Filed on: [Indigo]<CedarDocs>Language>Concepts.doc, press
Last written: May 29, 1982 5:11 pm
By: Jim Horning

DRATF XXX DARFT XXX DRFAT XXX DRAFTT XXX DRAFT!

Cedar Language Overview Version 3

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, Jim Mitchell, Ed Satterthwaite, Chuck Geschke, and Dick Sweet. Subsequently, the OPD-OSBU System Development Department assumed responsibility for development and maintenance. It has gone through a series of releases; the most recent is Mesa 8.0.

When CSL launched the Cedar Project in 1979, it was decided to use the Mesa language and system as a starting point. (Mesa 7.0 is its closest relative.) However, Mesa did not have some of the features that were believed 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 safety in either case.

This Overview is intended to introduce a competent programmer who knows some other language in the Pascal family 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 have already read the Briefing Blurb and Getting Started in 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. The Cedar Language Reference Manual defines the Cedar Language by means of *desugarings* to a *kernel language*; we briefly introduce its central concepts and terminology. Finally, there is a guide to sources of further information.

Most parts of the Cedar 3 documentation, certainly including the language documentation, are still in an interim form. Comments and suggestions on how it can be made more useful are welcome at any time. Although we intend to survey student users at the end of the summer to assess the effectiveness of the various kinds and pieces of documentation, you need not wait until then to let us know what you think.

Various proposals and descriptions of interim implementations from September 1979 onward have been given labels such as 5C1, 5C2, 6C2, 6C5, and 7T11. **Version 3 of the Cedar language documentation is intended to supersede all descriptions prior to June 1982. 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; wherever possible, still-valid material has simply been lifted from previous documents.

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.

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*, which 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 *identifier* of a constant or variable, or by an *expression*, which will itself contain other values. Every identifier occurring in the program must be introduced by a *declaration*. A declaration associates with an identifier both a data type and a constant value (which may itself be a variable, and *contain* a non-constant value).

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 using it, or it may be referenced by a type identifier, 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 identifiers 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 identifiers 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 of the structured type.

In an *array strucure*, 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 an identifier 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 identifiers. 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.

Explicitly declared variables are called *static*. The declaration associates an identifier with the variable, which may be used to refer to it in expressions. Variables may also be generated by executable statements. Such *dynamic* variable generation yields a *pointer* or *reference* value that subsequently serves to refer to the variable, in place of an identifier. Because dynamically generated variables may occur as values of components of structured values contained by variables that are themselves dynamically generated, finite graphs in their full generality may be represented using pointers or references.

Statements

The most fundamental 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* of a designated procedure value to the values of its *arguments*.

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 identifiers so declared have significance only within the block. Hence, the block is the *scope* of these identifiers, 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 an identifier 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 includes a form of *module* that allows separate compilation of program units without sacrificing strong typing; 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 modules define interfaces to abstractions. They typically declare some shared types, useful constants, and the argument and result types of a set of procedure identifiers. They compile into *symbol tables*, which are shared by both *clients* and *implementations*. Checks are performed when modules are *bound* into a system to ensure that separately compiled pieces have used consistent versions of the shared definitions. DEFINITIONS modules produce no executable code and "exist" at run-time only in the sense that their symbol tables are accessible (e.g., for debugging).

PROGRAM modules provide implementations of abstractions. They typically declare collections of variables that define their local 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 run-time, one or more *instances* of a PROGRAM module 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 module instances (unlike those of procedure applications) are not restricted to follow any particular discipline. Communication paths among modules are established dynamically and are not constrained by any (static or dynamic) nesting relationships; lifetimes and access paths are completely decoupled.

A module that relies on declarations from a DEFINITIONS module (e.g., using a type or calling a procedure in its interface) must explicitly *import* it. This makes its symbol table available when the importing module is compiled. Unless the importing module specifies that it SHARES the interface, only the PUBLIC identifiers are accessible to it. If a program 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 the implementations of items declared PUBLIC are type-consistent with the declarations in the exported interface(s). Modules that import or export other modules must include DIRECTORY statements associating their Mesa identifiers with file names.

Importation introduces dependencies into the compilation order of modules. Since it needs their symbol tables, each module must be compiled *after* the modules it imports (and recompiled if they change). But information does not flow in the other direction. PROGRAM modules are imported only in very unusual circumstances; those that are not may be freely recompiled without invalidating previous compilation and checking of any other modules.

A PROGRAM 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. Binding a group of modules together into a system involves assigning values from the export records to the corresponding fields in the interface records.

Types, as well as procedures, can be declared in DEFINITIONS modules and subsequently bound to concrete values supplied by PROGRAM modules. 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 DEFINITIONS module 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 is considered to have 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 intervals, even while their implementations are undergoing change.

A very common approach is to define, comment, and circulate for review, all of the interfaces in a (sub)system before writing any of the PROGRAM modules. DEFINITIONS modules 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 does not contain many 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). If an interface Xyz is implemented by a single module, the PROGRAM module is named XyzImpl.

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 program that *raises* them to another, dynamically-selected 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 procedure calls, 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. 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 SIGNAL s, which may be resumed, and ERROR s, which may not; in common parlance they are generally all called signals.

Since handlers may take parameters and return results, each exception *type* must be declared in a scope that includes both all points where it is raised and all 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."

Although its language proposals have not yet been implemented, [Indigo]<CedarDocs>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. The language should enforce this, but it currently doesn't.

Processes

Mesa provides 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 process synchronization and mutually exclusive access to resources by means of entry to *monitors* and waiting on *condition variables.*

The FORK pseudo-procedure makes it possible to start the execution of another procedure concurrently with the program that applies it. It returns a result of a process type, which is declared similarly to a procedure type, except that only the type of the result is specified. There is no rule against multiple coexisting instances of a procedure, either forked or called, although care must be taken to ensure appropriate discipline on accesses to shared global data.

The JOIN pseudo-procedure takes a single argument of process type. 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.

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 which perform the accesses.

A MONITOR is a module instance. It thus has its own data in its global frame, and its own procedures for accessing that data. Some of the procedures are public, allowing calls into the monitor 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. A call into a monitor (to an *entry procedure*) automatically acquires its lock (waiting if necessary), and a return releases it. The lock makes it possible for the programmer to ensure the integrity of the monitor's global data (i.e., that it satisfies a *monitor invariant*) simply by making sure 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 the condition true, some other process must perform a NOTIFY on the condition variable; this allows a waiting process to reacquire the lock and resume execution. Note that the monitor invariant must be restored before any WAIT, since it releases the lock.

The procedures of a monitor are classified as ENTRY, INTERNAL, and EXTERNAL. INTERNAL procedures may only be called 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 call any internal procedures; they are often used to provide a convenient interface that "hides" one or more calls on ENTRY procedures.

The attributes ENTRY and INTERNAL are associated with a procedure's body, not with its type; thus they do not appear in DEFINITIONS modules. 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 data, 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.

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. Selection can also be based on the type of a variant record (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. In addition to ordinary statements, the body may contain EXIT or GOTO statements to *forcibly* terminate its execution, and may be followed by an EXITS 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 a result, the RETURN statement may supply the value to be returned (which is otherwise 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 import 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 this record and the object around together.

Procedure constants may be declared in DEFINITIONS modules or locally. Unlike procedure variables, they may have the INLINE attribute. This is an instruction to the compiler to expand the body inline for each application, rather than compiling a call to out-of-line code. This attribute is intended to improve the speed without changing the semantics of the procedure (although it may do so in obscure cases, e.g., exception handling). It should be considered a form of tight binding best reserved for late stages of system tuning.

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 somewhat lower cost, but they are not currently much used in Mesa or Cedar.

Miscellaneous

Every expression in a Mesa program has a type that can be deduced by static analysis of the program text. Such analysis is 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. In principle, every identifier and every expression has an *inherent type* derived from its structure. The inherent type of an identifier is established by declaration; the form of a literal implies its type, and 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 exact type required by the context is known, and a given expression must conform to it. The required type is called the *target type*. Examples include 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 inherent 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 objects, 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.

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 calls; default fields of records make the corresponding constructors more concise and more convenient; default values are also 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 vast quantities 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 REF s

Although Mesa POINTER s 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 variable it refers to *after* it has become inaccessible.

Cedar provides three types of storage:

Frame: This is storage that is implicitly allocated by procedure application and module instantiation to hold variables declared in the corresponding scope. It is also implicitly deallocated, upon exit from the scope.

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 points 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.

POINTERs may not be used in CHECKED regions, and may 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 is *implementation independence*, which means that the effects of each program are explicable in terms of the *language* rather than its implementation even if the program is erroneous. Mesa comes rather close to meeting this goal (as evidenced by the fact that most Mesa programs can be debugged "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 to reduce implementation dependencies in Cedar on general grounds. 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 was 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. Cedar does not provide acceptably efficient substitutes for every use of Mesa's unsafe features, but it 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 run-time system. Collectively, the properties that must be maintained to ensure invulnerability will be 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, POINTER (but REF is SAFE), JOIN, @ (address of), 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, and the block itself implements a SAFE operation.

Even UNCHECKED regions are supposed to maintain the safety invariants, but the guarantee must be provided by the programmer, rather than the system. If a block is UNCHECKED, UNSAFE operations may be used internally, and the block itself is considered to implement an UNSAFE operation. 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 program consists entirely of safe regions (and the invariants holds initially), then by induction the program 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.

Type confusion

Mesa is a strongly typed language, which means that the types of variables are declared, and that the language imposes restrctions 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 declared 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 mistyping of some piece of 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 an address-containing (AC) 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., lengthening an INTEGER to a LONG INTEGER). In general, conversion in the other direction requires a run-time 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 error).

The built-in test ISTYPE can be applied to a value to determine whether it can be narrowed to a specified type without error.

Implicit narrowing is supplied automatically if the target type is uniquely determined by context. An explicit narrowing may cause an implicit widening of its argument; the combination of narrowing applied to widening is never supplied automatically.

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, 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 static checkability and (often) runtime efficiency.

Experience with languages like Lisp and Smalltalk, in which most binding is done dynamically, shows that, if type and/or implementation binding can be deferred, it is much easier to write certain kinds of programs, e.g., programming tools (debuggers, performance monitors) and knowledge representation systems. But most programs take advantage of this flexibility only occasionally. 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 string and array descriptors allow deferring the specification of an object’s length until it is allocated. Otherwise, all types must be fixed at compile time. This makes it virtually impossible to avoid LOOPHOLE s 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 checking at compile time. (Note that it adopts the view that an object’s type ius inherent in the object itself, rather than in the way the object is referred to.)

TYPE is a type in the Cedar Language. Its "structuring methods" (e.g., ARRAY , RECORD , and REF) are now viewed as operators that take TYPE arguments and return TYPE s. In the current language, the arguments to such operators must be compile-time constants, but it is likely that this restriction will be removed in the future.

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 would not be statically knowable. The SELECT 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 either its formal parameter record type or result record type. PROC values with specific parameter and result types may be widened to these dynamic types, and later tested and narrowed analogously to REF ANYs. The APPLY operator can be used to apply a dynamically typed PROC value to a dynamically typed argument record.

TVs and RTTs XXX.

Miscellaneous

Although Cedar was not intended as a research project in programming languages, its developers were not completely 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.

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.

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. 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 identity.

Dot notation for object-oriented programming

xxx

Interface records

xxx

Clusters

xxx

Kernel Language Concepts And Terminology

Group

xxx

Type predicate

xxx

Cluster

xxx

Mark

xxx

Name

xxx

Binding

xxx

Desugaring as language definition mechanism

xxx

For More Information . . .

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 provide an introduction to the style of programming that is common 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 chose 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."

Reference grammar

It seems traditional that the syntax of every programming language is defined using its own notation. Cedar is no exception. The reference grammar is written in BLOVOBNF (Butler Lampson's own variation on BNF). It provides a relatively compact source of information on the exact form of constructs accepted by the compiler, and will also allert you to the full variety of the language.

You should be warned that 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.

Cedar Style

Because Cedar programmers so frequently read each other's code, it is considered good citizenship to adhere to certain stylistic conventions. This document discusses the generally agreed conventions, and provides an annotated prototype that you will probably want to keep close to hand.

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 Cedar Style document also contains a list of the standard Cedar abbreviations and their expansions.

Reference Manual

Eventually, this is intended to be a full-fledged precise definition of the complete syntax and semantics of the Cedar Language. What exists at present, however, is just the essence, two pages of carefully condensed material describing the syntax and semantics of the entire language, with examples and notes. It is definitely not for those with weak eyes, and should probably not even be read until you have gotten a fair acquaintance with the language from other sources.

Cedar Catalog: packages and tools

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 approximation thereto that is available) for each component is stored on [Indigo]<Cedar>Documentation>.

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 whose developers consider "interesting." A component may be interesting because of what it provides, or because of how it does it. In the former case, you may wish to include it in your program, in the latter you may want to study it, or even take it as a prototype for your program.

For each entry, the Catalog indicates why it is interesting, and how to acquire documentation or the component itself. It also identifies the implementor, who is the ultimate source of advice and help.

Mesa 5.0 Manual

The *Mesa Language Manual, Version 5.0*, Technical Report CSL-79-3, is the most recent 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. However, you may be better off extrapolating from the examples in the Cedar Language documentation.

It is easy to get in trouble with signals unless you use them in straightforward ways. Chapter 8 gives some of the gory details.

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.

The ultimate authority on exactly what the Cedar compiler will accept, and what code it will produce is Ed Satterthwaite. Roy Levin knows the Cedar runtime system inside out; Paul Rovner is the runtime-type system expert; and Russ Atkinson can explain how to use BugBane to debug those obscure problems.