OverviewDoc.tioga
Last Edited by Horning on June 1, 1983 6:48 pm
Last Edited by Donahue on May 1, 1984 3:28:36 pm PDT
Last Edited by: Subhana, May 29, 1984 12:02:07 pm PDT
Last Edited by: John Larson, June 20, 1986 3:05:28 pm PDT

CEDAR LANGUAGE OVERVIEW

Cedar Language Overview

Version 5.2

Release as [Indigo]<Cedar5.2>Documentation>OverviewDoc.tioga, .press
Came from [Indigo]<CedarDocs>Manual>Overview.tioga, .press

Abstract: This Overview is intended to introduce you to the basic vocabulary and concepts that you need before plunging into sources of more detailed information about the Cedar Language. It assumes that you have already read the Briefing Blurb and the Introduction to Cedar. If you haven't, read them first and return. It starts with a brief review of the common concepts that Cedar shares with other members of the Pascal family, then gives a somewhat less hasty tour of the more novel features of Mesa, followed by a discussion of the additional changes that produced Cedar. Finally, there is a guide to sources of further information.

Version 5.2 of the Cedar language documentation corresponds to Release 5.2 of the Cedar system. It is intended to supersede all descriptions prior to June 1984. Previous documents may be read for historical interest, but are believed only at the reader's peril.

[If you are reading this document on-line, I suggest that you use the Tioga Levels and Lines menus to initially browse the top few levels of its structure before reading it straight through.]

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

For Internal Xerox Use Only

Cedar Language Overview: Contents

Introduction

The programming language of the Cedar Programming Environment (hereafter, Cedar Language, or just Cedar) has resulted from an evolutionary process in PARC and SDD that spanned more than a decade. Understanding what the language is, and why it is that way, may be somewhat easier with a little historical background:

> Mesa is a system implementation language in the "Pascal family," with extensive facilities for modularization and separate compilation, processes and monitors, exceptional – condition handling, and control of low – level hardware functions. It was initially designed and implemented in the PARC Computer Science Laboratory, primarily by Butler Lampson, Chuck Geschke, Jim Mitchell, Ed Satterthwaite, and Dick Sweet. Subsequently, the OSD System Development Department assumed responsibility for development and maintenance. It has gone through a series of releases.
>
> When CSL launched the Cedar Project in 1979, it chose to use the Mesa language and system as a starting point. (Mesa 6, 7, and 8 are its closest relatives.) However, Mesa did not have a few of the features that seemed to be important for an experimental programming environment, so some extensions and changes were designed. The major changes resulted from adding automatic storage deallocation (garbage collection) and facilities for delaying the binding of type information, without sacrificing complete type – checking in either case.

This Overview is intended to introduce a competent programmer to the basic vocabulary and concepts that are needed before plunging into sources of more detailed information about the Cedar Language. It assumes that you know some other language in the Pascal family. It also assumes that you have already read the Briefing Blurb and the Introduction to Cedar. If you haven't, read them first and return.

This Overview starts with a brief review of the common concepts that Cedar shares with other members of the Pascal family, then gives a somewhat less hasty tour of the more novel features of Mesa, followed by a discussion of the additional changes that produced Cedar. It ends with a survey of sources for further information.

This Overview does not provide the detail you need to actually write Cedar programs. (In particular, the reference grammar is included but not discussed.) But when you finish reading it, you should have a fair acquaintance with Cedar terminology and concepts, and you should have a good idea of what you need to learn. Different things are discussed in varying depth; generally the long discussions cover things that you should plan to study carefully.

Cedar documentation is still evolving. Comments and suggestions on how it can be made more useful are welcome at any time. Although we plan a systematic attempt to assess the effectiveness of the various kinds and pieces of documentation, you should not wait until asked to let us know what you think about it.

Various proposals and descriptions of interim implementations from September 1979 onward have been given labels such as 5C1, 5C2, 6C2, 6C5, 7T11, and Version 3. Version 5.2 of the Cedar language documentation corresponds to Release 5.2 of the Cedar system. It is intended to supersede all descriptions prior to June 1984. Previous documents may be read for historical interest, but are believed only at the reader's peril. This Overview has been compiled by Jim Horning and Jim Donahue; errors and sources of confusion should be reported to Jim Donahue. Most of the contents have been abstracted from previous documents, with a small amount of editing and validity checking.

Review of the Pascal – like features

The following summarizes aspects of Cedar (and Mesa) that are basically similar to those of other members of the "Pascal family" of languages (e.g., Euclid, Modula, Ada). If there are any concepts in this section that are not already familiar to you, you should probably find a Pascal textbook and study it before proceeding to further material on Cedar. (You will find that the names for these concepts vary somewhat from language to language.)

An algorithm or computer program consists of two essential parts, a description of actions that are to be performed, and a description of the data that are manipulated by these actions. Actions are described by statements, and data are described by type definitions.

Data and types

Data are represented by values. Values are immutable; they are not changed by computation. A constant always denotes the same value within a scope. A variable is a value that may contain another value; assignment changes the value contained by a variable, but not the value that is the variable.

A value used in a program may be represented by a literal constant, the name of a constant or variable, or by an expression, which will itself contain other values. Every name occurring in the program must be introduced by a declaration. A declaration associates with a name both a data type and a constant value (which may itself be a variable, and contain different values at different times).

A data type defines both a set of values and the actions that may be performed on elements of that set. It may either be directly described in a declaration that uses it, or it may be referenced by a type name, introduced in a type declaration. The type of every constant, variable, and expression can be deduced from static analysis. This analysis is performed by the compiler to ensure that all programs are type – correct; thus the language is said to be strongly typed.

An enumerated type definition indicates an ordered set of values, i.e., introduces names standing for each value in the set. The simple types are the enumerated types, the subrange types, and the built – in types, including BOOL, INT, REAL, and CHAR. There are standard denotations for literal constants of the built – in types: TRUE and FALSE for BOOL, numbers for INT and its subranges and for REAL, quotations for CHAR. Numbers and quotations are syntactically distinct from names▮as are the "reserved words" of the language. The set of values of type CHAR is an 8 – bit variant of the ASCII character codes.

A type may be defined as a subrange of a simple type by indicating the smallest and largest value of the subrange.

Structured types are defined by describing the types of their components, and indicating a structuring method: ARRAY or RECORD. These differ in the mechanism for selecting a component of a value.

In an array structure, all components are of the same type. A component is selected by a computable selector, or index. The index type, which must be simple, is indicated in the array type definition. It is usually a programmer – defined enumerated type, or a subrange of INT. Given a value of the index type, an array selector yields a value of the component type. Every array structure value can therefore be regarded as a mapping of the index type into the component type.

In a record structure, the components (called fields) are not necessarily of the same type. In order that the type of a selected component be evident from the program text (without executing

the program), a record selector is not a computable value, but must instead be a name uniquely denoting the component to be selected.

A record type may be specified as consisting of several variants. This allows different record values of the same type to have structures that differ in the number of components, their types, or their names. The variant describing a particular value is indicated by a special field, called its tag. Variants of a type may also share fields in addition to the tag.

An explicit variable declaration associates a name and a static variable; the name is used to denote the variable in expressions. Dynamic variables are generated by a special procedure (NEW) that yields a pointer or reference value that subsequently serves in place of a name to refer to the variable. Finite graphs in their full generality may be represented using pointers or references.

Statements

The simplest statement is the assignment statement. It specifies that a newly computed value be assigned to a variable (or a component of a variable). The value is obtained by evaluating an expression. Expressions consist of variables, constants, operators, and procedure values operating on arguments to produce new values. Constants are literal or declared; variables and procedures are built – in or declared; the set of operators is defined within the language, and includes operators for arithmetic, comparison, and logical operations.

The procedure statement causes the application (invocation, call) of a designated procedure value to the values of its arguments (actual parameters).

Basic statements are the components of structured statements, which specify sequential, selective, or repeated execution of their components. Sequential execution of a sequence of statements is specified by separating them by semicolons; conditional or selective execution by the if statement and the select statement; and repeated execution by loop statements.

A block can be used to associate declarations with statements. The names so declared have significance only within the block. Hence, the block is the scope of these names, and they are said to be local to the block. Since a block may appear as a statement, scopes may be nested.

A block can be the body of a procedure value. A procedure has a fixed number of parameters, each of which is denoted within the procedure by a name called the formal parameter. Actual argument values are supplied for parameters at each application.

Procedures may also have results; applications of such procedures may appear within expressions.

From Pascal to Mesa

Mesa extended Pascal in a number of directions intended to make it more effective for the development of large systems. Students of programming languages will discern influences from Algol 68, BCPL, and several other system implementation languages. It is a larger language, and is rather more difficult to master in its entirety, than Pascal. It is intended for professional programmers, not for beginning students.

Mesa modules are separately compiled program units, with type – checking preserved across module boundaries. Mesa provides mechanisms for systematic handling of exceptions, processes and monitors, procedures as first – class values that can be assigned to variables, and a fair number of

syntactic and semantic amenities intended to make programming more convenient.

The following sections introduce each of the major conceptual extensions, but do not explain them in great depth. See [Geschke, et al.] for a more extensive rationale, and CSL – 79 – 3 for full details.

Modules

Mesa modules are a "programming in the large" mechanism for partitioning a system into manageable units. They can be used to encapsulate abstractions, to provide a degree of protection, and to enforce "information hiding." They are also the units of separate compilation.

There are two kinds of modules: DEFINITIONS modules, which define interfaces, and PROGRAM modules, which contain the executable code to implement these interfaces.

Definitions (or defs) modules define interfaces to abstractions. They typically declare some shared types, useful constants, and the domains and ranges of a set of procedure names. They compile into symbol tables, which are shared by both clients and implementations. Checks are performed when modules are bound into a configuration to ensure that separately compiled pieces have used consistent versions of the shared definitions. Interfaces produce no executable code; they manifest themselves at runtime primarily as symbol tables that are accessible for debugging and similar purposes.

Program modules provide implementations of abstractions. They typically declare collections of variables that define their state and provide bodies for the procedures of their interfaces. Viewed as source text, they are similar to Pascal procedures and Simula class definitions. They can be loaded and interconnected to form complete systems.

At runtime, one or more instances of an implementation may be created. A separate global frame (activation record) is allocated for each, containing storage for its global variables (those which are declared outside its procedures), which persist between applications of its procedures. The lifetimes of implementation instances (unlike those of procedure applications) are not restricted to follow any particular discipline. Communication paths among implementations are established dynamically and are not constrained by any (static or dynamic) nesting relationships; lifetimes and access paths are completely decoupled. The module body itself generally contains the code to initialize the global variables and establish any necessary invariants. It will be executed when the module is started, or upon application of one of the module's procuedures, whichever comes first.

A module that accesses (relies on declarations from) other modules must include DIRECTORY statements, so the necessary symbol tables can be acquired. If it uses only a subset of the declarations, it is good practice to indicate which ones with a USING list. Declarations in an interface are public unless declared to be PRIVATE. Normally the importing module accesses only the public names; private declarations may be accessed by implementing modules that indicate they SHARE the interface. A directory statement may list the name of a file containing the symbol table to be used, but if the file name is the same as the module name (except for the extension .bcd) it is omitted.

A module that uses non – constant declarations (e.g., exported types and procedures) from another module must explicitly import it. If a module implements any part of an interface (e.g., by supplying the value of a procedure or type that it declares), it must explicitly export it. The compiler will check that its PUBLIC declarations are type – consistent with the corresponding declarations in the exported interface(s).

Each module is effectively parameterized by a set of interface records, one for each interface it imports, and supplies a set of export records, one for each interface it exports. Note that interfaces and implementations need not be in one – to – one correspondence. Binding a group of modules together into a configuration involves assigning values from the export records to the corresponding fields in the interface records. There is a special sublanguage, C/Mesa, to control this process.

Accessing other modules introduces compilation order dependencies. Each module must be compiled after the modules it accesses (and recompiled if they change), since the compiler needs their symbol tables. But information does not flow in the other direction. Modules that are not accessed by others (virtually all implementations) may be freely recompiled without invalidating previous compilation and checking of any other modules.

Types, as well as procedures, can be declared opaquely in interfaces and subsequently bound to concrete values supplied by implementations. This makes the internal structure of the type invisible to clients of the interface, and ensures that there can be no compilation dependencies between the definition of the concrete type and the interface module. The definition of the type can be changed at any time without requiring recompilation of the interface or any clients of the interface.

Effective use of Mesa requires a thorough understanding of modules and their use. They have significantly influenced our program design and construction techniques.

Programs are almost never self – contained modules; the importation and re – use of existing code has all the advantages of theft over honest toil▮without the moral stigma. Considerable emphasis is laid on the careful design of interfaces, and on their documentation. Since it is only interface changes that force recompilation (or perhaps even rewriting) of client programs, it is important that interfaces remain stable for substantial periods, even while their implementations are undergoing change.

A recommended approach is to define, comment, and circulate for review, all of the interfaces in a (sub)system before writing any of the implementations. Interfaces play much the same role as "program design languages" in other environments, with the additional advantages of being precisely defined and mechanically enforced.

The Mesa language definition omits many of the features commonly expected in programming languages, such as input/output and string – manipulation operations. Of course, these facilities are available to Mesa programmers, but they are provided by packages written in the language itself. The descriptions of standard packages in the Mesa Programmer's Manual, Version 8.0, run to more than 300 pages.

When managing large collections of modules (and in systems like the Mesa Development Environment and Cedar they run into the thousands), module names become very important. The use of cryptic or acronymic names is discouraged. By convention, source file names have the extension .mesa, and object file names have the extension .bcd (for Binary Configuration Description). The definitions module for an interface X is customarily named X; if it is implemented by a single program module, that is customarily named XImpl.

Exceptions

Mesa provides a way to indicate when exceptional conditions arise in the course of execution and an orderly means for dealing with them that is inexpensive if they do not arise. Exceptions cause a transfer of control from the statement that raises them to a dynamically – selected part of the program intended to handle the situation. They may be raised in response to the detection of

"impossible" situations, invalid inputs, the inability of an abstraction to supply its specified service, or simply unusual events.

Mesa exceptions are conceptually similar to procedures, except that the binding to the handler is determined by searching the catch phrases in the call stack of the process in which the exception is raised; the dynamically innermost handler that accepts the condition is applied. Like normal procedures, handlers can take parameters and return values. They are written in a distinctive syntax that clearly identifies them as code for the exceptional case.

Catch phrases are syntactically and semantically similar to SELECT statements, with test items indicating the exceptions for which the associated handler should be applied. There are special test items to catch arbitrary exceptions and to catch an attempt to unwind the application stack in response to an exception. A series of catch phrases may be associated with a procedure application, or enabled throughout a block.

A handler is like a procedure body, but when it completes, there are a number of additional control options: GOTO, EXIT, LOOP, RETRY, CONTINUE, REJECT, and RESUME. Resumption is analogous to returning from a procedure, possibly with a result. Exceptions are divided into SIGNALS, which may be resumed, and ERRORS, which may not; in common parlance they are generally all called signals.

Since handlers may take parameters and return results, each exception name must be declared in a scope that includes all the points where it is raised as well as all the catch phrases that accept it.

The cost of raising an exception is significantly higher than the cost of procedure application, but it shouldn't happen very often. The system guarantees that all exceptions are handled at some level; those that the program fails to catch are accepted by the debugger, keeping intact the state of the program that raised it.

Exceptions can be used in very intricate ways to achieve subtle effects (e.g., by raising another exception within a handler). Experience has shown that this is almost always a mistake. Some call it elegance, others call it incomprehensible:

> "For the programmer, the main import of nested signals is that one needs to consider, when writing a routine, not only what signals can be generated, directly or indirectly, by the called procedures, but also those which can be generated by catch phrases in that procedure or even the catch phrases of any calling procedures, also both directly and indirectly." [Mesa Language Manual]

Although his language proposals have not been implemented, The discussion in the working paper [Indigo]<Cedar5.2>Documentation>SignallingGuidelines.press is the best source of guidance on tasteful and appropriate uses of exceptions. The most important point is that the exceptions a procedure may raise must be considered part of its interface, and documented as such. Unfortunately, the compiler currently doesn't enforce this, and many otherwise excellent interfaces do not comply.

Processes, monitors, and condition variables

Mesa provides efficient mechanisms for concurrent execution of multiple processes within a single system. This makes it natural to structure programs to reflect their inherent concurrency. Mesa also provides facilities for mutually exclusive access to resources and process synchronization by means of entry to monitors and waiting on condition variables.

FORK makes it possible to start the execution of another procedure concurrently with the program that applies it. It returns a process, which may either be detached to proceed

independently, or saved for a future JOIN. There is no rule against multiple coexisting instances of a procedure, either forked or applied, although care must be taken to ensure mutual exclusion on accesses to shared global data.

JOIN takes a single process argument. When the forked procedure has executed a RETURN and the JOIN has been executed (in either order), the returning process is deleted, and the joining process receives its results and continues execution. A process type is declared similarly to a procedure type, except that only the type of the result is specified.

All processes execute in the same address space. This means that they are not protected from each other, which is presumably acceptable in a single – user system. It also means that process creation and switching between processes is cheap (not much more time – consuming than a procedure call).

Generally, two or more cooperating processes need to interact in more complicated ways than simply forking and joining. The interprocess synchronization mechanism provided in Mesa is a variant of "monitors" adapted from the work of Hoare, Brinch Hansen, and Dijkstra. The underlying view is that interaction among processes is always based on access to shared resources (e.g., data) and that a proper vehicle for this interaction must unify the synchronization, the shared data, and the procedures that perform the accesses.

A monitor is typically a module instance, with shared data in its global frame, and its own procedures for accessing them. Some of the procedures are public, allowing applications of monitor procedures from outside. Obviously, conflicts could arise if two processes were executing in the same monitor at the same time. To prevent this, a monitor lock is used for mutual exclusion. Application of one of a monitor's ENTRY procedures automatically acquires its lock (waiting if necessary), and a return releases it. An integrity constraint that the programmer imposes on the monitor's data is called a monitor invariant. The lock makes it possible for the programmer to ensure that this invariant will be true whenever an entry procedure begins execution—regardless of what is happening in various processes—simply by making sure that it is true initially and that every entry procedure restores it before returning.

Of course, a process may enter the monitor and find that the monitor data is in a good state but indicates that the process may not proceed until some other process enters the monitor and changes the situation. The WAIT operation allows a process to release the monitor lock temporarily (and suspend execution) without returning. The wait is performed on a condition variable, which is associated by agreement with the actual condition needed. After making a change that may have changed the condition, some other process must perform a BROADCAST or NOTIFY on the condition variable; this allows a waiting process to reacquire the lock, retest the condition, and resume execution if it is true. Note that since a wait releases the lock, the monitor invariants must be restored before waiting.

The procedures of a monitor are classified as entry, internal, and external. Internal procedures may only be applied by entry or internal procedures of the same monitor, since they are intended to be executed within the monitor's mutual exclusion, but do not acquire the monitor lock. External procedures are logically outside the monitor, but are declared within the same module for reasons of logical packaging. Being outside, they must not reference any monitor data nor apply any internal procedures; they are often used to provide a convenient interface that "hides" one or more applications of entry procedures.

The attributes ENTRY and INTERNAL are associated with a procedure's body, not with its type; thus they do not appear in interfaces. From the client side of an interface, a monitor appears like any other module.

In simple cases, a monitor's data comprises its global variables, protected by an implicit lock that is automatically allocated in its global frame. However, many applications deal with multiple objects, represented, say, as records accessed through pointers. It may be necessary to ensure that operations on these objects are atomic, i.e., once the operation has begun, the object will not be otherwise referenced until the operation is finished. It is possible to associate a lock with the object, rather than with the module's global frame, by declaring the data as a MONITORED RECORD. A single module instance can then implement each operation as an entry procedure, taking the object as a parameter. Locking is specified in the module heading by a LOCKS clause.

A somewhat subtle source of deadlocks occurs if control leaves an entry procedure by means of an uncaught exception. Unless it is certain that all exceptions (including those raised by invoked procedures) are handled, each entry procedure should include an UNWIND catch phrase, which will implicitly release the monitor lock.

Control constructs

Mesa's facilities for ordinary sequential "programming in the small" are extensive, but fairly conventional. The syntax is not exactly like that of any other language, but for the most part it can be picked up easily with a few minutes study of the grammar. (In fact, since most program text is produced either by editing existing programs or by the use of the Tioga editor to expand syntactic templates, you may be able to just "fake it.") This section mentions a number of areas where Mesa provides "convenience" extensions or conceptually small changes.

SELECT statements generalize Pascal's "case" construct by allowing several ways to specify how one statement is to be chosen for execution from an ordered list. The most common form is based on the relation between the value of a given expression and those of expressions associated with each selectable statement. The relation may be equality (the default), any relational operator appropriate to the types of the values involved, or containment in a subrange. A single selection may be prefixed by several selectors, and an optional ENDCASE statement is selected only if none of the others are. Discriminating selection is used to branch on the type of a variant record value (and in Cedar, on the current type referred to by a REF ANY). SELECT expressions are analogous, but choose from an ordered list of expressions.

Iteration is provided by loop statements in which several different kinds of control can be freely intermixed. A loop has a control clause and a body. The control clause may specify a logical condition for normal termination, possibly combined with a range or a sequence of assignments for a controlled variable. In addition to ordinary statements, the body may contain EXIT or GOTO statements to explicitly terminate its execution, and may be followed by a REPEAT clause that acts like a selection on the GOTO used to terminate the loop. (GOTO cannot be used to synthesize arbitrary control structures. It is much more like a "local" exception.)

In Pascal, procedure execution must proceed somehow to the end of the body before terminating; in Mesa, it can be terminated anywhere by executing a RETURN statement. If the procedure's type includes results, the return statement may supply the values to be returned▮otherwise they are taken from the result variables named in the type. Each procedure body is followed by an implicit return.

Pascal procedures are not values that may be assigned to variables; Mesa procedures are. In most cases, the programmer still thinks of a constant association between a procedure name and its body, but to truly understand what is going on when interface records are bound, it helps to realize that procedure values from the export records are being assigned to appropriate fields of the interface records. This same power is available to the Mesa programmer; one popular form of "object – oriented programming" is based on the creation of an explicit record of procedures for

each kind of object, and passing around together a pair of pointers, one to the procedure record, and another to the object instance data.

INLINE procedure constants may be declared in interfaces or locally. This is an instruction to the compiler to expand the procedure body inline for each application, rather than compiling a call to out – of – line code. It is intended to improve the speed without changing the semantics of the procedure. Inlines are not macros. INLINE should be considered a form of tight binding best reserved for late stages of system tuning; among other things, it can cause the compiler to run out of resources, even when compiling what appear to be small modules.

In addition to procedures and exceptions, Mesa has a third mechanism for transfer of control, called a PORT. When used in pairs, ports can provide a very general form of coroutine implementation. In some circumstances, coroutines have advantages similar to processes, at slightly lower cost, but they are not used much in Mesa or Cedar.

Miscellaneous

Every expression in a Mesa program has a syntactic type that can be deduced from its structure by static analysis of the program text, a process called type determination. The language imposes constraints on the type of each expression according to the context in which it is used, even in separately compiled modules.

The syntactic type of a name is established by declaration.

The form of a literal implies its type.

Each operator produces a result with a type that is a function of the types of the operands.

The type rules in Mesa take two general forms:

The type required by the context is known exactly, and a given expression must have it. The required type is called the target type. Examples occur in assignment, initialization, record construction, array construction, argument list construction, and array subscripting. Several coercions (e.g., pointer dereferencing, base/subrange conversion, single – component record to field) will be applied if needed to convert a value whose syntactic type is not its target type to one that is.

The exact type is not implied by context, but a relation that must be satisfied by a set of types is known. The process of finding types to satisfy that relation is called balancing. Examples include generic operators (such as relationals) that require two operands of the same type, conditional expressions, and select expressions. The common type selected will be the one requiring the fewest coercions.

A sequence in Mesa is an indexable collection of items, all of which have the same type. In this respect, a sequence resembles an array; however, the length of the sequence is not part of its type. The (maximum) length of a sequence is specified when the object containing that sequence is created, and it cannot subsequently be changed. It is the responsibility of the programmer to keep track of the number of items in the sequence at any time.

Mesa allows a default initial value to be associated with a type. If a type is constructed from other types using one of Mesa's structures, such as RECORD, an implicit default value for the constructed type is derived from the default values of the component types, but it can be overridden with an explicit default value. Default values for arguments can simplify procedure applications; default fields of records make the corresponding constructors more concise and more convenient; initial values are useful to ensure that the corresponding storage is always well – formed, even before the variable has been used by the program.

Dynamic variables in Mesa are allocated in zones. These are not necessarily associated with fixed areas of storage; rather, they are objects characterized by procedures for allocation and deallocation. There is a standard system zone, but programs that allocate substantial numbers of similar dynamic variables can often improve performance by segregating each kind into its own zone. The operator NEW is used to create a dynamic variable in a zone, and FREE to release it.

The MACHINE DEPENDENT attribute allows precise control of the representation of values at the bit level.

From Mesa to Cedar

The Cedar Language is very closely related to Mesa. The most radical change is the provision of automatic deallocation of dynamic storage, or garbage collection. Several other changes extend

the range of binding times available for such important attributes as the types of variables.

It is intended that most Cedar programs will be written in the safe subset, which imposes a number of restrictions not present in Mesa to ensure the safe operation of the garbage collector, and introduces some new (safe) features to make these restrictions tolerable. The full (unsafe) language is generally "upward compatible" with Mesa.

Garbage collection, collectible storage, and REFs

Although Mesa pointers are typed, they provide a rich source of opportunities for creation of safety problems, including the classical dangling pointer problem, where a pointer is used after the storage it refers to has been deallocated, and the opposite storage leak problem, where storage becomes inaccessible without being deallocated for reuse. Freeing the programmer from responsibility for deallocating storage at just the right time was a major goal of Cedar. It adds a new class of REF types that are just like the corresponding pointer types except that the system is responsible for freeing the dynamic variables they refer to after they have become inaccessible.

Cedar provides three types of storage:

Frame: This is storage that is implicitly allocated by a procedure application or an implementation instantiation to hold variables declared in the corresponding scope. It is also implicitly deallocated, upon exit from the scope (e.g., return from the procedure).

Collectible: This is storage that is explicitly allocated by NEW, and implicitly deallocated after there are no more accessible REFs to it. FREE applied to a REF variable will cause it (and REF fields in the dynamic variable it refers to) to be "NILed out," but the dynamic variable will only be freed when no other REFs to it remain.

Heap: This is storage that is explicitly allocated by NEW, and deallocated by (unsafe) FREE statements, as in Mesa. Heap storage is referenced by pointers, which may not be dereferenced in checked regions, and should not refer to dynamic variables containing REFs.

The introduction of collectible storage has substantially revised programming style and interface design in Cedar. When the project was being contemplated, some Mesa programmers indicated that as much as 40% of their time went into designing and checking the code to avoid dangling pointers and storage leaks, to tracking errors in this code, and to wasting time in tracking other errors by suspecting storage deallocation problems. With REFs and a reliable garbage collector that all goes away.

Frame (static) variables are still less expensive than dynamic variables, since entire frames are allocated and freed on procedure entry and exit (and the mechanism for doing it has been rather carefully tuned). However, it is entirely reasonable to use dynamic variables for data whose lifetime is not closely connected to a particular procedure application or module instance. Objects of large or varying size are almost always passed across interfaces by reference. Definitive measurements on the cost of garbage collection have not yet been made, but preliminary data indicates that it is generally less than 20%. Only in very special circumstances is heap storage worth the added program complexity and potential for errors.

Safety

A desirable property of a high – level language system is implementation independence. This means that the effects of (even erroneous) programs can be understood in terms of the language rather than requiring an understanding of the particular implementation. Mesa comes

rather close to meeting this goal (as evidenced by the fact that most Mesa debugging can be done "at the Mesa level," without ever worrying about the format of frames or the details of storage management), but it does contain some unsafe features whose use can lead to messy implementation dependencies.

It was desirable on general grounds to reduce implementation – dependence in Cedar. However, the decision to include facilities for garbage collection made it imperative. A collector can cause storage to be deallocated (permitting its subsequent reallocation and re – use) at times that are completely unpredictable from examination of the source program. A single programming error that smashes a REF used by the collector can destroy data structures in ways that make it difficult to reconstruct any evidence of the original cause of the crash.

A major goal for the Cedar Language was that it contain a useful subset for which garbage collection would be safe. The safe subset of Cedar is basically that part of the language where even incorrect programs cannot interfere with the reliable operation of the collector. The vast majority of Cedar programs should be written primarily (or entirely) in the safe subset. Safe Cedar does not provide acceptably efficient substitutes for every use of Mesa's unsafe features, so Cedar provides a means for indicating that some regions of a program are trusted. This inhibits compiler enforcement of the safety restrictions and indicates that the programmer has assumed the additional responsibility of ensuring that these regions of the program do not violate the integrity of the system.

Invulnerability, safety, and checking

It is an obviously desirable property of a programming system that no user programming error can "break" its abstract machine and reduce its world to a rubble of bits. We call this property invulnerability. In general, it can be ensured only by maintaining the integrity of certain data structures known to the runtime system. Collectively, the properties that must be maintained to ensure invulnerability are called the safety invariants; each part of the system is responsible for ensuring that they are not destroyed, and must assume that the rest of the system does likewise.

Unfortunately, invulnerability is not a local property. If any part of the system fails to maintain the invariants, the entire system (including programs that are themselves correct) is potentially vulnerable. We use the term safety for the property that the invariants cannot be invalidated locally, even by incorrect programs. Cedar operations, both built – in and programmer – defined, are classified as safe or unsafe. Most of the Cedar Language is safe.

> Unsafe constructs include LOOPHOLE, dereferencing POINTERs (but REFs are safe), JOIN, @ (address of), computed variant records, and non – copying variant discrimination.

A region of program text, bracketted to form a block, may be prefixed with CHECKED, TRUSTED, or UNCHECKED.

In checked program regions, language – enforced restrictions guarantee safety. If a block is checked, then within that block only safe operations may be used, the block itself implements a safe operation, and procedures declared in the block are treated as safe.

Even unchecked regions are supposed to maintain the safety invariants, but the guarantee must be by the programmer, rather than the system. If a block is unchecked, unsafe operations may be used internally, the block itself is considered to implement an unsafe operation, and procedures declared in the block are treated as unsafe. Generally even unchecked regions can be composed primarily of safe operations; unsafe operations should be used only for good reasons and with due caution.

A trusted block may also invoke unsafe operations, but it is assumed to implement an

operation that is safe by programmer guarantee. TRUSTED is a programmer assertion that cannot be checked by the compiler, and therefore represents a special kind of loophole.

For easy upward compatibility from Mesa, the following defaults have been adopted: If a module is prefixed with CEDAR, then the outermost block is CHECKED and all interfaces are assumed to be safe; otherwise, the outermost block is UNCHECKED and all interfaces are assumed to be unsafe. The checking attribute is inherited; unless a nested block is explicitly prefixed, it is checked or unchecked like the textually enclosing block.

If a system consists entirely of safe regions (and the invariants hold initially), then by induction the system is invulnerable. However, an error in an unchecked region can make even the checked regions vulnerable. Thus the CHECKED/UNCHECKED boundary limits responsibility, but not vulnerability. Confidence that errors in checked regions will not cause system crashes is based on the the automatic enforcement of safety restrictions. Confidence that unchecked regions will not cause system crashes is based on trust that they are free from errors that violate the safety invariants.

Caveat: The conversion of the Cedar system to safe interfaces is presently underway. The unsafe interfaces are beginning to disappear. You should program as safely as you can, but do not be surprised by the initial density of safety complaints from the compiler. A good rule is to prefix each module with CEDAR, and then to put TRUSTED on each block about which the compiler complains, after convincing yourself that the complaint is not your fault, because it results from a necessary use of an unsafe system interface. The reason for each TRUSTED should be documented in an accompanying comment.

Type confusion

Mesa is a strongly typed language, which means that the types of names are declared, and that the language imposes restrictions to keep values of one type from being accidentally interpreted as values of another. Because knowledge of the type structure of values in memory is so essential to the garbage collector (it must locate and follow REFS in order to determine current storage usage), it is particularly vulnerable to any operations that cause data in memory to be interpreted as having other than their true types. Thus, much of the effort in designing the safe subset went into identifying all the features in Mesa that allow type – checking to be circumvented (accidentally or deliberately) and designing safe replacements for the important uses of those features.

LOOPHOLE is a "type converter" in Mesa that allows any value to be treated as having any specified type; it is the most obvious breach of type security. It causes a safety problem only if it allows mistyped data to be stored into memory (i.e., if the target type contains an address, such as a pointer or procedure value); other uses will introduce implementation dependencies, but not threaten safety. Within checked regions, LOOPHOLE is not allowed to produce a value of a reference – containing (RC) type.

Narrowing and type discrimination

Cedar introduces a number of new type distinctions, frequently leading to a number of separate, but closely related types. It is often desirable to coerce a value of one of these types into a value of a related type. Where the types are such that it can be statically guaranteed that no information will ever be lost by the coercion, it is called a widening, and is performed automatically whenever demanded by context (e.g., assigning a bound variant value to a variant record variable). In general, conversion in the other direction requires a runtime check to ensure that information is

not being lost. To make the possibility of such failure explicit in the program text, the NARROW type converter may be applied (and may include a catch phrase to handle the NarrowFault exception).

The built – in test ISTYPE can be applied to a value to determine whether it can be narrowed to a specified type without error. If so, it is said to satisfy the type's predicate.

If the target type of a narrowing is uniquely determined by context, it need not be an explicit argument to NARROW.

Delayed binding

A desirable property of a high – level programming language is that is allow a wide range of binding times: that is, it should allow the programmer maximal control over when the attributes of a particular variable are determined, with different choices not requiring changes in all expressions containing the variable. Examples of such attributes are its type, storage allocation method, implementation (for abstract objects), and actual value; examples of binding times include program – writing time, compilation, configuration binding, program initialization, block entry, and statement execution. Generally speaking, deferring the binding of an attribute leads to greater generality in the program at the cost of decreased static checkability and (often) lower runtime efficiency.

Experience with languages like LISP and Smalltalk, in which most binding is done dynamically, shows that it is much easier to write certain kinds of programs, if type and/or implementation binding can be deferred. Programming tools (debuggers, performance monitors) and knowledge representation systems are typical examples. But few programs take full advantage of this flexibility very often. Cedar was designed to take advantage of early binding, as Mesa does, but to allow certain bindings to be explicitly deferred.

Dynamic typing, REF ANY, and dynamically typed procedure variables

Mesa provides very limited variability in the binding time of an object's type. Variant records allow a deferred choice between specific enumerated alternatives, and sequences allow deferring the specification of an object's length until it is allocated. Otherwise, all types must be static. This makes it virtually impossible to avoid LOOPHOLEs and ad hoc type tagging schemes when writing schedulers, sorters, output formatters, etc. that must operate on objects of unpredictable type.

Cedar's solution to this problem requires two new mechanisms: a runtime representation for types, and a way to associate a type with an object at runtime that is guaranteed consistent with the type system and static checking. (Note that Cedar adopts the view that an object's type is inherent in the object itself, rather than in the way the object is referred to.)

TYPE is a type in the Cedar Language. The "structuring methods" (e.g., ARRAY, RECORD, and REF) are viewed as operators that take type arguments and return type values as results. In the current language, the arguments to such operators must be static (compile – time) constants.

ANY is not a type in Cedar, but can stand in place of a type in the arguments to two operators: REF and PROC.

A REF ANY value may refer to a dynamic variable of any type whatsoever. Thus a REF T value, for any T, can be widened to a REF ANY value. But a REF ANY value cannot be directly dereferenced, because the type of the result is not static. The discriminating selection statement has been generalized to allow discrimination on the referent type of a REF ANY; within each selectable statement, the type is (statically) known to be the type specified in its test item. NARROW can also be used to safely convert a REF ANY value back to a REF T value; ISTYPE can be used to check whether NARROW will succeed.

A PROC type may also have ANY in place of the type of its formal parameter record type and/or result record type PROC values with specific domains and ranges may be widened to these dynamic types, and later tested and narrowed analogously to REF ANYs. They must be narrowed before being applied.

In principle, each value in Cedar carries its syntactic type with it at all times. In practice, almost all analysis and checking of types is done by the compiler, and both space and time efficiency are gained by not storing constant types with values. However, the symbol tables produced by the compiler contain enough information to recover any type on demand, made available through a standard package. AMTypes provides type – conversion routines in both directions between typed values (with type SafeStorage.Type) and ordinary Cedar values, and numerous operations on typed values to examine the type and structure of a typed value, to change its attributes, etc. Thus it is possible to write a program that deals with any given Cedar value or type without anticipating the specific type when the program is written. Programs such as BugBane (the Cedar debugger) absolutely require such flexibility.

The current implementation is too slow to be used effectively by client programs as a substitute for true polymorphism in the language, but is suitable for examining and changing variables interactively with the Cedar debugger.

Miscellaneous

Although Cedar was not intended as a research project in programming languages, its developers were not immune to the temptation to make Mesa better in ways that were not strictly required to enable the new programming environment. This section discusses a few of these new features.

Types as clusters of operations

Each type has an associated cluster of operations. The main purpose of this association is to support a style of "object oriented" notation. Using a record – like notation, a procedure "field" will be looked up in the cluster of the object's type, and then applied to the object and the other arguments.

It is preferred style in Cedar to use this object notation in invoking operations of interfaces designed to support it. Consult the relevant package documentation if in doubt.

Each built – in type and type constructor in Cedar implicitly supplies a standard cluster. The cluster extension mechanism is that each opaque or record type defined in a interface acquires all procedures declared in the same module as parts of its cluster.

ROPEs and IO

Mesa STRINGs are rather awkward objects, having been tuned for efficiency in a small – machine (Alto) world, rather than for flexibility and convenience. They are POINTERs to fixed – length sequences of characters. Considerable care is required to avoid surprising results, even for rather straightforward string – processing applications. Cedar ROPEs, on the other hand, are somewhat heavier – weight, more convenient to use, and less prone to surprises. Several different implementations of ropes, efficient for different purposes, provide the same interface.

Rope is a Cedar package that supports the creation and manipulation of immutable reference – counted sequences of characters. Procedures are provided for concatenation, taking substrings, scanning, and other operations. A client can provide specialized implementations for rope objects. The standard implementation attempts to avoid copying when performing Substr, Concat and Replace operations. The Rope package is the standard support for sequences of characters in Cedar,.

Most of the common operations on input/output streams, plus string conversions that are commonly used in dealing with input or formatting output, have been collected in the IO interface. Implementations are available for stream interfaces to all common devices, and to allow ropes and streams to be readily interconverted.

LISTs and ATOMs

Cedar includes LIST OF as a new type constructor for singly – linked (by REFs) lists, and a constructor for list values that mimics that of LISP, avoiding the need for a lot of NEWs or CONSs. The analog of LISP's CAR and CDR are provided by the standard fields first and rest. Unlike LISP, Cedar lists are statically typed (although the element type may be REF ANY).

Cedar also has a built – in type ATOM, which can be used for values that are uniquely determined by their print names. Any rope can be converted to an atom and conversely; the advantage of atoms is that, unlike ropes, it is very cheap to compare them for equality; atoms may also have property lists. Atom literals are just names prefixed by ¤.

Converting Mesa Programs to Cedar ▮ Jim Morris

This section assumes you already know how to program in Mesa (or that you have a Mesa program to be converted), and is intended to explain the differences for programming in Cedar.

Simple programs

Let's suppose you want to run a simple program in Cedar. If an existing Mesa 5 or 6 program uses fairly vanilla stuff, it's easy to convert:

The names of most interfaces and some procedures have changed, but the functionality is basically the same.

The most obvious differences will be with strings and I/O. You should only need to know about two interfaces for these: Rope and IO, respectively.

In general, the Cedar community has dropped the use of "Defs" as a suffix for definition file names, and introduced the suffix "Impl" for implementation files; e.g. "InlineDefs" became "Inline".

Here's what you need to do to your Mesa 5 or 6 program:

Change all STRINGs to ROPEs (actually Rope.ROPE). Remove all allocations and deallocations of strings. Change all references to StringDefs routines to use Rope or IO routines. Rope provides procedures to parse and manipulate ropes. IO provides procedures to convert ROPEs to numbers and back as noted below. One can now put special characters in rope literals by using the escape character "\". "...\n..." inserts a carriage return (newline), "...\t..." a tab, "...\\..." a backslash, and "...\123..." the character whose octal code is 123. Note a ROPE is immutable, unlike a string. Appending a character creates a new ROPE.

You should use specific subranges for numeric variables whenever possible. If you don't know the range, use INT (32 – bit integer), unless you know you don't need that big a number and know you need efficiency. In those cases use INTEGER or NAT = [0..77777B]. Avoid using CARDINALs or LONG CARDINALs; their main use is in dealing with STRINGs. The compiler recognizes the abbreviation INT for LONG INTEGER, BOOL for BOOLEAN, CHAR for CHARACTER, and PROC for PROCEDURE.

Change all references to I/O packages of all kinds (streams, files, TTY) to use equivalent IO routines. IO is the only interface you should need to know about for I/O of almost any type of variable or constant (ROPE, INT, etc.) to almost any type of device (keyboard, display, files, temporary buffer etc.). IO contains:

A set of CreateX routines for each kind of stream X▮file, display, etc.

A set of GetX routines for each type X (integers, ropes, etc.)

A PutF routine that can be used with any type (integers, ropes, etc.) via a set of inline procedures (int, rope, etc.) which are used to tag the type of the arguments. It also provides a format argument which may be used to get FORTRAN – style formatting of output. For example, the format "%g" prints almost anything in default free – format:

```
stream.PutF["The sum of %g and %g is %g.\n", int[x], int[y], int[x + y] ]
```

A PutFR routine that is identical to PutF except it produces a rope as output instead of putting its result on a stream, and a RS routine that makes a rope look like a stream so that the GetX procedures can be used. Thus one can convert various types to and from ropes, e.g. the following code which converts an integer to a rope and back:

```
r: ROPE _ PutFR[, int[i]];
    j: INT _ GetInt[RS[r]];
```

Make use of LISTs and SEQUENCEs instead of ARRAYs and DESCRIPTORs for ARRAYs. The interface List contains some useful routines.

New language features

The changes in the Cedar language from Mesa 6 are fairly easy to understand for simple programs:

(a) REFs provide automatic deallocation and easier allocation:

```
Node: TYPE = REF Rec;
    Rec: TYPE = RECORD[first: INTEGER, rest: Node];
    . . .
    x: Node _ NEW[Rec _ [5, NIL]];
```

(b) Runtime types via REF ANY give looser binding:

```
TNode: REF B1Rec;
    Node: REF B2Rec;
    x: Node _ . . . ;
    t: TNode _ . . . ;
    q: REF ANY;
    . . .
    q _ t; q _ x;  - - both of these are legal
    t _ NARROW[q];  - - raises NarrowRefFault if q is not a TNode
    - - q^ _ E is always illegal. You cannot update through a REF ANY.
    . . .
- - type can also be checked explicitly:
        WITH q SELECT FROM
            m: TNode = > {t _ m; q _ m.lson};
            n: Node = > {x _ n; q _ n.rest};
            ELSE ERROR;
- - or
    IF ISTYPE[q, TNode] THEN {t _ NARROW[q]; q _ t.lson}
        ELSE IF ISTYPE[q, Node] THEN {x _ NARROW[q]; q _ x.rest}
        ELSE ERROR
```

REF ANY is preferred to the use of variant records.

(c) Lists are built into the language:

```
Node: TYPE = LIST OF INT;
    x: Node _ CONS[5, NIL];
    y: Node _ LIST[5, 6];  - - same as CONS[5, CONS[6, NIL]]
    i: INT _ y.first;  - - i is 5
    z:Node _ y.rest;  - - z is CONS[6, NIL]
    FOR l: Node _ y, l.rest UNTIL l = NIL DO . . .
```

(d) ROPEs, ATOMs, SEQUENCEs, and INTs are also built – in.

(e) To protect yourself and the garbage collector from obscure errors you should program in the safe subset of the language. To get a program into the safe subset prefix each module (PROGRAM, MONITOR, or DEFINITIONS) with the word CEDAR. The compiler will then tell you when you are straying outside the safe subset. You can wave the compiler off any block by placing the word TRUSTED before it. If you call a procedure declared in an unsafe interface (i.e., one that doesn't start with CEDAR

DEFINITIONS), the compiler will complain unless the call is in a TRUSTED block. Most of the high – level interfaces in the Cedar system are now safe.

Restrictions of the safe language

The @ operator is not permitted. There are three general ways to cope with this restriction: specializing, copying, and indirecting. For example, suppose you have a program that says

```
W: ARRAY [0..100) OF Z;
   P[@W];
   FOR i IN [0..100) DO . . . Q[@W[i]] . . . ENDLOOP;
```

To eliminate the first @ by specializing we would make a copy of the procedure P that dealt with the W directly▮not very satisfactory. To eliminate the first @ by copying we would pass the array W in by value and back by result▮also not very satisfactory. It is best to deal with the first @ by indirecting; just allocate W from collectable storage, writing

```
W: REF ARRAY [0..100) OF Z = NEW[ARRAY [0..100) OF Z];
   P[W];
```

Eliminating the second @ by specialization is plausible if Q knows it is always dealing with array elements: pass a reference to W along with an index. Otherwise, deciding between copying and indirecting depends upon the size of a Z. If it is small copy it, writing "W[i]__Q[W[i]]". If it is big create references to it and pass those, writing

```
W: REF ARRAY [0..100) OF REF Z;
   P[W];
   FOR i IN [0..100) DO . . . Q[W[i]] . . . ENDLOOP;
```

The form of variant record discrimination that does not copy the value to a new location cannot be used. Suppose you have a variant – record data structure like

```
T: TYPE = REF TR;
   TR: TYPE = RECORD[SELECT t:* FROM
      name, string => [x: ROPE];
      link => [i: INT, r: T];
      ENDCASE];
```

and are accustomed to performing discriminations like

```
e: T;
   WITH x: e^ SELECT FROM
      name, string => "Statements using x";
      link => {S1[x.i]; S2[@x]};
      ENDCASE;
```

You should declare a set of REFs to bound variant types like

```
Name: TYPE = REF name TR;
   String: TYPE = REF string TR;
   Link: TYPE = REF link TR;
```

and rewrite the discrimination to be

```
WITH e SELECT FROM
   x: Name => "Statements using x";
   x: String => "Statements using x";
   x: Link => {S1[x.i]; S2[x]};
   ENDCASE;
```

The type of x is now a REF type, not a TR, so various other types need to be adjusted and the @ in S2 is no longer needed. If "Statements using x" is a large block, you will probably want to introduce a procedure to avoid copying it.

Variant records cannot be overwritten. Similiar techniques can be used for sanitizing a program that overwrites variant records. Assuming the declarations of T and TR from above, suppose you wanted to write

```
x: T ← NEW[TR ← [name["END"]];
x^ ← [link[5, x]];
```

The specialization/copying technique is to simply update the thing that points at the record, writing "x ← NEW[TR ← [link[5, x]]]". However, if you don't know all the places that point at the record, you must introduce another level of indirection, writing

```
T: TYPE = REF REF TR;
x: T ← NEW[REF TR ← NEW[TR ← [rope["END"]]];
x^ ← NEW[TR ← [link[5, x]]];
```

Unsafe procedures cannot be passed as arguments to safe ones. The symptom of a violation of this rule is generally a message complaining about an incorrect type when there is no obvious type mismatch. All procedure types in an interface prefixed by CEDAR are implicitly prefixed with SAFE. The simplest thing to do is to put SAFE in front of PROC in the argument procedure declaration, and put TRUSTED in front of its body. As with all uses of TRUSTED, you should verify that the safety invariants are actually maintained, and document the reason for the TRUSTED in a comment.

For More Information . . .

Cedar Language Syntax

This is a one – page reference grammar describing the complete syntax of the Cedar Language, in a compact variation on BNF developed by Butler Lampson. Keep it handy as you write programs. It provides a relatively compact source of information on the exact form of constructs accepted by the compiler. It will also alert you to much of the available variety in the language▮but of course, not every syntactically valid program makes semantic sense.

The parsing grammar used by the compiler is somewhat larger and more complex than the Reference Grammar. Some of this is for technical reasons associated with LALR(1) parsing, and some of it to enable the compiler to make certain semantic distinctions while parsing. The differences should be invisible when dealing with correct programs, but may affect the error messages given for incorrect ones.

Annotated Cedar Examples

This document contains four complete, runnable Cedar programs chosen to illustrate the use of most of the major features of the language, and to provide an introduction to the style of programming that is preferred in Cedar. You should certainly invest time in studying them before attempting to write Cedar programs. If you are one of those who learns best from examples, you may find them virtually the only tutorial information you need to learn the language.

These examples have been chosen so that they are also useful prototypes of kinds of programs you may want to write in Cedar. If you are like most Cedar programmers, you will probably find it easier to start from such a prototype, and change it to do what you want, than to enter a whole program "from scratch."

Stylizing Cedar Programs

Because Cedar programmers so frequently read each other's code, it is considered good citizenship to adhere to certain stylistic conventions. Stylizing Cedar Programs discusses the generally agreed conventions.

You can save yourself a lot of typing, and produce nicely formatted code at the same time, by using Tioga's abbreviation expansion mechanism to generate all the high – level structure of your program (at least, all the bits that aren't simply copied). The file Cedar.abbreviations lists the available macros and their expansions; you can add your own favorites.

Cedar Program Style Sheet

This is an annotated prototype that you will probably want to keep close to hand, because it compactly illustrates the most important principles from the previous document.

Cedar Language Reference Manual

Eventually, this is intended to be a precise definition of the complete syntax and semantics of the Cedar Language. It is still incomplete.

The formal definition of the language is given in terms of a kernel language, into which all Cedar constructs can be desugared to determine their precise semantics. The Reference Manual contains both the definition of the kernel, and an explanation of the desugarings. It also contains several tables that collect important information about the primitive types and type constructors of Cedar.

Cedar Language Reference Summary Sheets

This is intended to be the essence of the entire Cedar Language carefully condensed into two pages for ready reference. It covers both syntax and semantics, with examples and notes. It is definitely not for those with weak eyes, and should probably not even be read until you have studied the Reference Manual proper. But it should be very helpful in checking details that you may have forgotten. Keep it handy.

Cedar Catalog

Since so much Cedar programming is done "at the component level," you need to know what packages and tools are available and what they do. In general, full documentation (or at least the best available approximation thereto) for each component is stored on [Indigo]<Cedar5.2>Documentation>, or is referenced in the component's DF file, stored on [Indigo]<Cedar5.2>Top>.

The problem is finding out which components you should be interested in. That's where the Cedar Catalog comes in handy. It contains a somewhat structured list of all the components in Cedar considered "interesting" by their maintainers. A component may be interesting

because of what it provides (your program may become a client),

because of what it does (you personally may become a user), or

because of how it does it (you may study it or copy some part of it in your program).

For each entry, the Catalog indicates why it is considered interesting, and how to acquire documentation and the component itself. It also identifies the maintainer, who is the ultimate

source of advice and help.

Mesa 5.0 Manual

The Mesa Language Manual, Version 5.0, PARC Technical Report CSL – 79 – 3, is the most recent self – contained manual on the Mesa Language. It falls somewhere between a tutorial and a reference manual, and many users have complained that it isn't entirely satisfactory for either purpose. But if you need more information about the Mesa – like parts of Cedar, it may be your best source.

Chapter 4 gives the details of Mesa's basic control constructs.

Chapter 5 tells all about procedures.

Chapter 7 goes into more detail than you probably want about the fine points of modules, programs, and configurations. You may be better off extrapolating from the Annotated Cedar Examples.

Chapter 8 gives some of the gory details of exceptions and exception handling. It is easy to get in trouble unless you use them in straightforward ways.

Chapter 10 provides a pretty reasonable discussion of how to make effective use of processes, monitors, condition variables, etc.

Who to see

If you haven't managed to find information that you want after you have looked in what you consider to be the obvious places (or if you don't understand what you have found), don't hesitate to ask. Almost anyone in CSL is a fount of wisdom, willing to be asked almost any question on almost any subject. (Of course, the answers aren't equally reliable, but you can't have everything.) If the first person you ask doesn't know the answer, chances are good that you'll get a pointer to either a person or document that will have the answer. More specifically here are some good people to ask:

Russ Atkinson	BugBane, runtime system, general questions
Bob Hagmann	VM, Alpine, general questions
Rick Cattell	Cypress, Squirrel, Walnut
Willie – Sue Orr	Dorado microcode, Walnut, device heads
Jim Donahue	Walnut, Squirrel, Cypress, Alpine
Doug Wyatt	Viewers, Tioga, Graphics
Mike Plass	Viewers, Tioga, TSetter
Howard Sturgis	Cedar on DLions