE (§ 4.5.4).

4.5.2 Zone types

The zone class is a subclass of address (§ 4.5) and has the items:

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

Currently there are exactly three zone types:

> ZONE, with NEWTYPE=λ [*U*: TYPE] IN MKREF[*target*~*U*], which implies REF.
>
> UNCOUNTED ZONE, with NEWTYPE=λ [*U*: TYPE] IN MKPOINTER[*target*~*U*, *long*~TRUE], which implies LONG POINTER.
>
> *MDSZone*, with NEWTYPE=λ [*U*: TYPE] IN MKPOINTER[*target*~*U*, *long*~FALSE], which implies POINTER.

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

NEW is explained in § 4.3.1. FREE takes a pointer *p* to a variable *v* containing a reference *r* to a variable *fv*. For a ZONE, the expression denoting *p* must have the form @*v*. In spite of appearances, this is safe; think of *v* as a VAR parameter to FREE, and the @ as indicating in the application that it is modified. For example,

```
{v: REF~NEW[INT]; FREE[@v] }
```

The reference *r* must be supplied by the NEW proc of the same zone; this is checked for a ZONE. FREE sets *v* to NIL. In addition:

> For a ZONE, FREE sets all the REF variables of *fv* to NIL; this helps to break circular structures, but only the collector actually reclaims storage. Hence FREE for a ZONE is safe.
>
> For an uncounted zone, FREE reclaims the storage for *fv* by calling the *Dealloc* proc of the zone (see below); hence FREE is unsafe for an uncounted zone; the safety invariant demands that FREE not be called with a pointer unless the variable will not be used any more. It is best if no other pointers to *fv* exist.

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

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

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

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

```
z↑.procs↑.Alloc[z, T.SIZE]
```

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

```
{temp: LONG POINTER~p↑; p↑←NIL; z↑.procs↑.Dealloc[z, temp] }
```

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

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

•4.5.3 *POINTER TO FRAME types*

This class contains the types of the frames of instances of implementation modules (§ 3.3.5). It is a subclass of address, and has the items:

FRAME: TYPE	-- POINTER TO FRAME[*I*].FRAME = *I*.
FROMIMPLINST: PROC[FRAME]→[*T*]	-- Coercion from FRAME to *T*.
COPYIMPLINST: PROC[*i*: *T*]→[*T*]	-- Returns a copy of the module instance *i*. Denoted NEW *i*.
TOPROGRAM: PROC[*T*]→FRAME.PROGRAMPROC	-- Coercion from *T* to the program proc for the instance.

In addition, *T* has a field proc for each value in the frame.

Note that there are coercions from an imported module instance *II*: *I* to the corresponding POINTER TO FRAME, and from the latter to the program proc for the frame. You can get a POINTER TO FRAME[*I*] value from an imported implementation using the first coercion, or from NEW *PF*, where *PF* is an existing POINTER TO FRAME[*I*] value (an application of COPYIMPLINST).

4.5.4 *RELATIVE types*

Sometimes it is convenient to have addresses which are *relative* to the base of some region. Such pointers can be shorter than ordinary pointers. Also, the entire collection of variables in the region can be moved in storage simply by changing the base; in fact, it can be written out and later read in to a possibly different place, and any relative pointers stored in it will still be valid. Cedar provides some (unsafe) support for this facility, in the form of RELATIVE types. A RELATIVE type has a target type which plays the same role as the target type in an ordinary pointer. The analogy to dereferencing a pointer is applying a base pointer to the relative pointer. The RELATIVE class has no DEREFERENCE or APPLY proc. The only useful thing to do with a RELATIVE value is to apply a suitable BASE POINTER to it (§ 4.4.3).

Relative is a subclass of address (§ 4.5) and has items:

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

A relativeTC takes a pointer or descriptor type as its *range* argument. The TARGET of the RELATIVE type is the TARGET of the *range*. To indicate the desired size of a RELATIVE POINTER value, the type constructor for the *range* pointer type can specify a subrange of CARDINAL. There are coercions between RELATIVE POINTER types which differ only in their subranges; these are just like the coercions between subranges of CARDINAL (§ 4.7.3).

4.6 Record and union types

50 **recordTC** ::= ?access[12] (?MONITORED RECORD fields[43] | ‡ MACHINE DEPENDENT RECORD (mdFields | •fields[43])) — MKRECORD[**fields**] | MKMDRECORD[**mdFields** | **fields**]

51 ‡**mdFields** ::= [((n pos), !.. : ?•access[12] t), ...] --In 50, 52. — MKMDFIELDS[LIST[(LIST[([$n, pos]), ...] , t), ...]]

51.1‡**pos** ::= (e_1 ?(: e_2 .. e_3)) -- In 51, 53. — MKPOSITION[firstWord~e_1, firstBit~e_2, lastBit~e_3]

52 **unionTC** ::= SELECT tag FROM (n, ... => (fields[43] | mdFields[51] | •NULL)), ... ?, ENDCASE — MKUNION[selector~**tag**, variants~LIST[([labels~LIST[$n, ...], value~**fields**]), ...]]

Legal only as last type in a recordTC or unionTC.

53 **tag** ::= (n (‡pos[51.1] |) : ?•access[12] | ★†COMPUTED | ★†OVERLAID) (t | *) — [([$n, (pos | NIL)] | $COMPUTED' | $OVERLAID'), (t | TYPEFROMLABELS)]

In 44, 52. * only in unionTC[52].

Examples

```
Cell: TYPE~RECORD[next: REF Cell, val: ATOM];
Status: TYPE~MACHINE DEPENDENT RECORD [         -- Don't omit the field positions.
  channel (0: 8..10): [0..nChannels),           -- nChannels ≤ 8.
  device (0: 0..3): DeviceNumber,               -- DeviceNumber held in ≤ 4 bits.
  stopCode (0: 11..15): Color, fill (0: 4..7): BOOL,  -- No gaps allowed, but any ordering OK.
  command (1: 0..31): ChannelCommand ];         -- Bit numbers ≥16 OK; fields can cross
                                                --   word boundaries only if word-aligned.

Node: TYPE~MACHINE DEPENDENT RECORD [           -- rands is a union or variant part.
  type (0: 0..15): TypeIndex, rator (1: 0..13): Op54,  -- This is the common part.
  rands (1: 14..79): SELECT n (1: 14..15): * FROM      -- Both union and tag have pos.
    nonary=>[],                                 -- Type of n is {nonary, unary, binary}.
    unary=>[a (1: 16..47): REF Node],           -- Can use same name in several variants.
    binary=>[a (1:16..47), b(1:48..79): REF Node]  -- At least one variant must fill 1: 14..79.
    ENDCASE ];
```

Record types are Cedar's facility for grouping values of different types (since group and binding types cannot be named or written in ordinary declarations). Unions are closely related to records because they must be embedded within records in current Cedar.

4.6.1 Record types

RECORD is a subclass of assignable (§ 4.3.2), or of general (§ 4.3.1) if any component is not assignable. The MKRECORD type constructor takes one argument called *fields*: a declaration or group of TYPEs; in the latter case, it is rebound to a decl with secret names. If *fields*=[n_1: T_1, n_2: T_2, ..., n_k: T_k], MKRECORD produces a type with items:

```
n_i: PROC[T]→T_i                 -- One for each name in the decl.
FIELDS: DECL
CONS: PROC[b: FIELDS]→[T]        -- Apply by T b ; a coercion from the binding.
UNCONS: PROC[T]→[FIELDS]         -- No denotation; a coercion to the binding.
UNWRAP: PROC[T]→[U]              -- If fields=[n: U], i.e. for a single-component record
```

Nameless *fields* are not very useful, since there is no way to name the field procs. The values of the n_i procs are not accessible; they can only be applied with dot notation. Thus if *r* is a record value, $r.n_i$ denotes its *i*th field.

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

A variant record inherits some procs from the sequence or union type it contains (§ 4.4.2B, § 4.6.2).

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

Record types in interfaces are *painted*: each type produced by RECORD[...] (i.e., by MKRECORD or MKMDRECORD) in an interface has a unique mark. Thus two occurrences of a record type constructor in an interface always produce two *different types*. In this respect, recordTCs are like unionTCs and enumerationTCs, and differ from all other type constructors. In a program module, however, record types are not painted (unless they are machine-dependent or union-containing; this is a deficiency, and these should not be painted either in this context). The reason is to ensure that old values will still be useful after module replacement. Since painting is the only way to generate unique marks, it is the only way that an implementation can *guarantee* that its types cannot be forged. In practice, however, the protection afforded by opaque types (§ 4.3.4) is usually adequate.

Representation of records: A record variable is represented by a contiguous block of storage, in which the bits representing each field are contiguous and do not cross a word boundary unless they fill a block of words, but are otherwise arranged at the discretion of the compiler. It is not possible to obtain a REF to a record element; this is because the implementation of both reference counting and REF ANY discrimination requires more information about each VAR than is available for a record field. Unless a field fills one or more words, it is not possible to obtain a pointer to the field either (using @); this is because pointers point to words.

Restriction on record sizes: A record type T must have T.SIZE<2^{12}.

A MACHINE DEPENDENT RECORD type constructor can specify the exact arrangement of the fields in a record, using the syntax of rules 51-53. Examples are given with the rules. Fields must be arranged according to the following constraints.

> A pos[51.1] (w) for a field with type U means that the field occupies words w through $w+U$.SIZE, which are bits $16w$ through $16*(w+U.\text{SIZE})-1$, of the record variable; (w: $f..l$) means that it occupies bits $16w+f$ through $16w+l$ inclusive ($0 \leq f \leq l$ is required; there is no upper bound on l). Like everything else in a type constructor, all of w, f and l must be static.
>
> The pos must be large enough to hold a variable of the field type U: if U.SIZE>1, it must exactly fill U.SIZE words; if U.SIZE=1 and U is represented in less than 16 bits (possible for a discrete, row, or record type), it need only be as large as the representation, but may not cross a word boundary. Union fields are treated specially (§ 4.6.3).
>
> If there is a union field, the subfields of at least one case must exactly fill the pos specified for the union field.
>
> Fields may not overlap, and if they fill at least one word, they together must completely fill an integral number of words. The order of fields is not important, except that any variant part must come last both in the layout and in the constructor.
>
> If any field has a pos, each must have one. A machine dependent record may have no pos. In this case, the fields are arranged consecutively, and the constructor must be such that that the rules about word alignment and boundary crossing are not violated by this arrangement; this may require the presence of dummy fields which fill out unused space.

Note that a pos is really explicit code for the field proc, written in a rather restrictive special language.

4.6.2 Variant record types

There are two classes, unions (§ 4.6.3) and sequences (§ 4.4.2B), whose types are not first-class type values, but can only appear as the type of the last field of a record or union. A record whose last field is one of these types is a *variant record*, and its last field is a *variant field*. The other property shared by a union and a sequence type is that each is a generalization of a number of special cases: there is a single value called the *tag* which identifies the special case.

> For a union, the special cases are unrelated, and the tag is a value from an enumeration.
>
> For a sequence, the special cases are rows of different length, and the tag is a value from the row's domain. .

The tag[53] is treated as a field of the containing variant record. This field is readonly. †For a union it can be changed only by an unsafe assignment to the entire variant part or the entire variant record. There is no way to change the tag field of a sequence. †A tag of COMPUTED or OVERLAID means that there is no tag field; instead, the tag value must be supplied by an expression in a withSelect[34] when it is needed for specialization. Tags of * and OVERLAID are only for unions, and are explained in § 4.6.3.

The cluster of a variant record has the items:
The usual procs for the record fields (including the variant field itself, and the tag), and any items inherited by the record type.
For a union, the types of the bound variants; *T.n*=*T*.SPECIALIZE[$*n*].

TAGTYPE: TYPE	-- The type of the tag.
TAG: TAGTYPE;	-- Another proc for the tag field.
VARIANTTYPE: TYPE	-- The (union or sequence) type of the variant field.
VARIANTPART: PROC[*T*]→[VARIANTTYPE]	-- Another proc for the variant field.
SPECIALIZE: PROC[*x*: TAGTYPE]→[*BT*: TYPE]	-- *BT* is a bound variant of *T*; denoted *T*[*x*] for a sequence-containing variant record type.

Specialization yields a record type called a *bound variant* in which the type of the variant field is one of the special cases of the union or sequence. The bound variant differs by:

GENERAL: TYPE	-- The type of the unbound variant record.

lacking SPECIALIZE,
a readonly tag field,
for a union:
  VARIANTTYPE equal to the corresponding case,
  procs inherited from the corresponding case.

Note that if the special case is itself a union or sequence, the bound variant is still a variant record; otherwise it is an ordinary record. A bound variant of a union-containing variant record is denoted *T.n*, since the bound variant types are in the cluster of the variant record type (•alternate, obsolete notations are *T*[*n*] or *n T*). A bound variant of a sequence-containing variant record is denoted *T*[*e*].

Anomaly for equality of variants: A variant record type has EQUAL only if it does not have a SEQUENCE field, and for any two tag values *a* and *b*, *T.a*.SIZE=*T.b*.SIZE. Even if not all sizes are equal, the bound variants have an EQUAL which takes the variant record as its second argument; hence *bv*= *v* is always correct.

The special properties of the subclasses of variant records are given in the sections on unions (§ 4.6.3) and sequences (§ 4.4.2B).

4.6.3 Union types

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

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

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

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

> A union type is just a value, not constrained to be a REF. These values or their VARs can be embedded in records or arrays without paying for extra storage allocation or an extra level of indirection.
>
> The subtype of a union type can be discriminated somewhat faster than a REF ANY.

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

Like record, union is a subclass of assignable (§ 4.3.2), or of general (§ 4.3.1) if it has an unassignable component. Assignment to a union is unsafe. A union type is defined by a unionTC[52]; it is not a first-class type in Cedar, and can *only* appear as the type of the last field of a variant record (§ 4.6.2) or another union. A union type has items:
  the types of the union cases, named by their tags – thus case *n* of union type *T* is denoted by *T.n*;
  CONS (see below).

The types and tag are inherited by the containing variant record, which is the type a program normally deals with. Note that a union type is always painted (although it shouldn't be painted in an implementation).

A case of the union has items:
  the field procs for its fields;
  GENERAL: TYPE -- The union type of which this is a case.
These are inherited by the containing bound variant record in the obvious way.

The cases of the union are given by the arms of the SELECT. The type of the tag must be an enumeration, and each case is named by one or more literals of the enumeration. Thus *Node* in the example has cases *binary*, *unary* and *nonary*, and the type of the tag could have been written {*binary*, *unary*, *nonary*}. The * which actually appears for the tag type is short for an enumTC[54] which lists all the names preceding the => symbols of the SELECT in turn. If the tag type is given explicitly, any enumeration values which don't appear preceding a => symbol have empty cases.

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

Anomaly for union constructors: A constructor for a union value has the form *a*[...], where *a* is one of the enumeration literals of the tag type, and [...] is an ordinary argBinding[27] for the fields of case *a*. The literal *a* may not be omitted. Thus
  n: *Node*←[*rator*~*plus*, *rands*~*binary*[*a*~NIL, *b*~NIL]]
and also
  n: *Node.binary*←[*rator*~*plus*, *rands*~*binary*[*a*~NIL, *b*~NIL]]

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

The primitive ISTYPE can be used to distinguish the case of a union-containing variant record value *x*, and NARROW can be used to obtain a value *bx* of the bound variant type from *x*; see § 4.3.1. The safeSelect[32] construct is a useful and efficient combination of ISTYPE and NARROW which deals systematically with any number of cases. The withSelect[34] construct is an unsafe version of safeSelect which can be used with any union type, and is the only alternative when the tag is COMPUTED or OVERLAID. See § 3.8 for discussion of these forms.

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

A union *U* has machine-dependent fields if and only if its containing record type *R* is machine-dependent. *U* must be last both in the fields and in the representation. Its pos includes the tag. It need not be word-aligned, though its tag and each field in each case must obey the alignment rules for record fields (§ 4.6.1). If *R*'s representation is <16 bits in size, all cases must be the same size. Otherwise, all cases of *R* must be a multiple of 16 bits in size, and at least one case of *U* must exactly fill the space given by the pos for *U*.

4.7 Ordered types

Ordered types can be compared, and they have subranges. The subclasses of ordered are discrete, numeric, pointer, and subrange. Ordered is a subclass of assignable (§ 4.3.2), and has items:

```
LESS: PROC[T, T]→[BOOL];              -- Apply by infix <. See rules 19, 22.
GREATER: PROC[T, T]→[BOOL];           -- Apply by infix >. See rules 19, 22.
MAX: PROC[T←T.FIRST, ...]→[T];        -- Apply by MAX[x, y, ...].
MIN: PROC[T←T.LAST, ...]→[T];         -- Apply by MIN[x, y, ...].
```

All these procs do just what you expect. MAX and MIN accept more arguments than you have the patience to write. Pointers have these procs only if ORDERED=TRUE.

The class also has items:

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

These are discussed in § 4.7.3

4.7.1 Discrete types

The discrete types are those which have a useful bijection into an interval of the natural numbers: whole numbers and enumerations. These are the types that can be used as domains for row types (§ 4.4.2). The class is a subclass of ordered (§ 4.7), and has items:

```
FIRST: T
LAST: T
PRED: PROC[x: T]→[T]                  -- Predecessor. May cause a bounds fault.
SUCC: PROC[x: T]→[T]                  -- Successor. May cause a bounds fault.
```

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

4.7.1A Enumeration types

```
54 enumTC ::= { n, ... } |                          MKENUMERATION[ LIST[$n, ...]] |
   MACHINE DEPENDENT {( (n | ) (e) | n), ... }      MKMDENUMERATION[LIST[( [($n | NIL), e] | [$n, −1] ), ... ]]
```

Examples

```
Op: TYPE~{plus, minus, times, divide};
Color: TYPE~MACHINE DEPENDENT {                     -- A Color value takes 4 bits; green≡1.
  red(0), green, blue(4), (15)}; c: Color;
```

Enumeration is a subclass of discrete (§ 4.7.1). An enumeration type is isomorphic to a [0..*k*] subrange of the integers, without any of the arithmetic operators. The enumeration type $T=\{n_0, \ldots, n_k\}$ has in its cluster:

```
n₀: T;                                  -- •The value of the first element of T.
                                           Also denoted T[n₀]
...
n_k: T                                  -- •The value of the last element of T.
                                           Also denoted T[n_k]
FROMATOM: PROC[ATOM]→[T]                -- A coercion. The argument must be static.
ORD: [T]→[INT]                          -- T.FIRST.SUCCⁿ.ORD=n.
VAL: [INT]→[T]                          -- Denote by target typing only. VAL[x.ORD]=x
```

The ATOM to enumeration coercion is done only at compile-time; the effect is that you can write $\$n_i$ rather than $T.n_i$ for an enumeration literal anywhere except before a dot (and by desugaring, as the first operand of an operator). Note that when the n_i appear in the type constructor, or as tags in a variant record declaration or constructor, they are not expressions; hence this coercion doesn't apply, and you can't write $\$n_i$ in those contexts.

ORD and VAL convert between *T* and INT.

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

•*Anomaly for enumeration literals*: You can write n_i for $T.n_i$ in an argument or binding where the desired type is *T*. In these contexts, even if n_i is known in the current scope, it denotes $T.n_i$ and not the value it is bound to in the scope. Thus

Color: TYPE~{*red*, *blue*, *green*};
red: *Color*←*Color.blue*;
c: *Color*← *red*

leaves *c*=*Color.red*, not =*Color.blue*. In fact, *red*=*red* is false! It is best not to redeclare enumeration names. Better yet is to always write atoms for enumeration literals, and qualify explicitly with the type in the rare cases where this fails because the literal comes before a dot. Thus *red*=$*red* would be false, and $*red*=*red* would be illegal.

Representation of enumerations: The representation of n_i in an enumeration type is the same as that of the INT *i*. For a subrange of an enumeration, $T.\text{FIRST.SUCC}^i$ is represented by *i*.

The type BOOL or BOOLEAN

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

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

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

IF *test* THEN *ifTrue* ELSE *ifFalse*

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

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

The type CHAR or CHARACTER

This is an enumeration type which could be written {'\000, ..., '\377} if the CHAR literals were names; CHARACTER is a synonym for CHAR. CHAR literals are written:

> As '*c* for any character *c* except \, denoting the *i*th CHAR value, where *i* is the ASCII character code for *c*.
>
> As '*ddd*, where each *d* is an octal digit, denoting the *ddd*Bth CHAR value.
>
> As '*c* for various values of *c*, denoting the CHAR values for various non-printing or otherwise confusing characters (see rule 57).
>
> •As *ddd*C, denoting the same value as '*ddd* (obsolete).

Note that CHAR literals are not names, and you cannot use any of the notations for enumeration literals: CHAR[*cl*] or CHAR.*cl* or $*cl* are not allowed if *cl* is a CHAR literal.

CHAR also has the following dubious items:

•PLUS: PROC[*T*, INTEGER]→[*T*]	-- Denoted by infix +.
•MINUS: PROC[*T*, INTEGER]→[*T*]	-- Denoted by infix −.
•DIFF: PROC[*T*, *T*]→[INTEGER]	-- Also denoted by infix −.

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

4.7.2 Numeric types

Numeric types have arithmetic operations. There are no numeric type constructors, only the primitive types INT=LONG INTEGER, LONG CARDINAL, INTEGER, CARDINAL and REAL. All except REAL are subclasses of whole numbers, corresponding to different finite subsets of the integers, and are discrete as well (§ 4.7.1). The class is a subclass of ordered (§ 4.7), and has items:

PLUS: PROC[*T*, *T*]→[*T*]	-- Denoted by infix "+".
MINUS: PROC[*T*, *T*]→[*T*]	-- Denoted by infix "−".
TIMES: PROC[*T*, *T*]→[*T*]	-- Denoted by infix "*".
DIVIDE: PROC[*T*, *T*]→[*T*]	-- Denoted by infix "/". Truncates toward 0: $-(i/j)=(-i)/j=i/(-j)$, except for REAL, which normally rounds.
ABS: PROC[*T*]→[*T*]	
UMINUS: PROC[*T*]→[*T*]	-- Denoted by prefix "−".

4.7.2A Whole numbers

This class is a subclass of discrete (§ 4.7.1) and of numeric (§ 4.7.2), and has the item:

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

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

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

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

1) There is no overflow checking on the numeric procs, except for DIVIDE and REM, which may raise an ERROR defined in *Inline*.

2) A subrange with $\leq 2^{16}$ values is called *short* (currently all subranges have this property, as do INTEGER and NAT). If all arguments are short, the result of an arithmetic proc is truncated to 16 bits without notice (even if it is static). This means that the result is always IN $[-2^{15}..2^{15})$, and may differ from the correct result by some multiple of 2^{16}. You can force proper INT arithmetic by writing at least one argument as *x*.LONG rather than *x*. Thus the program

```
x, y: [0..10000)←1000;
z: INT←x*y;
w: INT←x.LONG*y
```

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

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

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

Octal, specified by B after the number. A sequence of digits without a B is *never* taken as octal, except in a CHAR literal.

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

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

num_1 R num_2 = num_1 0 R num_3 if $num_3 = num_2 - 1$;

num_1 R 0 = num_1 R

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

Representation of whole numbers: Short values are represented in one word; other INT values require two words. The representation is twos complement, with one more negative than positive value.

Performance of whole numbers: Arithmetic is less efficient on subranges with FIRST≠0 (except for INTEGER, which is efficient). Widening a short value to INT is more efficient if FIRST=0. Multiply and divide are quite slow when the arguments are not short. Short divide is faster when FIRST=0 than for INTEGER.

The interface *Inline* has inline procedures for doing bit manipulation on numbers, for obtaining the quotient and remainder simultaneously, and for doing certain other calculations more efficiently than is possible using the procs described above.

•*Cardinal types*

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

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

4.7.2B The type REAL

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

The interface *CedarReals* has items for handling exceptions that can arise in real arithmetic, for changing the rounding modes, etc.

4.7.3 Subrange types

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

In current Cedar the arguments of MKSUBRANGE must satisfy
$-2^{15} \leq first < 2^{15}$ AND $(last-first) < 2^{16}-1$ AND $last <$ (IF $first < 0$ THEN $2^{15}+first$ ELSE 2^{16})

There is a subrange class for each discrete type, which is a subclass of discrete (§ 4.7.1), with the items:

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

Note that there are coercions both to and from the ground type. The former cannot lose information or raise an exception, but the latter raises *BoundsFault* if its argument is not in the subrange. Subranges have their own FIRST, LAST, and ASSIGN items, as well as the items of general. They also inherit unchanged all the procs of the ground type with names not in the subrange class (including the MKSUBRANGE and EMPTYSUBRANGE type constructors); these procs still take the same arguments, and the coercions make it convenient to apply them to subrange values. There are no special arithmetic or comparison procs for subranges. Note that assigning a value of the ground type to a subrange variable will invoke the FROMGROUND coercion, with its attendant bounds check.

Representation of subranges: If *T* is a subrange type, *T*.FIRST is represented by the INT 0 (except for INTEGER, which has 0 represented by 0), and *T*.LAST by the INT (*T*.LAST − *T*.FIRST + 1). The number of bits required to represent a *T* value is the *n* such that
$2^{n-1} <$ (*T*.LAST − *T*.FIRST + 1) $\leq 2^n$
In current Cedar, a subrange value always fits in one word, because a subrange may not have more than 2^{16} values.

4.8 TYPE types

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

> An interface type parameter can be declared in a DIRECTORY statement, and the resulting interface type can be used to declare an interface parameter in an IMPORTS clause. The argument for this latter parameter is supplied by an implementation which exports the interface type. See § 4.3.5.
>
> An *opaque* or *exported* type can be declared in an interface module. An implementation of the interface provides the actual argument. See § 4.3.4.

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

> an interface type is returned by an interface module;
>
> an exported type is returned by an instance of an implementation.

The other possible uses of a type value are these:

> A type value appears in a declaration, after a colon; e.g., *i*: INT.
>
> A type value appears as a value bound to a type name; e.g., *T*: TYPE~INT.
>
> Some of the values in the cluster of a primitive type can be denoted by *T.n*. In general a proc cannot be denoted this way, though it is often possible to write *x.P*[...] to apply the primitive *P* to *x* and other arguments.
>
> Certain primitives take type arguments: CODE, DESCRIPTOR, FIRST, ISTYPE, LAST, LOOPHOLE, NARROW, NEW, SIZE and a number of type constructors.

The runtime type system (in the interface *AMTypes*) provides complete facilities for manipulating types during execution of the program (but currently not for constructing them). The type values it manipulates have the type *AMTypes.Type*, rather than TYPE. A *AMTypes.Type* can be obtained from a TYPE using the primitive:
  CODE: PROC [*T*: TYPE]→[*AMTypes.Type*].

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

> TYPE applied to a static integer *n* yields an opaque type of size *n*; applied to ANY it yields a fully opaque type (§ 4.3.4).
>
> A record type applied to a group or binding yields a record value; this is called a record constructor (§ 4.6.1). The same thing works for arrays (§ 4.4.2A).
>
> A sequence-containing record type applied to a (not necessarily static) CARDINAL yields a record type containing a sequence of definite length, which can only be used in NEW and SIZE (§ 4.4.2B).
>
> A subrange type (including NAT, INTEGER, or CARDINAL) applied to a value of its ground type yields a subrange value (§ 4.7.3).
>
> •A variant record type applied to a static tag value yields a bound variant type (§ 4.6.2).
>
> •An enumerated type applied to a name which is one of the enumeration literals yields the corresponding enumeration value (§ 4.7.1A).

The last two cases are obsolete notations for expressions which should be written with dot notation.

One other use of TYPE is to denote the type of an interface: TYPE *n* (§ 4.3.5).

4.9 Miscellaneous types

•4.9.1 *Unspecified*

The type UNSPECIFIED both implies and is implied by any type *T* with *T*.SIZE=1. The type LONG UNSPECIFIED is implied by any type *T* with *T*.SIZE=2, and implies any type *T* equal to a type of the form LONG ... or REAL. In a CHECKED block, *T* must not be RC (§ 4.5). These types are assignable (§ 4.3.2), and in addition have a peculiar collection of operations in their clusters; if you need to know about any of these, consult a wizard. The main use of unspecified types is as domains of procs which must accept an assortment of types as arguments. Their use should be avoided if at all possible.

4.9.2 Kernel types

Declarations are explained in § 2.4.5, groups in § 2.3.4, and bindings in § 2.3.5. There is a summary of the relations among these classes in § 2.8. The different kinds of constructor are explained in § 2.2.5. Precise definitions of the types and primitives are in § 2.2.1.

4.10 Concurrency

This section describes the Cedar facilities for concurrent programming, and offers some *very* sketchy guidance on the proper construction of concurrent programs. The paper by Lampson and Redell ("Experience with processes and monitors in Mesa," *Comm. ACM*, Feb. 1980) has more information on this subject.

4.10.1 Processes

FORK creates a new concurrent process *P*, which is returned as the value of the FORK. *P* runs the proc which is the first argument of the FORK. *P* is destroyed when the proc returns. JOIN *P* waits until *P* is destroyed, and returns the results returned by the proc. Thus

    x←JOIN FORK Proc[x, y]

is an inefficient way of doing

    x←Proc[x, y]

Process.Detach[*P*] never waits, and causes the results of *P* to be discarded silently. If you do neither JOIN nor *Detach*, the process stays around uselessly after its proc returns.

A FORKed proc runs just like one which is applied in the usual way, except that an exception which escapes from it is not propagated to the proc doing the FORK, but instead calls the debugger (an applEn[27.1] can be written on a FORK, but it does *not* catch exceptions from the new process). Thus any proc that can be FORKed can also be called normally, but not vice versa, since a proc to be FORKed must handle all exceptions.

4.10.2 Monitors

Monitors are for synchronizing access to shared variables. A monitor is a construct which unifies synchronization, declaration of shared data, and the code which touches the data. A monitor is a module which normally contains all the procs that access a certain set of shared variables. These are of two kinds (declared in the block which contains the proc body), ENTRY procs which can be called only from outside the monitor, and INTERNAL procs, which can be called only from within the monitor. A monitor module can also contain other, *external* procs; these are in the module, but are not considered to be in the monitor. They have no special properties, and should not access any shared data that changes; however, this rule is not enforced.

Only one proc in the monitor is allowed to run at a time, so that such a proc behaves as though only one process could access the data. Associated with a monitor there should be an *invariant*, which is true of the shared data whenever no monitor proc is running. This invariant can be assumed whenever an ENTRY proc is entered, and must be established whenever an ENTRY proc returns, and whenever a proc in the monitor does a WAIT. There should be *no shared variables not protected by a monitor*. Further discussion of how to write concurrent programs that work is beyond the scope of this manual.

There is exactly one MONITORLOCK variable associated with each monitor (*not* necessarily with each MONITOR module instance, though this is so in the simplest case). Note that this is a variable, and is not assignable; usually you use a reference to it. In most cases, however, this variable is not declared explicitly, but instead is declared implicitly with the name LOCK:

> A MONITOR module with no LOCKS clause has an implicit declaration of a variable LOCK: MONITORLOCK.
>
> A MONITORED RECORD has an implicit declaration of a field LOCK: MONITORLOCK.

The locks[4] clause in a MONITOR module determines which monitor all the entry and internal procs of the module belong to (i.e., which MONITORLOCK they lock and unlock). There are three cases, increasingly complicated to handle and providing increasing amounts of flexibility and concurrency. Use the simplest case you can get away with.

1) If there is no locks clause, the procs in one instance of the module all belong to a single monitor associated with the instance. The MONITORLOCK is the LOCK variable of the module instance.

2) If there is a locks clause but it has no USING clause, the *e* of the locks clause is evaluated to obtain the MONITORLOCK. This is done in the scope of the module parameters and any open on the module block. This case is useful when procs in several MONITOR modules must be part of the same monitor. One module declares the lock, and the others import it. Alternatively, it can be allocated elsewhere, and passed to each instance at initialization.

3) If the locks clause has a USING n_u: *T*, every proc P_m in the monitor must have a parameter n_u: *T*. The *e* in the locks clause is made into a proc

   P_u: PROC[n_u: *t*] RETURNS [MONITORLOCK]~{RETURN[*e*]}

   (in the same scope in which *e* is evaluated in (2)), and P_u is applied to the n_u parameter of P_m to yield the lock variable each time it is needed. This case is useful when there are many instances of the shared data, all operated on by the same procs, and each instance has an invariant which is independent of the others.

Restriction on LOCK expressions: The evaluation of the expression that yields a lock must not do a WAIT.

Caution that lock expressions must be functional: In cases (2) and (3), the expression that yields the lock variable is reevaluated *each time* the lock is needed, i.e., at start and end of each ENTRY proc application, and of each WAIT. Within a given application of an ENTRY proc, it must always yield the same variable, or chaos will result; however, this is not enforced.

Caution on global variables with USING: In case (3), the global variables of the MONITOR module instance are *not* protected by the lock. Almost certainly they should be changed only during initialization.

•In cases (2) and (3), the expression that yields the lock variable may yield a MONITORLOCK, a record containing a field LOCK: MONITORLOCK, or a reference value which can be dereferenced to yield one of these. This is a minor convenience to save you from writing ↑.LOCK, and it should be avoided.

An ENTRY proc may be inline, and may be declared in an interface. In this case the interface must have a locks clause, which probably refers to an interface variable or has a USING.

4.10.3 Conditions. WAIT and SIGNAL

Often a monitor proc cannot complete its job, but must wait for the state of its data to change (e.g., in a bounded buffer, the *Put* proc might find the buffer full, and must wait for space to be available). Waiting is done by a WAIT primitive, which specifies a *condition variable* of type CONDITION on which to wait. Note that this is a variable and is not assignable; usually you use a reference to it.

The WAIT releases the monitor lock for the monitor that encloses it, so the waiting process must establish the monitor invariant. Execution will resume after WAIT *c* at some time after one of the following is true:

> There is a BROADCAST done on *c*.
>
> There is a NOTIFY done on *c*, and the waiting process has the highest priority of any process waiting on *c*, and has been waiting on *c* longer than any other process with the same priority.
>
> The process has been waiting longer than the *timeout* interval associated with *c*. There are procs in the *Process* interface for setting timeout intervals. There is no special indication that waiting ended because of a timeout; the program can read the clock, or find this out in some other way.
>
> An ABORTED ERROR is caused in the process by some other process. There is a proc in the *Process* interface to accomplish this. The ABORTED is the result of the WAIT, and never arises from any other primitive.

A process continuing after a WAIT has no special priority, and may not assume anything about the monitor data except the invariant. Thus a WAIT should be inside a loop of the form

UNTIL data is such that the process can proceed DO WAIT *c* ENDLOOP

The idea is that WAIT is *simply* an optimization of busy waiting, in which the process repeatedly tests for the desired state, wasting a lot of processor cycles.

For this to work, when a monitor proc changes the data so that a waiting process might be able to proceed, it should do a BROADCAST to a condition variable which has been declared to reflect this fact. It may do a NOTIFY instead if only one process should proceed, and it is always the process at the head of the condition queue; this is an optimization which may avoid needless execution of several waiting processes (but if misused, it may prevent the right process from running). In a properly written program, BROADCAST is always correct.

There is no way to time out a process waiting to acquire a monitor lock.

Note that an internal proc doing a WAIT in a monitor with a USING clause must have a suitable n_u parameter.

4.10.4 Exceptions

An exception which is the result of an entry proc will not release the lock when the proc is finalized, unless there is an enChoice[9] which catches *only* UNWIND in the enable of the proc's block. Hence every entry proc should have such an enChoice, unless it is known that it never raises an ERROR or a SIGNAL that isn't resumed. Of course, the UNWIND enChoice should establish the invariant. If no work is required to do this, it can simply be NULL.

Anomaly about errors exiting from ENTRY procs: Recall that the current implementation of ERROR handling does not do finalization until there is a GOTO out of the enChoice that catches the error

(§ 3.4.3.1). This means that if the error came out of an entry proc the lock is not released; hence the enChoice should refrain from calling any monitor procs.

If the exception is actually raised in the ENTRY proc itself, an alternative is to raise it using RETURN WITH ERROR instead of ERROR. This causes the lock to be released first. Of course the monitor invariant should be established. In this case the lock is released before the error is propagated, so the enChoice that catches it is free to call the monitor again.

An enChoice on a WAIT, like all the other code in a monitor proc, is executed with the lock held.

4.10.5 Miscellaneous

The monitor data must be initialized before any entry procs are called. It is unwise to rely on a start trap (§ 3.3.2A) for this, since the monitor lock is *not* held during execution of the program proc. An initialization proc should be called (•or the module should be STARTed explicitly) before any processes are allowed to call entry procs of the monitor.

Performance of process primitives: WAIT, NOTIFY, BROADCAST, and entry to and exit from an ENTRY proc are quite efficient: each costs significantly less than an ordinary proc call. A process switch costs about as much as calling a null proc with no arguments or results. A FORK/JOIN pair costs about 30 times as much.

4.11 Defaults

```
55 defaultTC ::=                    CHANGEDEFAULT[oldT~t, (
   t ← |                                 Default~NIL, trashOK~FALSE] |
   t ← e |                               Default~INLINE λ IN e, trashOK~FALSE] |
   ★t ← e ⊥ TRASH |                     Default~INLINE λ IN e, trashOK~TRUE] |
   ★t ← TRASH                            Default~t.Trash, trashOK~TRUE] )
```

defaultTC legal only as the type in a decl in a body[9] or field[43] (n: t ← e), in a TYPE binding[13], or in NEW. Note the terminal |.
•TRASH may be written as NULL.

Examples

```
-- Except as noted, a constructor or application must mention each name and give it a value.
Q: TYPE~RECORD[                 -- Otherwise there's a compile-time error.
  i: INT,                       -- Q[], Q[i~ ] trash i (not in argBinding27).
  j: INT←,                      -- No defaulting or trash for j.
  k: INT←3,                     -- Q[], Q[k~ ] leave k=3.
  l: INT←3 | TRASH,             -- As k, but Q[l~TRASH] trashes l.
  m: INT←TRASH ];               -- Q[], Q[m~ ] trash m.
```

A default in a type cluster provides a value which is supplied automatically in a binding where no value is explicitly given. Example:

PutInt: PROC[*i*: INT, *radix*: [0..100]←10]

makes *PutInt*[*i*~*x*] short for *PutInt*[*i*~*x*, *radix*~10]. This is very convenient for infrequently-used arguments, if arguments are added to a widely-used proc, or to ensure that variables are initialized uniformly.

In summary, the usual cases for defaults and bindings are given in Table 4−6. It says that you can forbid defaulting by writing the defaultTC *T*←, and you can provide a default by writing *T*←*e*. Note that the default expression *e* is evaluated in the scope of the type *T*←*e*, *not* the scope of the binding.

Declaration		*n*: *T*←	*n*: *T*←*e*	*n*: *T* in drType[42]
Binding	short for			
n~*x*	*n*~*x*	*n*~*x*	*n*~*x*	*n*~*x*
n~ or nothing	*n*~OMITTED	ERROR	*n*~*e* (in scope of decl)	ERROR

Table 4–6: Usual cases for defaults

Anomaly on discarding defaults for domain and dange declarations: The last column says that if you just write *T* in a proc domain or range declaration, any default is discarded. This means that you can tell by looking at the declaration whether there will be defaulting, without knowing anything about the defaulting properties of the types.

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

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

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

CHANGEDEFAULT: PROC[*OldT*: TYPE, *Default*: PROC []→[*T*], *trashOK*: BOOL]→[*NewT*: TYPE]

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

> *NewT.Default* is *Default*.
>
> *NewT.Trash* is copied from *OldT.Trash* if *trashOK*=TRUE; a missing *OldT.Trash* causes an error in this case. *NewT.Trash* is omitted if *trashOK*=FALSE.

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

The two special values cannot be written explicitly in a program, but are supplied as follows:

> OMITTED – in an argBinding[27] the syntax *n*~ , which omits the value, means *n*~OMITTED. Then if there is a *Default*, OMITTED is coerced to *T.Default*[] to provide a value of type *T*. There is also a coercion which adds *n*~OMITTED to a binding which lacks *n*, so that *n* can be left out entirely with the same effect as writing *n*~ . You can write a denotation for OMITTED in a VAR constructor, i.e., on the left side of ←.
>
> In a group (constructor without names), an empty element means OMITTED; note that the group is first coerced to a binding by attaching the binding's names to the group elements in order (§ 2.2.6), and then if the resulting binding is too short, n~OMITTED elements are added for the trailing names.
>
> TRASH – a binding can specify this value explicitly with the syntax *n*~TRASH. It is unwise to use TRASH if the program uses the value. Its purpose is to avoid the cost of initializing a variable which is going to be reinitialized before it is read.

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

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

> CONDITION, MONITORLOCK and PORT have no *Trash* or *Default*; they do have an INIT proc which sets any variable to NIL.
>
> REF and PROC types have no *Trash*, and a *Default* which returns NIL.
>
> Bound variant records have no *Trash*, and a *Default* which sets the tag value appropriately.
>
> Composite types have a *Trash* or *Default* if all their component types do; it is the obvious concatenation of the component *Trash* or *Default* procs.

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

<table>
<tr><th colspan="2">Default type constructor</th><th>T←</th><th>T←e</th><th>T←e | TRASH</th><th>T←TRASH</th><th>T in domain/ range decl</th></tr>
<tr><td colspan="2">Default</td><td>—</td><td>λ [] IN e</td><td>λ [] IN e</td><td>T.Trash</td><td>—</td></tr>
<tr><td colspan="2">Trash</td><td>—</td><td>—</td><td>T.Trash</td><td>T.Trash</td><td>—</td></tr>
<tr><th>Declaration
Binding</th><th>short for</th><th>n: T←</th><th>n: T←e</th><th>n: T←e |TRASH</th><th>n:T←TRASH</th><th>n: T</th></tr>
<tr><td>n~x</td><td>n~x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>n~ or nothing</td><td>n~OMITTED</td><td>ERROR</td><td>e (Default[])</td><td>e (Default[])</td><td>T.Trash[]</td><td>ERROR</td></tr>
<tr><td>n~TRASH</td><td>n~TRASH</td><td>ERROR</td><td>ERROR</td><td>T.Trash[]</td><td>T.Trash[]</td><td>ERROR</td></tr>
</table>

Table 4 – 7: Complete cases for defaults

4.12 Type implication

A type T *implies* another type T' ($T \Rightarrow T'$ for short) if for any value x,

$$T.Predicate[x] \Rightarrow T'.Predicate[x]$$

In other words, if any value that has type T (satisfies T's predicate) also has type T', then T implies T'. A consequence is that a proc with domain type T' can safely be given a value of type T, since this value must also have type T', as required by the proc. We also say that a T value is *as good as* a T' value, or that T is a *subtype* of T'.

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

From the implementation's viewpoint (and after all, it is the implementation of an abstraction that is responsible for attaching marks), two values should have the same mark only if they both have representations with all the properties implied by that mark: occupy at least that much space, have the proper fields interpreted in the proper way, etc. This is the rationale for marks: to distinguish values which are not acceptable to the same primitives. Of course this is not an enforceable rule: an implementation can unwisely allow the marks it controls to be applied to unsuitable values.

For example, [0..5]⇒[0..7] because both occupy four bits and represent the integer unbiased. But [1..5] does not imply [0..7], because it happens that the implementation biases the representation of a subrange value, so that the value 1 is represented in [1..5] by binary 0000, but in [0..7] by binary 0001. [1..5] and [0..7] must have different marks, but [0..5] and [0..7] can have the same mark (which might be called "four bit unbiased representation for unsigned integer"), and distinguish the values with the rest of their predicates ($0 \leq x \leq 5$ vs $0 \leq x \leq 7$).

For T to imply T', there must be a proof that T's predicate implies T''s predicate. If T is an arbitrary type, and nothing is known about its relationship to other types, or if it tests for a unique

mark, then no such proof is possible. As a result, only an argument with syntactic type T is acceptable to a $T \rightarrow R$ proc. For built-in types and type-returning procs, however, the compiler knows the predicates and keeps track of the implications. The implies relations among built-in type are (the transitive closure of those) specified in the following table.

Certain points about the table are of special interest:

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

The line for transfer types (including PROC) says that $(D \rightarrow R) \Rightarrow (D' \rightarrow R')$ if $D' \Rightarrow D$ and $R \Rightarrow R'$. The relation is reversed for the domain types, because a $D' \rightarrow R'$ proc P' must accept any D', while a $D \rightarrow R$ proc P only accepts Ds. If P is used in the former context, it is only guaranteed to get a D', and that must imply a D.

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

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

4.13 Coercions

In a binding n: $t \sim e$, the value e must have the type t. To ensure that it does, the binding constructor is type-checked by requiring ∇e to imply t. If it does not, an attempt is made to find a *coercion* function C: $\nabla e \rightarrow t$ which can map the argument to the required type. If C is found, the binding is rewritten as n: $t \sim C[e]$, which typechecks. We say that e is *coerced* to the type D.

A coercion may also be done in an application such as $f[e]$; this is actually a special case of a binding. Note that infix operators, including assignment, are special ways of writing applications, and hence also do coercions. In particular, x: REAL; $x \leftarrow 3$ will coerce 3 to a REAL.

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

Note that if T implies U (see § 4.12), no coercion from T to U is needed to make an application type-check. Another way of thinking about this: $T \Rightarrow U$ means that there is a coercion function from T to U, but it does no computation. This is why REF T can be coerced to REF U if $T \Rightarrow U$.

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

There is currently no way for the program to specify coercion procs. However, there is a modest set of built-in coercions, which are are listed in table 4–9. These can be composed, if the types permit it, to yield a coercion function. None of them loses information, except those from various whole numbers to REAL; in other words, they all have inverses. None of them can raise an exception, except a coercion from a base type to a subrange, which can cause *Runtime.BoundsFault*. Any argument omitted from the type proc applications in the table may take any legal value, but it must take the same value in both applications in a single row.

In current form			In kernel form	
These types	**Imply these types**	**Conditions**	**These types**	**Imply these types**
[*n*: *T*, ...]	[*n*: *T′*, ...]	if $T \Rightarrow T'$	[*n*: *T*, ...]	[*n*: *T′*, ...]
Pointwise extension to bindings. Likewise for groups.				
T	*T* PAINTED *U*		*T*	REPLACEPAINT [*in*~*U*, *from*~*T*]
and vice versa.				
T	ANY	for any *T*		
VAR *T*	READONLY *T*		VAR *T*	READONLY *T*
READONLY *T*	READONLY *T′*	if $T \Rightarrow T'$	READONLY *T*	READONLY *T′*
PROC/ERROR/... [*T*] RETURNS [*U*]	PROC/ERROR/... [T′] RETURNS [*U′*]	if $T' \Rightarrow T$ and $U \Rightarrow U'$	MKXFERTYPE[*domain*~*T*, *range*~*U*]	MKXFERTYPE[*domain*~*T′*, *range*~*U′*]
Note the reversed implication for the domain type.				
SAFE PROC/ERROR/...	UNSAFE PROC/ERROR/...		MKXFERTYPE[*safe*~TRUE]	MKXFERTYPE[*safe*~FALSE]
ARRAY ... OF *T*	ARRAY ... OF *T′*	if $T \Rightarrow T'$	MKARRAY[*range*~*T*]	MKARRAY[*range*~*T′*]
If PACKED=FALSE or *T*.SIZE>1. If PACKED=TRUE and *T*.SIZE=1, the number of bits required to represent a *T* and to represent a *T′* must be equal when rounded up to the next power of 2. Likewise for SEQUENCE and DESCRIPTOR.				
REF *T*	REF READONLY *T*		MKREF[*readOnly*~FALSE]	MKREF[*readOnly*~TRUE]
and likewise for POINTER and LIST.				
REF READONLY *T*	REF READONLY *T′*	if $T \Rightarrow T'$	MKREF[*target*~*T*, *readOnly*~TRUE]	MKREF[*target*~*T′*, *readOnly*~TRUE]
and likewise for POINTER and LIST.				
REF *T*	REF ANY		MKREF[*target*~*T*]	MKREF[*target*~ANY]
ORDERED POINTER TO *T*	POINTER TO *T*		MKPOINTER[*ordered*~TRUE]	MKPOINTER[*ordered*~FALSE]
BASE POINTER	POINTER	and vice versa	MKPOINTER[*base*~TRUE]	MKPOINTER[*base*~FALSE]
T.n	*T*	if *T.n*.SIZE=*T*.SIZE	*T.n*	*T*
A bound variant implies the unbound variant.				
RECORD[*n*: *T*]	*T*	1-element record	MKRECORD[*fields*~[*n*: *T*]]	*T*
and likewise for MACHINE DEPENDENT RECORD.				
(PROC[*A*]→[*n*: *T*]).RANGE	*T*	1-element binding	[*n*: *T*]	*T*
(PROC[*A*]→[*T*]).RANGE	*T*	1-element group	CROSS[[*T*]]	*T*
T[*x*..*y*] etc.	*T*		*T*.MKSUBRANGE[*x*, *y*]	*T*
if *T*.FIRST=*x* and SIZE[*T*[*x*..*y*]]=SIZE[*T*].				
T[*x*..*y*] etc.	*T′*[*x′*..*y′*] etc.		*T*.MKSUBRANGE[*x*, *y*]	*T′*.MKSUBRANGE[*x*, *y*]
if *T*.GROUND=*T′*.GROUND and $x=x'$ and $y \leq y'$.				
T	*T*←*e* etc.		*T*	CHANGEDEFAULT [*T*, ...]
and vice versa; changing defaults doesn't affect the predicate.				

Table 4–8: Implies relations for primitive types

In current form These types can be coerced	to these types	Remarks	*In kernel form* These types can be coerced	to these types
[..., n: T, ...]	[..., n: T', ...]	if T coerces to T'	[..., n: T, ...]	[..., n: T', ...]
This is pointwise extension of coercion to bindings. Likewise for groups.				
$[T_1, ..., T_k]$	$[n_1: T_1, ..., n_k: T_k]$	group to binding	$T_1 \times ... \times T_k$	$[n_1: T_1, ..., n_k: T_k]$
$[n_1: T_1, ..., n_k: T_k]$	$[n_1:T_1, ..., n_k:T_k, n: T]$	if T has a default.	$[n_1: T_1, ..., n_k: T_k]$	$[n_1:T_1, ..., n_k:T_k, n: T]$
T	T'	if $T \Rightarrow T'$	T	T'
$T[x..y]$	T		T.MKSUBRANGE[x, y]	T
T	$T[x..y]$	may raise *Runtime.BoundsFault*	T	T.MKSUBRANGE[x, y]
and the same subrange coercions for relative address types.				
INT/INTEGER/ CARDINAL/ LONG CARDINAL	REAL	loses information	same	
POINTER	LONG POINTER		MKPOINTER[*long*~FALSE]	MKPOINTER[*long*~TRUE]
and likewise for DESCRIPTOR.				
$T.n$	T	bound variant	$T.n$	T
VAR T	T	variable to value	VAR T	T
ATOM	T	T an enumeration; static only.		
NIL	T	if T.NIL exists.		
POINTER TO FRAME [n]	PROGRAM[d] RETURNS[r]	if the *PP* of n has the PROGRAM type.		
OMITTED	T	if *T.Default* exists.		

Table 4 – 9: Coercions for primitive types

4.14 Dot notation

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

If b is a binding, then $b.n$ is the value of n in b; it is an error if b has no element n.

By a natural extension:

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

By a somewhat less natural, but very useful further extension (inspired by classical notation for records, and by Smalltalk):

If e is an expression not a type or binding, then let $P=(\nabla e).n$.

If P.DOMAIN$=[p\text{: } D]$, then $e.n$ is $P[e]$.

Otherwise, if P.DOMAIN$=[p_1\text{: } D_1,\ p_2\text{: } D_2,\ \ldots,\ p_n\text{: } D_n]$, $e.n$ is $\lambda\ [p_2\text{: } D_2,\ \ldots,\ p_n\text{: } D_n]$ IN $P[e,\ p_2,\ \ldots,\ p_n]$

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

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

Object notation is the most general, since any opaque, record or enumeration type D defined in an interface acquires a user-defined cluster by this method. The current implementation is clumsy: *all* the procs in the interface I from which D comes are added to D's cluster, with the names they have in I, except those whose names are already in D's cluster. Of course, an element of this cluster is only useful if it takes a D or reference to D as its first argument. The reference case is often useful because when these procs are inherited by a reference type, they are not modified. E.g., if P: [REF D]→[...] is in D's cluster, it will also be in REF D's cluster, and if r: REF D, then $r.P$ will be correct.

The interface I from which P is obtained is normally an interface *instance* I (which is imported), not an interface *type* IT (declared in the DIRECTORY clause), because only the instance provides a proc value for P. See § 3.3 for more on interfaces.

Restriction on object notation with multiple imported instances: The value for P always comes from the principal imported instance of IT (see § 3.3.3). You can ignore this if only one IT value is imported. If more than one is imported, however, confusion can result. If it does, consult a wizard.

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

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

Case	Source for n	$\nabla e.n$	$e.n$	$e.n[p_2\sim x, ...]$
Meaning		can't write this literally.	$(\nabla e).n[e]$	$(\nabla e).n[e][p_2\sim x, ...]$ or $(\nabla e).n[p_1\sim e, p_2\sim x, ...]$
Object notation (∇e must be record, enumeration, or opaque type).	n: PROC[p_1: D]→[T] declared in same interface I as ∇e. Useless unless ∇e coerces to D or reference to D.	$I.n$	≡$I.n[e]$, since n≡$I.n$	*
	n: PROC[p_1: D, p_2: D_2, ...]→[T] declared in same I as ∇e. Useless unless ∇e coerces to D.	$I.n$	No (can't get the value of the curried proc).	≡$I.n[p_1\sim e, p_2\sim x, ...]$
Record	RECORD [..., n: T, ...]	No (can't get the record selector value).	≡a VAR T for field n of record e.	*
Imported interface	IT: DEFS{...; n: T;...}; DIRECTORY IT: TYPE; IMPORT e: IT;	No (can't get the interface selector value).	≡the value exported as n in the e instance of IT.	*
Interface type	IT: DEFS{...; n: $T\sim v$;...] DIRECTORY e: TYPE IT;	No (it would be TYPE.n).	≡v (need a binding for n, not just n: T).	*

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

Table 4 – 10: Cases for dot notation in current Cedar

Index of points to note

This section lists the headings of the paragraphs throughout the manual calling attention to points that should be specially noted: anomalies, cautions, performance, representation, restrictions, and style notes. It also gives the number of the page on which each note can be found. See § 3.1.3 for an explanation of these categories.

Anomalies

ALL 84
Applying a parameterless proc 58
Arithmetic 102
Arrays with empty domains 84
CHAR MINUS 101
CODE 82
Discarding defaults for domain and range declarations 109
Enables in funnyAppls 61
Enumeration literals 100
Equality of variants 75, 97
Errors exiting from ENTRY procs 107
FORK 81
Garbage collection 88
GOTO and procs 58
GOTO and UNWIND 53
GOTO FINISHED 59
LOOPHOLE on variable types 75
MINUS on pointers 92
Mixed integer and cardinal arithmetic 103
Narrowing to a short pointer 92
NEW 61
Order of evaluating bindings 49
Order of finalization 52
Order of initializing variables 50
Parameter and result names 56
Relative array descriptors 87
RESUME 53
StringBody 86
Separate name space of labels 53
Separators for SELECT 58
SIZE 74
Space in ATOM literals 92
Target typing of DESCRIPTOR 86
Target typing of NARROW and LOOPHOLE: 75
Union constructors 98
Union values 98

Cautions

ANY and UNWIND 52
Dangling references to frames 50
Errors in finalization 52
Exceptions in enable choices 52
Exporting a name to several interfaces 47
Finalization 76
Global variables with USING 106
Initializing monitors 43
Inlines in interfaces 44, 57
Inlines in interfaces 44
Lock expressions must be functional 106
Referencing module variables before initialization 43
Uninitialized interface variables 46
Uninitialized RC variables 50
Use of reserved words and predefined names 38

Performance

Block entry and exit 50
Converting between opaque and concrete types 78
Inlines 57
Interface variables 46
ISTYPE for PROC ANY 75
Proc calls 56
Process primitives 108
Row arguments and results 83
SELECT 63
Static expressions 64
Whole numbers 102

Restrictions

ASSIGN procs 76
Bindings in interfaces 45
Cross types 19
Dot notation 18
EQUAL procs 75
Importing a principal instance into imported interfaces 44
Importing multiple instances 44
Inlines 57
LOCK expressions 106
NEW 77
Object notation with multiple imported instances 114
Record sizes 96
Referring to names introduced in an interface 45
Row sizes 83
Types, declarations, bindings and unions 18
Values of opaque types 78
Variables 19

Representation

Address equality 75
ASSIGN 76
Base pointers 87
Enumerations 100
Records 96
Rows 83
Standard procs 78
Subranges 103
Transfer types 80
Whole numbers 102

Style

Expressions in bindings and initializations 50
Nameless open 51
Rows of variable length 86
SELECT 63
Using precedence 61

Index

abbreviations 34
Abort 81
ABORTED 81, 107
ABS 101
abstract 18
access 48, 55
Acknowledgements 1
address 79, 87, 90, 93
  equality 75
  fault 92
ALL 61, 83, 84
allocation 6
  of variables 49
allocator 88
alternation 35
ambiguous 34, 58, 63, 71
AMTypes 67, 104
AND 12, 61, 100
anomaly 33, 36, 116
ANY 11, 35, 51, 63, 75, 79, 80, 87
  and UNWIND 52
applEn 60, 70
application 3, 5, 9, 12, 16, 50, 61, 79
  of a program 81
  of an error 82
  of a module 42, 45
APPLY 12, 17, 79, 84, 87, 90, 91
argBinding 60, 62
argument 3, 6, 16, 18, 19, 20
arithmetic 101-103
array 20, 83
  with empty domains 84
arrayTC 82
arrayType 86
arrow type 9
as good as 110
ASCII 37, 101
assertion language 67
ASSIGN 75, 76, 85, 89
assignable 74, 75, 79, 83, 87, 95, 98, 99, 105
assignment 3, 58, 61, 64
  of a local proc 64
ATOM 5, 8, 11, 92, 100
  literal 92
AtomsPrivate 92
attribute 36

b 13, 34, 39, 55
BASE 83, 86, 92, 94
BASE POINTER 87, 91, 94
bcd 40, 42
BEGIN 39
benign side-effects 50
bind by name 37
binder 40
BINDING 11, 45
binding 3, 10, 19, 20, 34, 41, 42, 50, 55, 64, 114
  constructor 12
  in interfaces 45, 64
block entry and exit 50
block of storage 19, 76, 83, 87, 96
BNF 34
body 9, 16, 45, 48, 57, 59
bold parentheses 35
BOOL 11, 100
BOOLEAN 100
bound variant 75, 76, 78, 97, 110
BoundsFault 84, 85, 99, 101, 103, 111
braces 39
brackets 14
BROADCAST 91, 107
BTOD 10, 11
BTOV 10
built-in 60, 71, 72
builtInType 67
BUT 13, 15
butChoice 15

C 101
call by name 50, 63
CARDINAL 101, 102
carriage return 38
cases of unions 63, 98
catch 51, 82
caution 36, 116
CDOTG 10
CEDAR 54
Cedar kernel 2
CedarReals 67, 103
CHANGEDEFAULT 76, 109
CHAR 101
  literal 38
  MINUS 101
CHARACTER 101
CHECKED 53, 64
checking 6, 64, 88
choice 62, 63
chooses 37
class 1, 4, 65, 70-73
  hierarchy 65, 66
cleanup 52

closure 9, 16
CLRMFullGram.press 33
CLRMSafeGram 33
CLRMSumm.press 33
cluster 4, 11, 12, 56, 61, 114
Cluster 74
CODE 35, 45, 56, 82, 104
code 15
coercion 3, 4, 6, 16, 18, 20, 71, 75, 76, 79, 83, 86, 91, 94, 96, 100, 101, 103, 111, 113
colon 5
command line 44
comma 39, 58
comment 38
commentary 36
compatibility 2
compile-time 86
compiler 40, 41, 42
  compiler switch 64
component 19, 76, 82
  proccomponent 77
composite 77
  composite type 110
  composite variable 77, 89
computation 16
COMPUTED 63, 85, 97, 98
concepts 2
concrete type 78
concurrency 6, 76, 105
CONDITION 107, 110
  condition variable 6, 107
CONS 61, 64, 83, 91, 95
constant 59
constructor 3, 5, 14, 20, 70
  for a union 98
contain 3, 18
container 18
contents 5, 18
contiguous block of storage 83, 96
CONTINUE 51, 58
CONTROL 41, 43, 47, 81
control variable 59
conveniences 6, 7
converting between opaque and concrete types 78
COPYIMPLINST 47, 78, 81, 94
core language 7, 8
counted storage 76, 88
cross type 10, 19, 80
cross-reference 36
current environment 9
cyclic structure 88

d 8, 13, 34, 39, 55
dangling reference 50, 81
DDOTP 11
DDOTT 11
deallocated 50
decimal 102
decimal point 103
DECL 11
declaration 4, 11, 34, 41, 55, 80
  declaration type 111
DECREASING 59
default 3, 17, 20, 62, 83, 91, 108
  default access 48
  defaults for domain and range declarations 109
Default 75, 109
defaultTC 78, 108
deferred 9
DEFINITIONS 40
delimiter 38
denormalized 103
denote 9
dereference 50, 64, 90, 92
  dereferencing NIL 64
DEREFERENCE 90
descriptor 82, 83, 84, 85, 86, 90, 91
DESCRIPTOR 64
descriptorTC 82
desugaring 1, 7, 33, 35
Detach 105
DF files 40
DIFF 92, 101
different types 96, 100
digit 37
DIRECTORY 41, 43
discrete type 82, 99
discriminating 63
DIVIDE 101
domain 4, 9, 12, 15, 56
  domain declaration 109
  domain type implication 111
DOMAIN 79, 80, 81, 84, 87
dot 5, 44
dot notation 5, 12, 20, 61, 70, 79, 114, 115
drType 39, 80
DTOB 11
dynamically narrowing 97

e 8, 13, 14, 15, 34, 36, 39, 48, 55, 57, 60, 62, 67, 95, 99, 108
editor 40
efficiency 34
elements 10, 19
elementwise 111
empty 39
  empty domain 84
  empty subrange 84, 103

EMPTYSUBRANGE 103
enable 48, 51, 57, 59
  enable choice 51, 52, 56, 58, 61, 62, 82
  enable in funnyAppl 61
enChoice 48, 60
END 39
endChoice 62
ENDCASE 35, 63
endpoints 62
English 7
ENTRY proc 105, 107
enumeration 98, 99, 100, 114
  enumeration literal 100
enumTC 77, 99
ENV 9
environment 5, 9
EQUAL 74, 75, 85
equality of variant records 75, 97
error 82
  exiting ENTRY procs 107
  in finalization 52
ERROR 35, 51, 54, 61, 64
escape 57, 58
essential concepts 3
establish 4
evaluate 5, 16
exception 6, 7, 15, 62, 70, 103, 105, 107
  exception handling 51
  exception value 82
  exceptions in enChoices 52
EXIT 59
exiting from ENTRY procs 107
EXITS 53, 59
exponent 37, 103
export 44, 48, 57, 78, 104
  to several interfaces 47
  exported type 46
  exported variable 19
EXPORTS 48
expression 4, 5, 9, 34, 58, 60
expressions in bindings and initializations 50
extendedChar 37
extension 37
external 105
extra separator 39
extractor 61, 75

FALSE 11
field proc 94
fields 80, 95
file 40, 44
finalization 6, 52, 76
FINISHED 59
FIRST 99, 103
first 91
first-class values 18
FIX 9, 10
flat 10
FOR 59
forbid defaulting 108
FORK 57, 61, 81, 105
formal 7
FRAME 94
frame 41, 47, 49, 50, 81, 86, 92
  frame allocation 88
FREE 93
free variables 52
FROMATOM 100
FROMGROUND 103
FROMIMPLINST 94
fully opaque 77, 78
function 17
functional 17, 50, 68, 77
funny application 71, 73
funnyAppl 60, 61

garbage collector 6, 88
General 74, 76, 79, 85, 91, 95, 97, 98
general type 74
global frame 41, 89
global variables 106
good 88
GOTO 51, 58
  GOTO and procs 58
  GOTO and UNWIND 53
  GOTO FINISHED 59
grammar 34
GREATER 99
GROUND 103
group 3, 10, 19
guarantee 96

hasNIL 76, 79, 80, 87
header 54
HEX 15
hexadecimal 102
HIDE 15
hiding 40
history 2

identifier 5, 34
idiom 2
IEEE floating point 103
if 12, 62
  if expression 61
IFPROC 100
immutable 3
imperative 7, 14
implementation 39, 40, 46, 69, 78, 81, 100
implies 9, 11, 66, 80, 87, 98, 105, 110, 111
  for primitive types 112

import 39, 41, 43, 114, 115
  interface 44, 78
  module instance 94
  multiple instances 44
  principal instance 44
  values 64
IN 35, 62
IN 13
incompatibilities 2
incremental collector 88
index 82
  indexed set of values 82
infix 70
  infixOp 13, 60, 61
informal 3
inherit 65, 76, 84, 90, 92, 96, 97, 98
INIT 49, 64, 74
initialize 4, 16, 43, 46, 85, 108
  initialization 43, 50
  initializing monitors 43
  initializing variables 50
inline 44, 57, 107
  inlines in interfaces 44, 57
INLINE 44, 45, 57, 64
Inline 67, 102
inner scopes 49
instance 40, 41, 44, 50, 78, 114
INT 101
INTEGER 101
interface 39, 40, 41, 44, 45, 50, 57, 69, 77, 100, 107, 115
  interface instance 40, 91
  interface type 40, 41, 74, 78, 114
  interface variable 46
INTERFACETYPE 43
INTERNAL 105
invariant 6, 53, 106
ISLONG 76
ISREADONLY 76
ISTYPE 63, 74, 75, 91, 98
  ISTYPE for PROC ANY 75
item 65
iterator 57, 59

JOIN 61, 81, 105

kernel 1, 2
  kernel definition 7
  kernel expressions 13
  kernel types 105
keyboard 2
keyword argument list 19

λ-expression 9, 56, 80
labels 53
LAST 99, 103
LENGTH 83, 86
lengthening 92
LESS 99
LET 12, 20
library 2
Lisp 52
LIST 61, 91
lists of items 39
ListsAndAtoms 67, 92
listTC 87
literal 5, 8, 13, 37, 64, 85
literal parentheses 35
loader 40, 41, 42
local 3, 56
  local proc 50, 64
  local string 86
LOCALSTRING 86
LOCK 106
  LOCK expression 106
LOCKS 56
locks 39
LONG 35, 92
  LONG CARDINAL 101, 102
  LONG INTEGER 101
  LONG POINTER 92, 93
  LONG STRING 86
  LONG UNSPECIFIED 105
  LOOKUP 10
LOOKUPC 12
loop 14, 57, 58, 59, 64
LOOPHOLE 50, 53, 61, 63, 74, 75
  on variable types 75

MACHINE CODE 56
machine-dependent 34, 77, 99
MACHINE DEPENDENT 85
  MACHINE DEPENDENT RECORD 96
machine instructions 56
main data space 92
map 80, 83, 87
  map type 79
mark 4, 77, 96, 110
matches 4
MAX 99
mdFields 95
MDSZone 93
meaning 7, 68
Mesa 2
  Mesa manual 81
MIN 99
MINUS 71, 92, 101
  MINUS on pointers 92
mixed integer and cardinal arithmetic 103
MKBINDD 10
MKBINDP 10
MKCROSS 10, 19

MKDECL 10, 11
MKEMPTYSUBRANGE 99
MKENUMERATION 68
MKINTTYPE 45
MKPAIR 10
MKRECORD 68
MKSUBRANGE 99, 103
MKUNION 68
MKVAR 74, 76
MKXFERTYPE 80
MOD 101
model 18, 40
module 39, 40, 54
  module body 45
  module replacement 69
  module variables 43
monitor 6, 40, 43, 106
  monitor lock 52
  MONITORED RECORD 106
  MONITORLOCK 106, 110
multiple instances 43, 44
multiple imported instances 114

n 8, 13, 34, 37, 39, 48, 55, 57, 60, 62, 67, 87, 95, 99
n-opaque 78
name 3, 5, 18, 19, 20, 34, 37, 48, 49, 51, 52, 68, 70, 71, 72, 73
  names introduced in an interface 45
  name space of labels 53
nameless open 50
NARROW 61, 74, 75, 78, 91, 98
narrow 64, 79, 80, 83, 103
  to a short pointer 92
NarrowFault 75
NarrowRefFault 75
negative literal 102
NEW 47, 61, 64, 74, 76, 77, 85, 88, 91, 93
NEWEXCEPTIONVALUE 82
NEWFRAME 49
NEWTYPE 93
next 59
NIL 79, 90
NILDECL 11
node boundary 38
non-terminal 34, 35
NOT 35, 100
notation 34, 115
notified 6
NOTIFY 91, 107
NULL 35, 58, 63
num 37
numeric type 101

object 115
  object notation 114
object code 40
obsolescent 68
obsolete 34
octal 102
omit 62
OMITTED 20, 109
opaque 74, 77, 91, 92, 104, 114
  opaque type 46, 78
open 39, 48, 50, 56, 57, 59, 63, 79
operator 2, 70, 71
option 70
optional 34, 35
OR 12, 61, 100
ORD 100
order 9
  of evaluation 17, 61
  of evaluating bindings 49
  order of finalization 52
  of initializing variables 50
ordered type 99
overflow checking 102, 103
OVERLAID 85, 97, 98
overlap 19, 89

p 8, 13
package finalization 88
PACKED 35, 83
  PACKED ARRAY 74, 77
painted 69, 96, 98, 100
  paintedTC 77
pair 10
parameter 3, 16
  parameter name 56
parameterless proc 58
partial ordering 67
pattern 11, 14
PC type 90
performance 36, 98
  performance tuning 68
PLUS 14, 92, 101
pointer 87, 90, 92, 93
  pointer-containing 90
POINTER TO FRAME 44, 46, 47, 57, 94
PointerFault 47
pointerTC 84, 87
PORT type 81, 110
pos 95
positional 17, 19
post-condition 4
postfix 61, 70
pragma 6
pre-condition 4
precedence 61
PRED 99

predeclared types 68
predefined names 38
predicate 4, 11
Predicate 74, 110
prefix 61, 70
prefixOp 60
prime 36
primitive 1, 2, 8, 9, 16, 54, 64, 65
  primitive application 16
  primitive proc 61, 70, 73
  primitive type 67-69
principal instance 44, 114
PrincOps 67, 80
priority 107
PRIVATE 46, 48
proc 3
  PROC 110
  PROC ANY 63, 74, 75
  PROC bindings 56
  proc calls 56
  PROC type 4, 80
  PROC type implication 111
  proc value 56, 57
procedure 3
process 6, 105
  process array 18, 88
  process primitive 108
  PROCESS type 81
  Process 67
PROGRAM 40
  PROGRAM arguments 41
  program instance 41, 42, 47
  program proc 43, 94, 108
  PROGRAM type 81
  program value 81
properties 4
PUBLIC 46, 47, 48
punctuation 37

radix 102
RAISE 15
range 4, 12, 56
  range declarations 109
  RANGE 79, 80, 84, 91
reachable 88
READONLY 18, 46, 76, 84, 90, 97, 114
REAL 64, 101, 103
REC 5, 14
reclaimed 88
record type 20, 50, 95, 96, 114, 115
  record field 97
  record field proc 91
  record size 96
  recordTC 77, 95
recursion 5, 57
REF 91, 93, 110
  REF ANY 63, 74, 91, 97
  REF TEXT 85
reference 91, 92
  reference type 90
reference-containing 88
  RC type 78
  RC variables 50
reference-counting 76
referencing opaque types 78
referring to module variables before initialization 43
referring to names introduced in an interface 45
refTC 84, 87
REJECT 51
related structures 40
RELATIVE 92, 94
  relative array descriptor 87
  relative pointer 87, 94
  relativeTC 87
relOp 60, 62
rely 4
REM 101
REPEAT 53, 59
ReportStartFault 43, 47
representation 36, 116
reserved word 2, 37, 38
rest 91
RESTART 61, 81
restrict 44
restriction 2, 33, 36, 116
result 3, 16, 18
  result name 56
RESUME 53
retained 47, 50
RETRY 51, 58
RETURN 51, 56
  RETURN WITH ERROR 108
rewriting 35
ROPE 75, 85
  ROPE literal 38
  Rope 67
row 82, 83
  row as an argument or result 83
  row size 83
  row type 64
  rows of different sizes 82, 86
rule 69
Runtime 67
runtime type system 104

s 48, 34, 39, 57
safe 6, 87, 93
  SAFE 53
  safe language 33
  safe reference 88
  safe storage 88

SafeStorage	67, 76, 88, 93
safe variable	88
safeChoice	62
safeSelect	62, 63, 75, 91, 98
safety	39, 50, 53
safety invariants	6, 88
SANS-SERIF SMALL CAPS	36
scale factor	102
scope	5, 49, 56, 59, 106, 108
SELECT	35, 62, 63, 98
selector	63
semicolon	39, 58
separator	34, 39
separators for SELECT	58
seqTC	82
sequence	74, 83, 84
sequence-containing record	68
set of values	4
shared variables	105, 106
SHARES	35, 48
short	102
SHORT	76
short pointer	92
side-effects	3, 50
signal	51, 52, 82
SIGNAL	35
simple	2
simpleLoop	14
single component	50, 96
SIZE	74, 85
size restrictions	64
Smalltalk	114
source text	40
space	36, 38
space in ATOM literals	92
specialization	85
SPECIALIZE	97
SR	88
sS	57
standard application	9
standard implementation	78
standard proc	78
START	46, 61
start trap	43, 108
StartFault	43
state	16
statement	3414, 17, , 58
static	6, 36, 45, 46, 56, 64, 68, 96, 102
static error	36, 75, 103
static expression	64
static negative value	102
STOP	61, 81
storage	88
strict	15
string	90
STRING	86
StringBody	86
style	36, 116
sub-expression	9
subclass	4, 66, 80
subrange	37, 60, 62, 64, 82, 87, 92, 94, 99, 101, 103
subtype	110
SUCC	99
summaries	33
superscript	34, 36
SV	88
switch	64
synchronization	6, 76, 105
syntactic type	4, 7, 8
syntax	7, 33
t	8, 13, 34, 36, 39, 55, 57, 62, 67, 76, 77, 82, 87, 95, 108
T[*n*]	68, 104
tab	38
tag	6, 82, 84, 95, 97, 98
TAG	63
TAGTYPE	97
TARGET	86, 90, 94
target type	86, 87
target typing	61, 71, 75
of DESCRIPTOR	86
of NARROW, LOOPHOLE	75
terminal symbol	34, 35, 37
TEXT	85
textual	35
THEN	10
THEND	11
THROUGH	59
tilde	5
time	36
timeout	6, 107
TIMES	101
Tioga	38
TOGROUND	103
token	37
TOPROGRAM	94
TOVOID	58, 75
trace-and-sweep	88
transfer	80, 81, 82
transfer type	54, 79, 80, 111
transferTC	80
TRASH	35, 62
Trash	75, 84, 109
TRUE	11
truncated	102
TRUSTED	6, 35, 53
two-character symbols	37
type	4, 11, 18, 34, 65, 70, 114, 115
TYPE	11
TYPE binding	56
type constructor	65, 68
type expression	64

type implication 110
type option 36, 70
type value 104
TYPE *n* 43, 78
type-checking 4, 11, 43
typeCons 67
typeName 36, 60, 67, 77, 87
TYPE[ANY] 46, 68, 77
TYPE[*n*] 46, 68, 77

"u" switch 64
UMINUS 101
UNBOUND 47
UnboundProc 47, 80
unchecked 6, 88
UNCHECKED 53, 64
UNCONS 49, 95
uncounted zone 90, 93
underflow 103
underlined 35
UNHIDE 15
uninitialized 43
interface variables 46
RC variables 50
union 75, 76, 84, 91, 97
union constructor 98
union value 98
unionTC 95
Unnew 47
unpainted record 91
UNREF 50, 63
unsafe 34, 54, 63, 92, 93, 97, 98
UNSAFE 74, 75, 77, 82, 85, 86
unsafe storage 88
UnsafeStorage 67, 93
unspecified 76, 105
UNTIL 59
UNWIND 52, 53, 107
UNWIND and ANY 52
UNWIND and GOTO 53
UNWRAP 95
upper case 37, 38
user-defined cluster 114
USING 35, 44, 106

VAL 61, 100
VALUE 76, 87
value 353, 9, 16, 18, , 51, 52
VALUEOF 76
values of opaque type 78
VAR 3, 18, 46, 76, 83, 90, 111, 114
VAR ANY 91
variable 3, 18, 49, 75, 79, 87, 90
variable type 75, 76
variant field 97
variant record 63, 74, 75, 76, 84, 96, 97
VARIANTPART 97
VARIANTTYPE 97
varTC 55, 76, 82, 87
VARTOPOINTER 76, 92
VOID 14, 58

WAIT 6, 61, 91, 107
WHILE 59
white space 38, 92
whole number 101, 102
whole number literal 102
widening 103
WITH 35, 56
withChoice 62
withSelect 63, 97, 98
wizard 81, 105, 114
write-only 111

*x*DOT*y* 7
xfer 80

ZONE 6, 75, 76, 93

! 37, 70, 71, 72
!! 70, 71, 72
" 37
37
$ 5, 37, 92
$*n* 11
% 37
& 37
' 37
'*c* 101
'*ddd* 101
' 36, 37
( ) 35, 37
* 35, 37, 85, 97, 101
+ 35, 37, 101
, 35, 37
-- 38
. 35, 37
.. 35, 37
/ 37, 101
: 14, 35, 37
:: 7
; 35, 37
< 37, 99
<= 37
= 35, 37, 75
=> 35, 37, 51, 53, 63
> 37, 99
>= 37
@ 37, 50, 64, 76, 77, 83, 85, 92, 93, 96
[ ] 13, 35, 37
\ 37, 101

↑ 37
← 35, 37, 75, 108
{ } 35, 37, 39
| 8, 13
~ 14, 35, 37
~< 37
~= 37
~> 37
~~ 35, 37, 50
¯ 35, 37, 101
^ 37
★ 34
● 34
† 34
‡ 34

Cedar Safe Language Syntax

§3.3 1 **module** ::= DIRECTORY (n_d ?(: TYPE n_i) ?(USING [n_u, ...])), ... ; (interface | implementation)

2 **interface** ::= n_m, !.. : CEDAR DEFINITIONS ?locks ?imports ~ { ?open[7] (d | b); !.. } .

3 **implementation** ::= n_m : CEDAR (PROGRAM ?drType[42] | MONITOR ?drType[42] ?locks) ?imports ?(EXPORTS n_e, ...) ~ block .

4 **imports**::= IMPORTS ((n_{iv} : |) n_{it}), ...) *--In 2. 3.*

5 **locks** ::= LOCKS e ?(USING n_u: t)

---

§3.4 6 **block** ::= ?(CHECKED | UNCHECKED | TRUSTED) { ?open ?enable ?body ?(EXITS (n, !.. => s); ...) } *--In 3. 13. 14.*

7 **open** ::= OPEN (n ~~ e | e), !.. ; *--In 2. 5.*

8 **enable** ::= ENABLE { enChoice; ... };

9 **enChoice** ::= (e, !.. | ANY) => s *--In 7. 27.1.*

10 **body** ::= (d | b); !.. ; s; ... | s; !.. *--In 5. 17.*

---

§3.5 11 **d***eclaration* ::= n, !.. : ?(PUBLIC | PRIVATE) varTC[40] *--In 2. 10. 43.*

13 **b***inding* ::= n, !.. : ?(PUBLIC | PRIVATE) t ~ (e | t_2 *-- t=TYPE--* | CODE | ?INLINE ?(ENTRY|INTERNAL) block[6]) *--In 2. 10.*

---

§3.6 14 **s***tatement* ::= e_1←e_2 | e | block[6] | escape | loop | NULL

16 **escape** ::= GOTO n | EXIT | CONTINUE | (RETURN|RESUME) ?e

17 **loop** ::= ?iterator ?(WHILE e | UNTIL e) DO ?body[10] ?(REPEAT FINISHED => s) ENDLOOP

18 **iterator** ::= THROUGH e | FOR n : t (?DECREASING IN e | ← e_1 , e_2)

e is a subrange. n is readonly.

---

§3.7 19 **e***xpression* ::= n | literal[57] | (e) | (e | typeName[37]) . **(9)** n | prefixOp e | e_1 infixOp e_2 | e_1 AND **(2)** e_2 | e_1 OR **(1)** e_2 | e ↑ **(9)** | ERROR | [argBinding[27]] | application[26] | builtIn [e, ... ?applEn[27.1]] | funnyAppl e ?([?argBinding[27] ?applEn[27.1]]) | subrange[25] | if[28] | select[29] | safeSelect[32] | s

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 select choices[30 33].*

20 **prefixOp** ::= @ **(8)** | − **(7)** | (~ | NOT) **(3)**

21 **infixOp** ::= * | / | MOD **(6)** | + | − **(5)** | relOp **(4)** | ← **(0)**

22 **relOp** ::= ?NOT (?~ (= | < | >) | <= | >= | # | IN) *--In 21. 30.*

23 **builtIn** ::= -- These are enumerated in Table 4−5.

24 **funnyAppl** ::= FORK | JOIN | WAIT | NOTIFY | BROADCAST | SIGNAL | ERROR | RETURN WITH ERROR

25 **subrange** ::= ?typeName[37] ([| ( ) e_1 .. e_2 ( ] |)) *--In 19. 39.*

26 **application** ::= e [?argBinding ?applEn]

27 **argBinding** ::= (n ~ ?e), !.. | (?e), !.. *--In 19. 26.*

27.1 **applEn** ::= ! enChoice[9]; ... *-- In 19. 26.*

---

§3.8 28 **if** ::= IF e_1 THEN e_2 ?(ELSE e_3)

29 **select** ::= SELECT e FROM choice; ... endChoice

The ";" is "," in an expression, here and in 32.

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

31 **endChoice** ::= ENDCASE ?(=> e_3) *--In 29. 32. 34.*

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

33 **safeChoice** ::= n : t => e_2

---

§3.2 56 **n***ame* ::= letter (letter | digit)... *-- Not a reserved word (Table 3−2).*

57 **literal** ::= num ?(D | B) | digit (digit |A|B|C|D|E|F) ... H | ?num . num ?exponent | num exponent | $ n | ' (extendedChar | ' | ") | " (extendedChar | ') ... "

58 **exponent** ::= E ?(+ | −) num

59 **num** ::= digit !..

60 **extendedChar** ::= space | \ extension | anyCharNot"Or\

61 **extension** ::= $digit_1$ $digit_2$ $digit_3$ | N | R | T | B | F | L | ' | " | \

---

§4 36 **t***ype* ::= typeName | builtInType | typeCons

37 **typeName** ::= n | typeName . n | typeName[e] *--In 19.25.36.40.1*

§4.2 38 **builtInType** ::= INT | REAL | TYPE | ATOM | CONDITION | MONITORLOCK

See Table 4 − 2. TYPE *only in a b or an interface's d.*

39 **typeCons** ::= subrange[25] | paintedTC[40.1] | transferTC[41] | arrayTC[44] | seqTC[45] | refTC[46] | listTC[47] | recordTC[50] | unionTC[52] | enumTC[54] | defaultTC[55]

§4.3 40 **varTC** ::= (| READONLY | VAR) t | ANY

In 11, 46, 47. ANY *only in refTC.* VAR *only in interface decl.*

40.1 **paintedTC** ::= typeName[37] PAINTED t

§4.4 41 **transferTC** ::= ?(SAFE | UNSAFE) xfer ?drType

41.1 **xfer** ::= PROC | PROGRAM | PORT | PROCESS | SIGNAL | ERROR

42 **drType** ::= ?$fields_1$ RETURNS $fields_2$ | $fields_1$ *--In 3. 41.*

43 **fields** ::= [d[11], ...] | [t, ...] | ANY *--In 42. 50. 52.* ANY *only in drType.*

44 **arrayTC** ::= ARRAY t_1 OF t_2

45 **seqTC** ::= SEQUENCE n : t_1 OF t_2 *-- Only as last type in 50 or 52.*

§4.5 46 **refTC** ::= REF ?varTC[40]

47 **listTC** ::= LIST ?(OF varTC[40])

§4.6 50 **recordTC** ::= ?MONITORED RECORD fields[43]

52 **unionTC** ::= SELECT n : (t | *) FROM (n => fields[43]), ... ?, ENDCASE *-- Only as last type in fields of 50 or 52.*

§4.7 54 **enumTC** ::= {n, ...}

§4.11 55 **defaultTC** ::= t ← | t ← e

Only as t in a decl in body[9] or field[43] (n: t ← e), in a TYPE *binding[13] or in* NEW.

Cedar Full Language Syntax

§3.3 1 **module** ::= DIRECTORY (n_d ?(: TYPE ?n_t)
?(USING [n_u, ...])), ... :
(interface | implementation)

2 **interface** ::= n_m, !.. : ?CEDAR DEFINITIONS ?locks ?imports
?•(SHARES n_s, ...) ~ ?•access[12] { ?open[7] (d | b); !.. } .

3 **implementation** ::= n_m : ?CEDAR ?safety
(PROGRAM ?drType[42] | MONITOR ?drType[42] ?locks)
?imports ?(EXPORTS n_e, ...) ?•(SHARES n_s, ...)
~ ?•access[12] block .

3.1 **imports**::= IMPORTS ((n_{iv} : |) n_{it}), ...) *--In 2, 3.*

4 **safety** ::= SAFE | UNSAFE *--In 3, 41.*

5 **locks** ::= LOCKS e ?(USING n_u: t)

---

§3.4 6 **block** ::= ?(CHECKED | UNCHECKED | TRUSTED)
{ ?open ?enable ?body ?(EXITS (n, !.. => s); ...) } *--In 3, 13, 14.*

7 **open** ::= OPEN (n ~~ e | e), !.. ;
In 2, 5, 17. •The ~~ may be written as :.

8 **enable** ::= ENABLE (enChoice | {enChoice; ...}); *--In 5, 17.*

9 **enChoice** ::= (e, !.. | ANY) => s *--In 7, 27.1.*

10 **body** ::= (d | b); !.. ; s; ... | s; !.. *--In 5, 17.*

---

§3.5 11 **d***eclaration* ::= n, !.. : ?access 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 **b***inding* ::= n, !.. : ?access t ~ (
e | t_2 -- *if t =* TYPE-- | CODE |
?INLINE ?(ENTRY | INTERNAL) block[6])
‡† ?TRUSTED MACHINE CODE { (e, ...); ... } |
*In 2, 10. •The ~ may be written as =. •*ENTRY *or* INTERNAL *may be written before t. Block or* MACHINE CODE *only for proc types.*

---

§3.6 14 **s***tatement* ::= e_1 ← e_2 | e | block[6] | escape | loop | NULL
In 6, 9, 10, 17, 19.

16 **escape** ::= GOTO n | GO TO n | EXIT | CONTINUE | •LOOP |
•RETRY | •REJECT | (RETURN | RESUME) ?e | †‡e ← STATE

17 **loop** ::= ?iterator ?(WHILE e | UNTIL e)
DO ?•open[7] ?•enable[8] ?body[10]
?•(REPEAT (n, !.. => s); ...) ENDLOOP

18 **iterator** ::= THROUGH e |
FOR (n : t | ★n) (?DECREASING IN e | ← e_1 , e_2)
e is a subrange. In FOR *n: t ..., n is readonly.*

---

§3.7 19 **e***xpression* ::= n | literal[57] | (e) | (e | typeName[37]) . (9) n |
prefixOp e | e_1 infixOp e_2 | e_1 AND (2) e_2 | e_1 OR (1) e_2 |
e ↑ (9) | •STOP | ERROR | [argBinding[27]] |
application[26] |
builtIn [e, ... ?applEn[27.1]] |
funnyAppl e ?([?argBinding[27] ?applEn[27.1]]) |
subrange[25] | if[28] | select[29] | safeSelect[32] | •withSelect[34] | s
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 select choices[30 33 35].*

20 **prefixOp** ::= @ (8) | − (7) | (~ | NOT) (3)

21 **infixOp** ::= * | / | MOD (6) | + | − (5) | relOp (4) | ← (0)

22 **relOp** ::= ?NOT (?~ (= | < | >) | <= | >= | # | IN) *--In 21, 30.*

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]
([e_1 .. e_2] | [e_1 .. e_2) | (e_1 .. e_2] | (e_1 .. e_2)) *--In 19, 39, 48.*

26 **application** ::= e [?argBinding ?applEn]

27 **argBinding** ::= (n ~ (e | | ★TRASH)), !.. | (e | | ★TRASH), !..
*In 19, 26. •The ~ may be written as :. •*NULL *may be written for* TRASH.

27.1 **applEn** ::= ! enChoice[9]; ... *-- In 19, 26.*

---

§3.8 28 **if** ::= IF e_1 THEN e_2 ?(ELSE e_3)

29 **select** ::= SELECT e FROM choice; ... endChoice
The ":" is "." in an expression, here and in 32 and 34.

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

31 **endChoice** ::= ENDCASE ?(=> e_3) *--In 29, 32, 34.*

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

33 **safeChoice** ::= n : t => e_2

34 **•withSelect** ::= WITH (n_1 ~~ e_1 | • e_1) SELECT ?†e_{11} FROM
withChoice; ... endChoice[31] *--•The ~~ may be written as :.*

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

---

§4 36 **t***ype* ::= typeName | builtInType | typeCons

37 **typeName** ::= n_1 | typeName . n_2 | typeName[e] |
•n_2 typeName *--In 19, 25, 36, 40.1, 49.*

§4.2 38 **builtInType** ::= INT | REAL | TYPE | ATOM | CONDITION |
MONITORLOCK | •LONG CARDINAL | •†?LONG UNSPECIFIED |
•†MDSZone | ★ ?†UNCOUNTED ZONE
See Table 4 − 2. TYPE *only in a b or an interface's d.*

39 **typeCons** ::= subrange[25] | paintedTC[40.1] | transferTC[41] |
arrayTC[44] | seqTC[45] | †descriptorTC[45.1] | refTC[46] | listTC[47] |
†pointerTC[48] | •†relativeTC[49] | recordTC[50] | unionTC[52] |
enumTC[54] | defaultTC[55]

§4.3 40 **varTC** ::= (| READONLY | VAR) t | ANY
In 11, 45 − 48. ANY *only in refTC.* VAR *only in interface decl.*

40.1 **paintedTC** ::= typeName[37] PAINTED t

§4.4 41 **transferTC** ::= ?safety[4] xfer ?drType

41.1 **xfer** ::= PROCEDURE | PROC | PROGRAM | PORT |
PROCESS | SIGNAL | ERROR

42 **drType** ::= ?$fields_1$?(RETURNS $fields_2$) *--In 3, 41.*

43 **fields** ::= [d[11], ...] | [t, ...] | ANY *--In 42, 50, 52.* ANY *only in 42.*

44 **arrayTC** ::= ?★PACKED ARRAY ?t_1 OF t_2

45 **seqTC** ::= ?★PACKED SEQUENCE tag[53] OF t
Legal only as last type in the fields of a recordTC or unionTC.

45.1 **†descriptorTC** ::= ?LONG DESCRIPTOR FOR varTC[40]
varTC must be an array type.

§4.5 46 **refTC** ::= REF ?varTC[40]

47 **listTC** ::= LIST ?(OF varTC[40])

48 **†pointerTC** ::= ?LONG ?ORDERED ?•BASE POINTER
?•subrange[25] ?(TO varTC[40]) | •POINTER TO FRAME [n]
Subrange only in a relativeTC; no typeName[37] on it.

49 **•†relativeTC** ::= typeName[37] RELATIVE t

§4.6 50 **recordTC** ::= ?access[12] (?MONITORED RECORD fields[43] |
‡ MACHINE DEPENDENT RECORD (mdFields | •fields[43]))

51 **‡mdFields** ::= [((n pos), !.. : ?•access[12] t), ...] *-- In 50, 52.*

51.1 **‡pos** ::= (e_1 ?(: e_2 .. e_3)) *-- In 51, 53.*

52 **unionTC** ::= SELECT tag FROM
(n, ... => (fields[43] | mdFields[51] | •NULL)), ... ?, ENDCASE
Legal only as last type in the fields of a recordTC or unionTC.

53 **tag** ::= (n ?‡pos[51.1] : ?•access[12] | ★†(COMPUTED|OVERLAID))
(t | *) *--In 44, 52. * only in unionTC.*

§4.7 54 **enumTC** ::= {n, ...} | MACHINE DEPENDENT {(?n (e) | n), ...}

§4.11 55 **defaultTC** ::= t ← | t ← e | ★t ← e ⊥ TRASH | ★t ← TRASH
defaultTC legal only as the type in a decl in a body[9] or field[43] (n: t ← e), in a TYPE *binding[13], or in* NEW. *Note the terminal |. •*TRASH *may be written* NULL.

---

§3.2 56 **n***ame* ::= letter (letter | digit)... *-- Not a reserved word (Table 3 − 2).*

57 **literal** ::= num ?((<u>D</u>|<u>d</u> | <u>B</u>|<u>b</u>) ?num) *-- INT literal.* |
digit (digit |<u>A</u>|<u>B</u>|<u>C</u>|<u>D</u>|<u>E</u>|<u>F</u>) ... (<u>H</u>|<u>h</u>) ?num *-- Hex INT literal.* |
?num . num ?exponent | num exponent *-- REAL literal.* |
' (extendedChar | ' | ") | • digit !.. (<u>C</u>|<u>c</u>) *-- CHAR literal.* |
" (extendedChar | ') ... " ?•(<u>L</u>|<u>l</u>) | $n *-- ROPE, ATOM literal.*

58 **exponent** ::= (<u>E</u>|<u>e</u>) ?(+ | −) num

59 **num** ::= digit !..

60 **extendedChar** ::= space | \ extension | anyCharNot"Or\

61 **extension** ::= $digit_1$ $digit_2$ $digit_3$ |<u>n</u>|<u>N</u>|<u>r</u>|<u>R</u>|<u>t</u>|<u>T</u>|<u>b</u>|<u>B</u>|<u>f</u>|<u>F</u>|<u>l</u>|<u>L</u>|'|"|\

Syntax	Meaning	Examples	Notes
module ::= DIRECTORY (n_d (: TYPE (n_t |) |) ?(USING [n_u, ...])), ... ; (interface | implementation) interface ::= n_m !.. : ?CEDAR DEFINITIONS IN ?locks (imports |) ?•(SHARES n_s, ...) ~ ? •access[12] { ?open[7] (d | b): !.. } . implementation ::= n_m : ?CEDAR ?safety (PROGRAM ?drType[22] | MONITOR ?drType[22] (| locks)) (imports |) ?(EXPORTS n_e, ...) ?•(SHARES n_s, ...) ~ ?•access[12] block . with the **block**'s body desugared to: [(d | b), ..., n_m: PROGRAM drType~{s; ...}] imports ::= IMPORTS ((n_v: |) n_t), ... --*In 2.3* safety ::= SAFE | UNSAFE --*In 3. 41.* locks ::= LOCKS e ?(USING n_u: t)	λ [(n_d : ((TYPE n_t | TYPE n_d)), ...] IN LET (n_d~RESTRICT[n_d, [\$$n_u$, ...]]), ... IN (interface \| implementation) LET r'~[n_m: INTERFACETYPE[[\$$n_m$, ...]]] IN (imports | λ⇒r') -- SHARES allows access to PRIVATE names in n_s. LET REC n_m~open [?(l'~locks,) (d | b), ...] IN n_m LET r ~[(n_e: n_e) , ... , FRAME: TYPE n_m , n_m: FRAME, CONTROL: PROGRAM] IN (imports | λ⇒r') IN (| LET l'~(LET LOCK ~NEWLOCK IN (λ IN LOCK) | locks) IN) LET b ~NEWPROGINSTANCE[**block**].UNCONS IN [(n_e~BINDDFROM[n_e, b' PLUS n_m~b'.n_m]), ... , FRAME~MKINTTYPE[**block**], n_m~b', CONTROL~b'.n_m] λ [(n_i:n_t), ...] => r' IN LET [((n_i|n_t)~(n_t PLUS n_i.BINDING)), ...] λ ?([n_u : t]) IN e	DIRECTORY Rope: TYPE USING [ROPE, Compare], CIFS: TYPE USING [OpenFile,Error,Open,read], IO: TYPE IOStream, Buffer: TYPE; BufferImpl: MONITOR [f: CIFS.OpenFile] LOCKS Buffer.GetLock[h]↑ USING h: Buffer.Handle IMPORTS Files: CIFS, IO, Rope EXPORTS Buffer ~ { -- module body -- } .	-- There should always be a USING clause -- unless most of the interface is used -- or it is a standard one like *Rope* or *IO* -- or it is exported. -- Implementations can have arguments. -- LOCKS only in MONITOR, to specify -- a non-standard lock. -- Note the absence of semicolons. -- EXPORTS in PROGRAM or MONITOR. -- Note the final dot.

block ::=
?(CHECKED | UNCHECKED | TRUSTED)
{ ?open ?enable ?body
?(EXITS (n, !.. => s); ...) }
--*In 3. 13. 15.*
open ::= OPEN (n ~~ e | e), !.. ;
In 2. 5. 17. •The ~~ may be written as :.
enable::= ENABLE (enChoice |
{enChoice; ...});
In 5. 17.
enChoice ::= (e, !.. | ANY) => s
In 7. 27.1.
body ::= (d | b); !.. ; s; ... | s; !..
In 5. 17.
	<u>open</u> LET n'', ... : EXCEPTION~NEWLABEL[] , ...
	IN ((<u>body</u> <u>enable</u>) BUT { (n'', ... => s); ... })
	-- *But n'' is not visible in s.*
	(LET n~λ$_{open}$ IN e.UNREF | --*The IN before !.. is a separator.*
	LET BINDP[(∇(e.UNREF)).P,
	OPENPROCS[(∇(e.UNREF)).P, λ IN e.UNREF]]) IN !.. IN
	BUT ({ <u>enChoice</u> } |
	{ <u>enChoice</u>; ... })
	
	(e | ANY), ... => { s; REJECT; EXITS
	Retry' =>GOTO Retry''[14]; Cont' =>GOTO Cont''[14] }
	LET NEWFRAME[REC [(d | b), ...]].UNCONS IN { s; ...}
		CHECKED {
		OPEN Buffer, Rope;
		ENABLE Buffer.Overflow =>GOTO HandleOvfl;
		stream: IO.Stream~IO.CreateFileStream["X"];
		x: INT←7;
		{OPEN b~~buffer;
		ENABLE {
		Files.Error--[error, file]-- =>{
		stream.Put[IO.rope[error]]; CONTINUE };
		ANY =>{ x←12; GOTO AfterQuit } };
		y: INT←9; ... };
		x←stream.GetInt; ...
		EXITS
		AfterQuit =>{...};
		HandleOvfl =>{...} };
			-- Unnamed OPEN OK for exported
			-- interface or one with a USING clause.
			-- A single choice needn't be in {}.
			-- Use a binding if a name's value is fixed.
			-- Better to initialize declared names.
			-- A statement may be a nested block.
			-- Multiple enable choices must be in {}.
			-- ERRORs can have parameters.
			-- Choices are separated by semicolons.
			-- ANY must be last. ENABLE ends with :.
			-- Other bindings, decls and statements.
			-- Other statements in the outer block.
			-- Multiple EXIT choices are *not* in {}.
			-- *AfterQuit, HandleOvfl* declared here,
			-- legal only in a GOTO in the block.

declaration ::= n, !.. : ?access[12] varTC[40] *In 2. 10. 43.* VAR, READONLY *only for interface var.* access ::= PUBLIC | PRIVATE *In 2. 3. 11. 13. 50. 51. 53.* *binding* ::= n, !.. : ?access[12] t ~ (e | t$_2$ -- *if t=*TYPE | CODE | ?INLINE (ENTRY | INTERNAL |) block[6] | ‡† ?TRUSTED MACHINE CODE {(e, ...); ...}) *In 2. 10. •The ~ may be written as =.* *Block or* MACHINE CODE *only for proc types.* •ENTRY *and* INTERNAL *can also be before t.*	(n: varTC), ... n, ... ~ LET x' : t ~ (e | t$_2$ -- *Same as e except for conflicting syntax.* | NEWEXCEPTIONCODE[] --*t⇒*SIGNAL *or* ERROR | λ [d': t.DOMAIN] IN LET r'~NEWFRAME[t.RANGE].UNCONS IN (LET r' IN ({t.DOMAIN←d'; (l'.ENTER; |) block; RETURN} (FINALLY l'.EXIT |)) BUT {Return'' =>r'}) | MACHINECODE[(BYTESTOINSTRUCTION[e, ...]), ...]) IN x' -- *e is evaluated only once.*	HistValue: TYPE[ANY]; Histogram: TYPE~REF HistValue; baseHist: READONLY Histogram; AddHists: PROC[x, y: Histogram] RETURNS [Histogram]; LabelValue: PRIVATE TYPE~RECORD[first,last:INT,s:ROPE,x:REAL,f,g:INT,r:REF ANY]; Label: TYPE~REF LabelValue; Next: PROC[l: Label] RETURNS[Label]~ INLINE { RETURN[NARROW[l.r]] }; H: TYPE~Histogram[11]; Size: INT~10; HistValue: PUBLIC TYPE~HV[40.1]; baseHist: PUBLIC H←NEW[HistValue←ALL[17]]; x, y: HistValue←[20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0]; FatalError: ERROR[reason: ROPE]~CODE; Setup: PROC [h: Handle[1], a: INT]~ENTRY {...}; i,j,k: INT←0; p,q: BOOL; lb: Label; main: Handle;	-- Interface: An exported type. -- A type binding. -- An exported variable . -- An exported proc. -- PRIVATE only for secret stuff in an interface. -- An inline proc binding. -- Implementation: Binds a TYPE and INT. -- .PUBLIC for exports. -- An exported variable -- with initialization. -- Binds an error. -- Binds an entry proc.
Statement ::= sS *In 6. 9. 10. 17. 19.* sS ::= e$_1$←e$_2$ | e | block[6] | escape | loop | NULL escape ::= GOTO n | GO TO n | EXIT | CONTINUE | •LOOP | •RETRY | (RETURN | RESUME) ?e | •REJECT | †‡e ← STATE loop ::= (iterator |) (WHILE e | UNTIL e |) DO ?•open[7] ?•enable[8] ?body[10] ?(REPEAT (n, !.. =>s); ...) ENDLOOP iterator ::= THROUGH e | FOR (n : t | ★n) ((| DECREASING) IN e | ← e$_1$, e$_2$) *e is a subrange. In* FOR *n: t ... n is readonly except for the assignment in the iterator's desugaring.*	{ SIMPLELOOP {sS; GOTO Cont''; EXITS Retry'' =>NULL}; EXITS Cont'' =>NULL } [e$_1$←e$_2$].TOVOID | e --*must yield* VOID-- | --*all four yield* VOID-- HEX[exception[code~ n'', args~NIL]] | GOTO (Exit''[17] | Cont''[9] | Loop''[17] | Retry''[9]) | { ?(r''[13]←e;) GOTO (Return''[13] | Resume') } | THISEXCEPTION[] | DUMPSTATE[e] {(<u>iterator</u> ; | done'~FALSE; Next'; PROC~{};) { Test'~λ IN (NOT e | e | FALSE); { <u>open</u> SIMPLELOOP { IF Test'[] OR done' THEN GOTO FINISHED; { <u>enable</u> <u>body</u> EXITS Loop' =>NULL }; Next'[] } EXITS Exit'⇒NULL; (n, !..⇒s); ...; FINISHED⇒NULL}}} <u>FOR x': e IN e</u> | (n: t; |) (Range': TYPE~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); | done': BOOL~FALSE; Next': PROC~{n←e$_2$}; n←e$_1$) ;	x←AddHists[baseHist, baseHist]↑; Setup[bh~main, a~3]; {ENABLE FatalError =>RETURN[0]; []←f[3]; ...}; IF i>3 THEN RETURN[25] ELSE GOTO NotPresent; FOR t:INT DECREASING IN [0..5) UNTIL f[t]>3 DO u: INT←0; ... ; u←t+4; ... REPEAT Out =>{...}; FINISHED =>{...} ENDLOOP; THROUGH [1..4] DO i←i*i ENDLOOP; FOR i: INT←1, i+2 WHILE i<8 DO j←j+i ...; FOR l: Label←lb, l.Next WHILE l # NIL DO ...;	-- A statement can be an assignment, -- or an application without results, -- or a block, -- or an IF or an escape statement. -- or a loop. Try to declare *t* in the FOR -- as shown. Avoid OPEN or ENABLE -- after DO (use a block). FINISHED -- must be last. -- Raises *i* to the 16th power. -- Accumulates odd numbers in [1..8). -- Sequences through a list of *Labels*.
expression ::= n | literal[57] | (e) | application[26] | (e | typeName[17]) . (9) n | prefixOp e | e$_1$ infixOp e$_2$ | e$_1$ relOp (4) e$_2$ | e$_1$ AND (2) e$_2$ | e$_1$ OR (1) e$_2$ | e ↑ (9) | •STOP | ERROR | builtIn [e, ?(, e$_2$, !..) ?applEn[27]] | funnyAppl e ?([?argBinding[27] ?applEn[27]]) | [argBinding[27]] | subrange[25] | if[18] | select[24] | safeSelect[12] | •withSelect[14] | s *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[?] and select choices[40,13,15].* prefixOp ::= @ (8) | − (7) | (~ | NOT) (3) infixOp ::= * | / | MOD (6) | + | − (5) | ← (0) relOp ::= ?NOT (?~ (= | < | >) | # | (<= | >=) | IN) --*In 19. 30.* builtIn ::= -- These are enumerated in Table 4 – 5. funnyAppl ::= FORK | JOIN | WAIT | NOTIFY | BROADCAST | SIGNAL | ERROR | RETURN WITH ERROR | •NEW | •START | •RESTART | ‡†TRANSFER WITH | ‡†RETURN WITH subrange ::= (typeName[17] |) ([| ( ) e$_1$ .. e$_2$ ( ] |)) --*In 19. 39. 48.*	 e . <u>prefixOp</u> | e$_1$. <u>infixOp</u>[e$_2$] | (λ [x': ∇e$_1$, y' :∇e$_2$] =>[BOOL] IN <u>relop</u>) [e$_1$, e$_2$] | IF e$_1$ THEN e$_2$ ELSE FALSE | IF e$_1$ THEN TRUE ELSE e$_2$ | e . DEREFERENCE | STOP[] | ERROR NAMELESSERROR | e . <u>builtIn</u> ?([e$_2$, ...] ?applEn) | e . <u>funnyAppl</u> ?([?argBinding ?applEn]) | --Binding must coerce to a record, array, or •local string-- | VARTOPOINTER | UMINUS | NOT TIMES | DIVIDE | REM | PLUS | MINUS | ASSIGN ?NOT (?NOT x'.(EQUAL | LESS | GREATER)[y'] | x'~= y' | x' = y' OR x' (< | >) y' | x' >= y'.FIRST AND (x' <= y'.LAST BUT {BoundsFault =>FALSE})) LET t'~(typeName | INT) , first'~(e$_1$ | e$_1$.SUCC) IN t'.MKSUBRANGE[first', (e$_2$ | e$_2$.PRED)] BUT {BoundsFault =>t'.MKEMPTYSUBRANGE[e$_1$]}	lv: LabelValue[11]←[i, 3, "Hello", 31.4E—1, (i+1), g[x]+lb.f+j.PRED, NIL]; p1: PROCESS RETURNS [INT]←FORK f[i, j]; ERROR NoSpace; WAIT bufferFilled; RT: RTBasic.Type←CODE[LabelValue[11]]; h[– 3, NOT(i>j), i*j, i+3, i NOT >j, p OR q, lb.r↑]; lv[19]←[first~0,last~5,x~3.14,g~2,f~5,r~NIL,s~"1"]; [first~i, last~j]←lv[19]; b: BOOL←i IN [1..10]; FOR x: INT IN (0..11) DO ...; b←(c IN Color[5](red..green] OR x IN INT[0..10));	-- A constructor with some sample -- expressions. -- FunnyAppls take one unbracketted -- arg; many return no result, so -- must be statements. -- An application with sample expressions. -- Short for *lv←LabelValue*[11][...]. -- Assignment to VAR binding -- (extractor). -- Subrange only in types or with IN. -- The INT is redundant.

Syntax	Meaning	Examples	Notes

```
26 application ::= e [?argBinding ?applEn]
27 argBinding ::= (n ~ (e | | ★TRASH)), !.. |
        (e | | ★TRASH), ...
   In 19, 26. •TRASH may be written as NULL, ~ as :.
27.1 applEn ::= ! enChoice⁹; ...-- In 19, 26.
```

```
LET m'~e, a'~[argBinding] IN ( (m', APPLY ▸a' ) ?applEn )
(n ~ (e | OMITTED | TRASH) ), !.. |
(e | OMITTED | TRASH ), ...
BUT { enChoice: ... }
```

```
fh←Files.Open[name~lb.s, mode~Files.read
  ! AccessDenied=>{...}; FatalError=>{...}];
(GetProcs[j].ReadProc)[k];
file.Read[buffer~b, count~k];
f[i~3, j~ , k~TRASH]; f[i~3, k~TRASH];
f[3, , TRASH];
```

```
-- Keywords are best for multiple args.
-- Semicolons separate choices.
-- The proc can be computed.
-- ≡File.Read[file, b, k] (object notation).
-- j and k may be trash (see defaultTC⁵⁵).
-- Likewise, if i, j, and k are in that order.
```

```
28 if ::= IF e₁ THEN e₂ (ELSE e₃ | )
29 select ::= SELECT e FROM
      choice; ... endChoice
   The ":" is "," in an expression; also in 32 and 34.
30 choice ::= ( ( | relOp²² ) e₁ ), !.. => e₂
31 endChoice ::= ENDCASE ( => e₃ | )
   In 29, 32, 34.
32 safeSelect ::= WITH e SELECT FROM
      safeChoice; ... endChoice³¹
33 safeChoice ::= n : t => e₂
34 •withSelect ::= WITH (n₁ ~~ e₁ | • e₁ )
   SELECT ( | †e₁₁) FROM
      withChoice; ... endChoice³¹
   •The ~~ may be written as :.
35 •withChoice ::= n₂ => e₂ |
   n₂, n₂, !.. => e₂
```

```
IF e₁ THEN e₂ ELSE (e₃ | NULL)
LET selector'~e IN
    choice ELSE ... endChoice
-- ELSE is a separator for repetitions of the choice.
IF ( (selector' ( = | relOp ) e₁) OR ... ) THEN e₂
ELSE (e₃ | NULL)
LET v'~e IN
    safeChoice ELSE ... endChoice
IF ISTYPE[v', t] THEN LET n : t←NARROW[v', t] IN e₂
OPEN v'~~e₁ IN LET n'~($n₁ | NIL), type'~∇v',
  selector'~(e₁.TAG | e₁₁) IN withChoice ELSE ... endChoice
-- e₁₁ must be defaulted except for a COMPUTED variant.
IF selector' = $n₂ THEN OPEN
  (BINDP[n', LOOPHOLE[v',type'.n₂]] | BINDP[n', v'] ) IN e₂
```

```
i←(IF j<3 THEN 6 ELSE 8);
IF k NOT IN Range THEN RETURN[7];
SELECT f[j] FROM
  <7=>{...};
  IN [7..8]=>{...};
  NOT <=8=>{...};
  ENDCASE=>ERROR;
WITH r SELECT FROM
  rInt: REF INT=>RETURN[Gcd[rInt↑, 17]];
  rReal: REF REAL=>RETURN[Floor[Sin[rReal↑]]];
  ENDCASE=>RETURN[IF r=NIL THEN 0 ELSE 1]
nr: REF Node⁵²~...; WITH dn~~nr SELECT FROM
  binary=>{nr←dn.b};
  unary=>{nr←dn.a};
  ENDCASE=>{nr←NIL};
```

```
-- An IF with results must have an ELSE.
-- SELECT expressions are also possible.
-- ≡t:INT~f[j]; IF t<7 THEN {...} ELSE ...
-- 7, 8=> or =7, =8=>{...} is the same.
-- ENDCASE=>{...} is the same here.
-- Redundant: choices are exhaustive.
-- Assume r: REF ANY in this example.
-- rInt is declared in this choice only.
-- Only the REF ANY r is known here.
-- See rule 52 for the variant record Node.
-- dn is a Node.binary in this choice only.
-- dn is a Node.unary in this choice only.
-- dn is just a Node here.
```

```
36 type ::= typeName | builtInType | typeCons
37 typeName ::= n₁ | typeName . n₂ |
   typeName [e] | •n₂ typeName
   In 19, 25, 36, 40.1, 49.
38 builtInType ::= INT | REAL | TYPE | ATOM | MONITORLOCK | CONDITION |
   ★ ?†UNCOUNTED ZONE | •†MDSZone | •LONG CARDINAL | •† ?LONG UNSPECIFIED  -- See Table 4-2.
   TYPE only as t in a b or an interface's d. INTEGER, CARDINAL, NAT, TEXT, STRING, BOOL, CHAR are predefined.
39 typeCons ::= subrange²⁵ | paintedTC⁴⁰·¹ | transferTC⁴¹ | arrayTC⁴⁴ | seqTC⁴⁵ | †descriptorTC⁴⁵·¹ |
   refTC⁴⁶ | listTC⁴⁷ | †pointerTC⁴⁸ | •†relativeTC⁴⁹ | recordTC⁵⁰ | unionTC⁵² | enumTC⁵⁴ | defaultTC⁵⁵
40 varTC ::= ( | READONLY | VAR ) t | ANY        ( VAR | READONLY | VAR ) t | ANY
   In 11, 45–48. ANY only in refTC. VAR only in interface decl.
40.1 paintedTC ::= typeName PAINTED t        REPLACEPAINT[in: t, from: typeName]
   typeName must be an opaque type, t recordTC or enumTC.
41 transferTC ::= ?safety⁴ xfer ?drType      MKXFERTYPE[drType, flavor~xfer]
41.1 xfer ::= PROCEDURE | PROC | PROGRAM |
   PORT | PROCESS | SIGNAL | ERROR
42 drType ::= ?fields₁ RETURNS fields₂ | fields₁     domain~fields₁, range~fields₂
   No domain for PROCESS. In 3, 41.
43 fields ::= [d¹¹, ...] | [t, ...] | ANY
   ANY only in drType. In 42, 50, 52.
44 arrayTC ::= ?★PACKED ARRAY ?t₁ OF t₂      MKARRAY[domain~t₁, range~t₂]
45 seqTC ::= ?★PACKED SEQUENCE tag⁵³ OF t     MKSEQUENCE[domain~tag, range~t]
   Legal only as last type in a recordTC or unionTC.
45.1 †descriptorTC ::=
   ?LONG DESCRIPTOR FOR varTC⁴⁰               MKARRAYDESCR[arrayType~varTC]
   varTC must be an array type.
46 refTC ::= REF ( varTC⁴⁰ | )                 MKREF[target~( varTC | ANY )]
47 listTC ::= LIST ( OF varTC⁴⁰ | )            MKLIST[range~( varTC | REF ANY )]
48 †pointerTC::= ?LONG ?ORDERED ?•BASE         MKPOINTER[target~( varTC | UNSPECIFIED ),
      POINTER ?•subrange²⁵ (TO varTC⁴⁰ | ) |              subrange~subrange ] |
   •POINTER TO FRAME [ n ]                     n
   Subrange only in a relativeTC; no typeName¹⁷ on it.
49 •†relativeTC ::= typeName¹⁷ RELATIVE t      MKRELATIVE[range~t, baseType~typeName]
   t must be a pointer or descriptor type. typeName a base pointer type.
50 recordTC ::= ?access¹² (
   ?MONITORED RECORD fields⁴³ |                MKRECORD[ fields] |
   ‡ MACHINE DEPENDENT RECORD
      (mdFields | •fields⁴³) )                 MKMDRECORD[mdFields | fields]
51 ‡mdFields ::= [( (n pos), !.. : --In 50, 52.   MKMDFIELDS[LIST[ ( LIST[ ([$n, pos] ), ... ], t ), ... ] ]
      ?•access¹² t), ...]
51.1 ‡pos ::= ( e₁ ?(: e₂ .. e₃) )  -- In 51, 53.   MKPOSITION[firstWord~e₁, firstBit~e₂, lastBit~e₃]
52 unionTC ::= SELECT tag FROM                 MKUNION[selector~tag, variants~LIST[
   (n, ... => (fields⁴³ | mdFields⁵¹ | •NULL) ), ...       ( [ labels~LIST[ $n, ...], value~fields ] ), ...] ]
   ?, ENDCASE
   Legal only as last type in a recordTC or unionTC.
53 tag ::= (n (‡pos⁵¹·¹ | ) : ?•access¹² |       [ ( [ $n, (pos | NIL) ] | $COMPUTED' | $OVERLAID' ) ,
   ★†COMPUTED | ★†OVERLAID ) (t | *)            ( t | TYPEFROMLABELS ) ]
   In 44, 52. * only in unionTC⁵².
54 enumTC ::= { n, ... } |                     MKENUMERATION[ LIST[$n, ...] ] |
   MACHINE DEPENDENT {( (n | ) (e) | n), ... }  MKMDENUMERATION[LIST[( [($n | NIL), e] | [$n, -1] ), ... ]]
55 defaultTC ::=                               CHANGEDEFAULT[oldT~t, (
   t ← |                                         Default~NIL, trashOK~FALSE] |
   t ← e |                                       Default~INLINE λ IN e, trashOK~FALSE] |
   ★t ← e | TRASH |                              Default~INLINE λ IN e, trashOK~TRUE] |
   ★t ← TRASH                                    Default~t.Trash, trashOK~TRUE] )
   defaultTC legal only as the type in a decl in a body⁹ or field¹¹ (n: t ← e), in a TYPE binding¹¹, or in NEW. Note the terminal |.
   •TRASH may be written as NULL.
```

```
P: PROC[ b: Buffer¹.Handle,
        i: INT←TEXT[20].SIZE ];
TypeIndex: TYPE~[0..256);
BinaryNode: TYPE~Node⁵².binary;
HV: TYPE~Interface.HistValue PAINTED
  RECORD[...]
Enumerate: PROC[
  l: RL,
  p: PROC[x: REF ANY] RETURNS [stop: BOOL]]
RETURNS [stopped: BOOL];
p2:PROCESS RETURNS[i:INT]←FORK stream.Get;
failed: ERROR [reason: ROPE]~CODE;
Vec: TYPE~ARRAY [0..maxVecLen) OF INT;
Chars: TYPE~RECORD [text: PACKED SEQUENCE
  len: [0..INTEGER.LAST] OF CHAR]; ch: Chars;
v: Vec~ALL[0];
dV: DESCRIPTOR FOR ARRAY OF INT~
  DESCRIPTOR[v];
ROText: TYPE~REF READONLY TEXT;
RL: TYPE~LIST OF REF READONLY ANY; rl:RL;
UnsafeHandle: TYPE~LONG POINTER TO Vec⁴⁴;
Cell: TYPE~RECORD[next: REF Cell, val: ATOM];
Status: TYPE~MACHINE DEPENDENT RECORD [
  channel (0: 8..10): [0..nChannels),
  device (0: 0..3): DeviceNumber,
  stopCode (0: 11..15): Color, fill (0: 4..7): BOOL,
  command (1: 0..31): ChannelCommand ];
Node: TYPE~MACHINE DEPENDENT RECORD [
  type (0: 0..15): TypeIndex, rator (1: 0..13): Op⁵⁴,
  rands (1: 14..79): SELECT n (1: 14..15): * FROM
    nonary=>[],
    unary=>[a (1: 16..47): REF Node],
    binary=>[a (1:16..47), b(1:48..79): REF Node]
    ENDCASE ];
Op: TYPE~{plus, minus, times, divide};
Color: TYPE~MACHINE DEPENDENT {
  red(0), green, blue(4), (15)}; c: Color;
Q: TYPE~RECORD[
  i: INT,
  j: INT←,
  k: INT←3,
  l: INT←3 | TRASH,
  m: INT←TRASH ];
```

```
-- A type from an interface.
-- A bound sequence; only in SIZE, NEW.
--n₂ names a variant.
-- A subrange type.
-- A bound variant type.
-- See 13 for use.
-- A record with just a sequence in it.
-- ch.text[i] or ch[i] refers to an element.
-- NARROW[rl.first, ROText]↑ is a
--   READONLY TEXT (or error).
-- Don't omit the field positions.
-- nChannels ≤ 8.
-- DeviceNumber held in ≤ 4 bits.
-- No gaps allowed, but any ordering OK.
-- Bit numbers ≥16 OK; fields can cross
--   word boundaries only if word-aligned.
-- rands is a union or variant part.
-- This is the common part.
-- Both union and tag have pos.
-- Type of n is {nonary, unary, binary}.
-- Can use same name in several variants.
-- At least one variant must fill 1: 14..79.
-- A Color value takes 4 bits; green≡1.
-- Except as noted, a constructor or application must mention each name and give it a value.
-- Otherwise there's a compile-time error.
-- Q[], Q[i~ ] trash i (not in argBinding²⁷).
-- No defaulting or trash for j.
-- Q[], Q[k~ ] leave k=3.
-- As k, but Q[l~TRASH] trashes l.
-- Q[], Q[m~ ] trash m.
```

```
56 name ::= letter (letter | digit)...
57 literal ::= num ?( (D|d [B|b) ?num ) |
   digit (digit |A|B|C|D|E|F) ... ( H|h ) ?num |
   ?num . num ?exponent |
   num exponent |
   ' (extendedChar | ' | ") | • digit !.. (C|c) |
   " (extendedChar | ') ... " ?•(L|l) |
   $ n
58 exponent ::= (E|e) ?(+ | -) num
59 num ::= digit !..
60 extendedChar ::= space | \ extension | anyCharNot"Or\
61 extension ::= digit₁ digit₂ digit₃ |
       (n|N | r|R) | (t|T) | (b|B) |
       (f|F) | (l|L) | ' | " | \
```

```
-- But not one of the reserved words in Table 3-2.
-- INT literal, decimal if radix omitted or D, octal if B. |
-- INT literal in hex; must start with digit. |
-- REAL as a scaled decimal fraction; note no trailing dot. |
-- With an exponent, the decimal point may be omitted. |
-- CHAR literal; the C form specifies the code in octal. |
[ ('extendedChar| '), ...] -- Rope.ROPE, TEXT, or STRING. |
-- ATOM literal.
-- Optionally signed decimal exponent.
-- The character with code digit₁ digit₂ digit₃ B. |
-- CR, '\015 | TAB, '\011 | BACKSPACE, '\010 |
-- FORMFEED, '\014 | LINEFEED, '\012 | ' | " | \
```

```
m, x1, x59y, longNameWithSeveralWords: INT;
n: INT~1+12D+2B3+2000B
     +1H+0FFH;
r1: REAL~0.1+.1+1.0E-1
     +1E-1;
a1: ARRAY [0..3] OF CHAR~['x, '\N, '\', '\141];
r2: ROPE~"Hello.\N...\NGoodbye\F";
a2: ATOM~$NameInAnAtomLiteral;
```

```
-- = 1+12+1024+1024
-- +1+255
-- = 0.1+0.1+0.1
-- +0.1
```