Inter-Office Memorandum

To	Cedar Users	Date	February 15, 1982
From	Ed Satterthwaite and Jim Donahue	Location	Palo Alto
Subject	Cedar 7T11 Language and Compiler Changes	Organization	CSL

XEROX

Filed on: [Indigo]<CedarLang>Doc>Cedar7T11.Bravo

DRAFT

The document [Indigo]<CedarDocs>Lang>Cedar6T5.press describes the Cedar language. This memo summarizes the significant changes to the language and compiler since that document was prepared.

Types in Cedar

This section sketches some current thinking about the Cedar type system and might help you to understand the motivation for some of the changes described below. (See also Lampson, *Cedar abstract machine* [CedarAM.memo, February 1980].)

Types as Predicates

Every type is characterized by some predicate; a value *x* has type *T* iff *x* satisfies the predicate for *T*. In general, such predicates are defined in terms of a set of marks (tags, etc.) carried by each value; however, the Mesa type system is designed so that most mark manipulation can be done statically (by the compiler), and the usual representations of most values do not include explicit marks.

A given expression has some fixed *syntactic type* that depends upon the form of the expression and the declared types of constituent identifiers. The value denoted by an expression always satisfies the predicate characterizing its syntactic type, but such a value will often satisfy predicates characterizing other types as well. In this sense, a Cedar value may have an arbitrary number of types. For example:

> If *Thing* is a variant record type with a variant *red*, a reference to a *Thing* might simultaneously satisfy the predicates for REF ANY, REF *Thing*, and REF *Thing*[*red*] (formerly REF *red Thing*, see below).
>
> An opaque type and the corresponding concrete type are distinct, even within an exporter of the concrete type, but the predicates for the two types are identical.

Roughly speaking, the primary job of the predicates associated with types is to provide correct answers to questions about low-level representational conventions so that, e.g., the Cedar garbage collector can operate correctly.

The form ISTYPE[*x*, *T*] returns the result of applying the predicate characterizing *T* to the value *x*. In Cedar 7T11, ISTYPE has been redefined to work in a somewhat more general and uniform way, and the operations of NARROWing and (type-based) SELECTion have been defined in terms of ISTYPE.

Types as Clusters of Operations

In addition to its predicate, a *cluster* of operations (sometimes called a *group*) can be associated with a type. The main purposes of this grouping are to provide a number of packaging conveniences and to support so-called "object oriented" notation. If *x* has (syntactic) type *T*, *x*.*Op*[*args*] means *Op*[*x*, *args*] where *Op* is found by looking in the cluster associated with *T*. Two types may be characterized by the same predicate but have different associated clusters; in current Cedar, this is true of, e.g., an opaque type and the corresponding concrete type.

Each of the type constructors in Cedar supplies a standard and implicitly defined cluster for each type that it constructs. The only mechanism currently available for the explicit construction of such a cluster is the interface module, and previous versions of Cedar have limited support of this mechanism to opaque types. If *T* is an opaque type declared by, e.g.,

T: TYPE;

in some interface *Defs*, operations (procedures) declared in *Defs* become components of the cluster associated with *T* and may be invoked using object notation. Cedar 7T11 extends this support to allow construction of similar clusters for record types. If *T* is declared in *Defs* by

T: TYPE = RECORD [...];

the operations declared in *Defs* become part of the cluster associated with *T*. In this case, however, they augment the operations already supplied for *T* by the RECORD type constructor.

Defining clusters in this way has some drawbacks. The use of interfaces as the units of grouping somewhat overloads the existing notion of an interface; note that all operations declared in an interface become parts of the clusters of all types declared in that interface. Also, requiring a type and the operations in its cluster to be defined in the same interface occasionally conflicts with other criteria for partitioning interfaces. On the other hand, this method of defining clusters seems to cover the important cases well enough to be acceptable in practice. In addition, there is a fairly well worked-out plan for supporting clusters in a comprehensive, uniform way and for using them to explain parts of the Cedar abstract machine. We therefore recommend the following style guidelines for your Cedar programming:

Partition interfaces so that a single interface defines both a main type *T* (record or opaque) and all the operations to be provided in the cluster of *T* (or REF *T*). Define multiple main types within an interface only if the sets of meaningful operation names for those types are disjoint.

Use object notation in clients of interfaces designed to support it; i.e., use *x*.*Op*[*args*] in preference to *Defs*.*Op*[*x*, *args*].

(For Humus veterans) Avoid interface designs that require clients to write *x*.*Op*[*x*, *args*], *x*.*ops*.*Op*[*x*, *args*] or the like. Use an inline definition of *Op* within *Defs* to achieve such an effect.

LANGUAGE CHANGES

Syntax for Discriminated Types

If *V* is a type expression designating some variant record type with variant *a*, *V*[*a*] is a type expression designating the discriminated type. Thus forms such as

Object[*red*]   *Object*[*red*][*short*]   *Object*[*red*][*long*][*80*]

are equivalent to the old forms

red Object   *short red Object*   *long red Object*[*80*].

In Cedar 7T11, both forms are acceptable.

Dynamically Typed Procedures

Cedar 7T11 provides dynamically typed procedures as well as dynamically typed references. If *U* and *V* are (possibly empty) field lists, then the type PROC [*U*] RETURNS [*V*] conforms to any of the following types:

PROC ANY RETURNS ANY
PROC [*U*] RETURNS ANY
PROC ANY RETURNS [*V*]

Furthermore, the last two types also conform to the first. Similar types are available for declaration and manipulation of dynamically typed signals or errors.

In all three cases, the corresponding values are dynamically typed, and you must narrow or disciminate such a value before using it to invoke any kind of transfer operation. The forms available for doing this exactly parallel the forms available for REF ANY discrimination and are described in the next section. Note, however, that discrimination of procedure values is significantly more expensive than discrimination of reference values as implemented in 7T11.

Type Discrimination

Cedar 7T11 unifies the mechanisms for discriminating variant records with those for discriminating values with types REF ANY, PROC [*U*] RETURNS ANY, PROC ANY RETURNS [*V*], or PROC ANY RETURNS ANY. This unification affects the operators ISTYPE and NARROW as well as discriminating selection.

Type Testing

The primitive function ISTYPE tests whether a given value satisfies the predicate characterizing a specified type. You will probably have little direct use for ISTYPE; its importance lies in its use to define other, more common operations as described below. Let *x* be an expression with syntactic type *S*. In Cedar 7T11, the value of ISTYPE[*x*, *T*] is determined as follows, where *V* is any variant record type:

(1) It is TRUE (at compile time) if

S and *T* are equivalent types; or
S is an opaque type and *T* is the corresponding concrete type; or
S is a concrete type exported as the opaque type *T*.

The last two cases are recognized only within program modules that export the concrete type.

(2) It is determined dynamically by a test of the value *x*, yielding TRUE or FALSE, if

S is REF ANY and *T* is REF *U* for any *U* except ANY; or

S is PROC ANY RETURNS ANY, PROC ANY RETURNS [*U*] or PROC [*V*] RETURNS ANY and *T* is PROC [*U*] RETURNS [*V*] for any *U* and *V* except ANY; or

S is equivalent to *V* and *T* is equivalent to *V*[*a*]; or
S is equivalent to REF *V* and *T* is equivalent to REF *V*[*a*]; or
S is equivalent to (LONG) POINTER TO *V* and *T* is equivalent to (LONG) POINTER TO *V*[*a*];

where *V*[*a*] is a particular variant of *V*, perhaps discriminated to several levels. Note that the result is TRUE if the value of *x* is NIL.

(3) In all other cases, ISTYPE is unimplemented and is treated as a compile-time error.

Note that ISTYPE cannot currently be used to test a value for membership in a subrange.

Narrowing

NARROW[*x*, *T*] allows a value *x* to be viewed as a value of type *T* and succeeds iff ISTYPE[*x*, *T*] is TRUE. More precisely, NARROW[*x*, *T*] has (syntactic) type *T*, and its value is given by

IF ISTYPE[*x*, *T*] THEN *x* ELSE ERROR <Error>

where <Error> is

RTTypesBasic.*NarrowRefFault*[*x*, CODE[*T*]]	if ISTYPE[*x*, REF ANY]
RTTypesBasic.*NarrowFault*[]	otherwise.

The following situations correspond to the three cases enumerated in the definition of ISTYPE above:

(1) NARROW[*x*, *T*] is guaranteed (at compile time) to succeed.
(2) NARROW[*x*, *T*] may succeed or fail at run time.
(3) NARROW[*x*, *T*] is unimplemented.

Case (2) arises only when the syntactic type of *x* is related to *T* in one of the ways described above for ISTYPE. In Cedar 7T11, case (3) is treated as a compile-time type error. Fine point: NARROW[*x*, *T*] is also considered a compile-time error if the only possible value of *x* yielding TRUE is NIL. Use *x* = NIL instead.

In case (1), NARROW is an identity operation but can be useful to change the (syntactic) type of *x* without using a LOOPHOLE or requiring any code to be executed. Example:

```
Defs: DEFINITIONS = {
  T: TYPE;
  R: TYPE = RECORD [g: REF T, ... ];
  Pn: PROC [r: REF R];
  ... }.

Impl: PROGRAM EXPORTS Defs = {
  T: PUBLIC TYPE = RECORD [n: NAT, ...];
  Pn: PUBLIC PROC [r: REF Defs.R] = {
    r.g.n _ 0;                    -- invalid; r.g^ is opaque, with no field selection operations
    NARROW[r.g, REF T].n _ 0;     -- valid (because Impl exports Defs)
    ...};
  }.
```

As before, NARROW[*x*, *T*] may be written as NARROW[*x*] when the target type *T* is implied by context.

Discriminating Selection

The syntactic form of WITH ... SELECT that is currently used for REF ANY discrimination has been extended to discriminate any value for which ISTYPE performs a dynamic test of that value (see case (2) in the discussion of ISTYPE). The form

```
WITH v SELECT FROM
  v1: T1 => s1;
  v2: T2 => s2;
  ...
  vn: Tn => sn;
  ENDCASE => se;
```

is, by definition, equivalent to

```
u: T = v;
IF u # NIL AND ISTYPE[u, T1] THEN {v1: T1 _ NARROW[u]; s1}
ELSE IF u # NIL AND ISTYPE[u, T2] THEN {v2: T2 _ NARROW[u]; s2}
  ...
ELSE IF u # NIL AND ISTYPE[u, Tn] THEN {vn: Tn _ NARROW[u]; sn}
ELSE se;
```

where *T* is the (syntactic) type of *v*. The tests against NIL are omitted if *T* does not have a NIL value.

Note that this form always copies the discriminated value. Thus

```
r: REF V;
. . .
WITH r SELECT FROM
  x: REF V[a] => { ... x ...};       -- x is a copy of r with type REF V[a]
  ...
  ENDCASE;
WITH r^ SELECT FROM
  x: V[a] => { ... x ...};       -- x is a copy of r^ with type V[a]
  ...
  ENDCASE;
```

Contrast these with the old form of variant record discrimination, which does not copy the discriminated value and reevaluates the discriminating expression each time that it is used:

```
WITH x: r SELECT FROM
  a => { ... x ... };      -- x is a synonym for r^ (but with syntactic type V[a])
  ...
  ENDCASE;
```

The new forms are easier to make type-safe, and you should use them whenever possible. Unfortunately, the old form is still required, at least outside the checked language, for dealing with computed variants and with pointers having non-standard dereferencing operations, such as the current relative pointers).

Interaction with Opaque Types

If *T* is any exported type, REF *T* must have the "standard" implementation of type discrimination. We impose this requirement in anticipation of making REF ANY discrimination work correctly with opaque types (it still doesn't in 7T11). As a consequence, discriminated variant record types cannot be exported as the concrete values of opaque types.

Object Notation

The form *x*.*Op*[*args*] is interpreted as *Defs*.*Op*[*x*, *args*] if the type of *x* is (REF | POINTER TO)* *T* for some opaque type *T* declared in an interface, the principal instance of which is *Defs*. In other words, all the operations defined in *Defs* become part of the cluster of the type *T*.

This convention applies within the corresponding DEFINITIONS module (for writing inlines, etc.) as well as within importers of such modules. This is only a notational extension; the bindings of implicitly imported values are determined as before.

The clustering mechanism has also been extended in Cedar so that all operations declared in an interface become components of the clusters of any record types defined in that interface. With this extension, *Op* can be inline in more interesting ways. In addition, you may now be able to use object notation more extensively to invoke operations in existing interfaces, many of which are written in terms of (concrete) record types.

Note that every operation declared in an interface module becomes part of the cluster of every (record or opaque) type declared in that interface. Although the type of a particular operation normally will make it a useful component of only one cluster, its name appears in every other cluster and potentially hides or precludes a more appropriate definition of that name for that cluster. You therefore should define more than one main type per interface only if the sets of meaningful operation names for those types are disjoint.

Other points to note when using this convention with record types include the following:

In determining the binding of *Op*, the field identifiers declared in *T* take precedence over the identifiers declared in the interface *Defs*.

A value *x* with a record type *T* having a single component can be coerced to a value with the type of that component. In the form *x.id*, the lookup of *id* considers first the field identifier of the single component, then identifiers declared in the interface defining *T*, and finally any interpretation given to *id* by applying the coercion. You abuse this feature at your own risk (but see the discussion of clusters above). Example:

```
Defs1: DEFINITIONS = {
    ...
   T1: TYPE = RECORD [f1: REF Defs2.T2];
   ...
   OpN: PROC [self: T1, ...];
   ...}.

Defs2: DEFINITIONS = {
   ...
   T2: TYPE = RECORD [ ... ];
   ...
   OpM: PROC [self: REF T2, ...];
   OpN: PROC [self: REF T2, ...];
   ...}

 r1: Defs1.T1;
 r2: REF Defs2.T2;

 ... r1.OpN[...]  means Defs1.OpN[r1, ...]          -- from the cluster defined by Defs1
 ... r1.OpM[...]  means Defs2.OpM[r1.f1, ...]       -- from the cluster defined by Defs2 (after coercion)
 ... r2.OpN[...]  means Defs2.OpN[r2, ...]
 ... r1.f1.OpN[...] means Defs2.OpN[r1.f1, ...]     -- dubious style
```

Predeclared Types

To support the currently recommended Cedar standards, the types *BOOL*, *INT* and *CHAR* are predeclared, with the following definitions:

> *BOOL:* TYPE = *BOOLEAN*;
> *CHAR:* TYPE = *CHARACTER*;
> *INT:* TYPE = LONG *INTEGER*;

Also, the definition of the predeclared type *CONDITION* has been changed. The default value for the timeout interval now is effectively infinite; i.e., a WAIT on a condition variable with default initialization will never time out. (The previous default provided a timeout after 100 ticks.) Use a runtime procedure such as *Process*.*SetTimeout* to change the default setting.

Numeric Types and Conversions

Cedar 7T11 provides automatic conversion from types LONG *INTEGER* and LONG *CARDINAL* to types *INTEGER*, *CARDINAL* and any subranges thereof. If you request bounds checking, any loss of information in such a conversion causes the signal *Runtime*.*BoundsFault* to be raised.

In addition, you may use types such as *INT* that are equivalent to LONG *INTEGER* or LONG *CARDINAL* to define subrange types; however, any subrange type *T* must satisfy the constraints that

> $-2^{15} \leq$ FIRST[T] $< 2^{15}$ and |LAST[T] - FIRST[T]| $\leq 2^{16}-1$.

Furthermore, any numeric index type used to define an array or sequence type must satisfy the same constraints.

Rope Literals

The Cedar language now provides rope literals. Such a literal is denoted by a quoted string, e.g., "This is a rope literal". Its value is a reference to a rope object in the standard (counted) zone provided by the Cedar system.

The target type established by the context in which a quoted string literal appears determines the interpretation of that literal. There are three cases:

> If the target type is *Rope*.*ROPE*, *Rope*.*Ref* or *Rope*.*Text*, the quoted string denotes a rope literal and has type *Rope*.*Ref*.
>
> If the target type is any other REF type, the literal has type REF TEXT.
>
> Otherwise, the literal has type STRING.

In the first case, the test is actually for equivalence between the target type and either REF *Rope*.*RopeRep* or REF *Rope*.*TextRep*. The matching is performed on the names of the interface (*Rope*) and referent type (*RopeRep* or *TextRep*), not on the structure of the referent type. Since this is a loophole in the type checking, use nonstandard versions of the *Rope* interface very cautiously.

Escape Convention for Literals

Cedar provides an escape convention to allow denotations of nonprinting characters in character and string literals (cf. the escape convention for the language C). The escape character is \, and the following codes are recognized:

Code	Interpretation	
\n, \N, \r, \R	*Ascii.CR*	
\t, \T	*Ascii.TAB*	
\b, \B	*Ascii.BS*	
\f, \F	*Ascii.FF*	
\l, \L	*Ascii.LF*	-- note that \n = LF in C
\ddd	*ddd*C	-- where *d* is an octal digit, *ddd* < 377B
\\\\	\\	
\’	’	
\"	"	

Anything else following a \ is an error.

You can use the escape convention in character literals (e.g., ’\n or ’\032) or string literals (e.g., "abc\ndef").

APPLY and RETURN

Cedar is based upon a model of interprocedural control transfer in which the construction of an argument record is clearly separated from the actual transfer of control. In the usual forms for specifying call or return, however, these operations are syntactically indivisible. There are now alternative syntactic forms that allow you to invoke transfer operations using already constructed argument records.

This extension is not fully general. The existing record must have a type compatible with the type required by the transfer operation, and the only types compatible with argument record types are other argument record types. Such types are defined implicitly by the definitions of transfer types, and they are always anonymous. Thus you cannot declare variables having such types, nor can you construct values with such types unless the target type is established by a transfer operation of some sort.

The operator APPLY is used to apply a value with some transfer type to an argument record. The syntactic form is

Call	::=	...
	|	APPLY [**Expression** , **Expression**]
	|	APPLY [**Expression** , **Expression** ! **CatchSeries**]

The type of the first **Expression** must be some transfer type (i.e., a type built using PROC, SIGNAL, ERROR, PROCESS, PORT or PROGRAM), and the second **Expression** must have a record type as good as the argument type required for the transfer (see below). The effect is to invoke the transfer operation appropriate to the type of the first **Expression**, i.e., to call a procedure, raise a signal, join a process, etc. The scope of the optional catch phrase is just the transfer itself.

Note that the first **Expression** implies a target type for the second, which can be (but normally would not be) a constructor. For example,

p[*x*, *y*]	can be written as	APPLY[*p*, [*x*, *y*]]	
q[*x*]	can be written as	APPLY[*q*, [*x*]]	-- not APPLY[*q*, *x*]

The corresponding forms for returning an existing record are

ReturnStmt	::=	...
	|	RETURN **Call**
	|	RETURN (**Expression**)
ResumeStmt	::=	...
	|	RESUME **Call**
	|	RESUME (**Expression**)

In these forms, the required type is established by the context in which the statement appears. The type of the **Call** or **Expression** must be a record type as good as the result type of the procedure body in which the **ReturnStmt** appears (or of the catch phrase in which the **ResumeStmt** appears).

An argument record type T_1 is as good as an argument record type T_2 if both of the following conditions are satisfied:

T_1 and T_2 have the same number of fields, say *n*.

For each *i*, $1 < i < n$, the type of the *i*-th component of T_1 is as good as the type of the *i*-th component of T_2; in addition, if both these components are named, the names are identical (i.e., names of field selectors must match, but an anonymous component matches any named component).

Note that this rule is more liberal than the rule for explicitly declared record types.

In the terminology of the Mesa 5 manual, T_1 is as good as T_2 iff T_1 conforms freely to T_2; e.g., [0..10) is as good as [0..100). In the new view of types, we would say that T_1 is as good as T_2 iff the predicate for T_1 implies the predicate for T_2.

In Cedar 7T11, the constructs described above do not work for empty argument records; i.e., you cannot nest applications of procedures taking/returning nothing.

Examples:

```
P1: PROC [x, y: INT] RETURNS [m, n: INT] = {...};

P2: PROC [m, n: INT] RETURNS [u, v: INT] = {...};

P3: PROC [a, b: INT] RETURNS [u, v: INT] = {
  RETURN APPLY[P2, P1[a, b]]};

i, j: INT;
. . .
[i, j] _ APPLY[P2, IF i < j THEN P1[i, j] ELSE [j, i]];
[i, j] _ APPLY[P3, [0, 0] ! s => {GOTO L}];          -- [i, j] _ P3[0, 0 ! s => {GOTO L}]
```

Safe Subset Checking

The Cedar 6T5 document defines a "safe subset" of Cedar in which even incorrect programs cannot interfere with the reliable operation of the garbage collector. To quote from Cedar 6T5,

> "We intend that the vast majority of programs will be written primarily in the safe subset. However, we recognize that is it not presently feasible to provide acceptably efficient substitutes for all uses of Mesa's unsafe features. Instead, we will provide textual means for indicating that some regions of a program are *unchecked*. This will inhibit compiler enforcement of the safety restrictions and simultaneously indicate that the programmer has assumed the additional responsibility of ensuring that these regions of the program cannot violate the integrity of the system."

In Cedar 7T11, operations are classified as SAFE or UNSAFE. The precise description of SAFE operations is given below. A region of program text, bracketed to form a block, may be prefixed with one of the keywords CHECKED, TRUSTED or UNCHECKED. If a block is CHECKED, then within that block only safe operations may be used, and the block itself implements a SAFE operation. If it is UNCHECKED, unsafe operations may be used internally, and the block itself is considered UNSAFE. A TRUSTED block may also invoke unsafe operations, but it is assumed to present an external interface that is SAFE. Thus TRUSTED is an assertion by the programmer that a region using unsafe operations nevertheless preserves the Cedar system invariants as described in Cedar 6T5. Such an assertion cannot be checked by the compiler, and TRUSTED is a restricted form of LOOPHOLE.

Safe Operations

An operation is safe if it is guaranteed not to produce a result that would allow violation of the integrity of the system, including the code and the various data structures maintained by the present runtime system. In particular, the type integrity of the system must be preserved; in the presence of a garbage collector, this means that all operations that reference storage through paths unknown to the garbage collector must be regarded as potentially unsafe. Thus, the following primitive Cedar operations are not allowed to appear in CHECKED regions:

@ and the functions DESCRIPTOR and BASE that manipulate array descriptors,

^ or FREE applied to a value of type POINTER or LONG POINTER, and all pointer arithmetic,

variant discrimination that does not copy the discriminated value or a reference to it (see above),

JOIN or port call.

In addition, any use of LOOPHOLE to produce a value with an RC type is prohibited. Finally, assignment is restricted within CHECKED regions by:

restricting the type UNSPECIFIED to conform only to itself so that, e.g., you cannot assign a value of type UNSPECIFIED to any variable of type other than UNSPECIFIED,

prohibiting assignment of locally declared procedures to any procedure variable and prohibiting passing such procedures as arguments in FORKed calls (thus guaranteeing that procedure values cannot contain references to frames that may have been reclaimed).

prohibiting assignments that change the tags of variant records.

The transfer types of Cedar have been extended to allow the programmer to specify that a particular procedure or signal or process or program is SAFE or UNSAFE; inside CHECKED regions only transfer values with SAFE types can be used. Moreover, the bodies that are associated with SAFE procedures, processes or signals must themselves be safe. There are two ways to achieve this: one is to declare the body (either in the declaration of a procedure or the catch phrase for a signal) to be CHECKED, thus guaranteeing that the compiler will enforce the necessary safety; the other is to declare the body to be TRUSTED, in which case the body is UNCHECKED and the programmer takes upon himself the obligation to guarantee that the necessary system invariants are preserved. Thus, the following combinations are allowed:

```
P: PROGRAM =
  CHECKED{ F1: SAFE PROC[] RETURNS[ INTEGER ] = { RETURN[ 1 ] };
                F2: SAFE PROC[] RETURNS[ INTEGER ] = TRUSTED{ RETURN[ LOOPHOLE[ 'A ] ] };
                F3: UNSAFE PROC[] RETURNS[ INTEGER ] = UNCHECKED{ RETURN[ LOOPHOLE[ 'A ] ] };
                [] _ F1[]; [] _ F2[]; UNCHECKED{ [] _ F3[] } }.
```

ERRORs are treated differently from SIGNALs in this regard. SIGNALs (like procedures) may be declared as either SAFE or UNSAFE; if a SIGNAL is SAFE, then all of the catch phrases associated with it must either be CHECKED or TRUSTED. ERRORs on the other hand are not considered SAFE or UNSAFE; a catch phrase associated with an ERROR must be CHECKED or TRUSTED if the surrounding scope is CHECKED, i.e., the body of the error must behave like a SAFE procedure (that may or may not be called) if it is used in a CHECKED region. The same rule applies to catch phrases for UNWIND. (This treatment of ERROR is consistent with a termination semantics for ERRORs; we have adopted this view of safety in anticipation of adopting the semantics in the near future.)

Defaults

One of the major problems in introducing the safe Cedar subset is the use of old (potentially) unsafe interfaces in new (CHECKED) programs. Thus, we have adopted the following rules for setting defaults: If the module begins with *ModuleId:* CEDAR . . ., then the outermost block is to be CHECKED and all interfaces are assumed to be SAFE; otherwise, the outermost block is to be UNCHECKED and all interfaces are assumed to be UNSAFE.

Unless you explicilty use one of the keywords SAFE or UNSAFE, the safety of each transfer type that you define is determined by the default established in the module header. On the other hand, the checking attribute is inherited. By default, a nested block is CHECKED if the textually enclosing block is CHECKED; otherwise, it is UNCHECKED.

COMPILER CHANGES

Version Stamps

In its intermodule type checking, Cedar uses so-called version stamps to identify independently compiled modules. The version stamps computed by the Cedar 7T11 compiler are functions of the identity of that compiler and of its inputs. You can now recompile the same source file, with the same included modules, the same compiler and the same switch settings to get an object file with the same version stamp.

This stamp, which is essentially a 48 bit hash, is computed recursively as follows. Assume that any existing derived object (including the compiler itself) has a version stamp. The stamp for a new derived object is a hash of

the creation time of the source file
the version stamp of each bcd mentioned in the DIRECTORY clause
the version stamp of the compiler
the compiler switches (with those controlling only compile-time feedback masked off)

There is also a 7T11 binder that computes version stamps for its output in the same way.

Note

In the past, the version stamp has been a concatenation of a machine identifier and the creation time of the derived object. Many existing utility programs therefore print the version stamp formatted as a machine and network number, a date and a time. These programs give strange-looking output but, in all cases known to us, perform correctly.

Tioga Source Files

The compiler and binder ignore text in Tioga trailers. Any occurrence of a pair of NUL characters (characters with value 0C) in a source file marks the logical end of that source file.

File Locking

The Cedar 7T11 compiler is designed to be run under control of the system modeller. It also exports an interface allowing it to be run from Tajo or from the temporary Cedar executive. When it is run in this mode, the (rather minimal) facilities in PreCascade for obtaining exclusive access to a file are bypassed. Use caution.

Compiler Switches

The Cedar compiler is no longer able to generate object code for an Alto (or D-machine emulating an Alto). The switch `/a` is ignored. The switch `/l` has a new interpretation; if it is set (the default), the compiler leaves space for code links in the BCD that it produces.

There is a Cedar switch `/c`; if it is set (the default), the code for FORK assumes the availability of the Cedar runtime. If you plan to run your program directly under Pilot, compile with `/-c`. If you are in doubt about how your processes will interact with the Cedar runtime, consult a wizard.