The ClassAd Language Reference Manual Version 2.2

Marvin Solomon
Computer Sciences Department
University of Wisconsin—Madison
solomon@cs.wisc.edu

October, 2004

1 Overview

This document provides a formal specification of the syntax and semantics of the *ClassAd* (Classified Advertisement) language. The name of the language derives from its origin in the Condor distributed computing system [7, 8], where it is used for discovery and allocation of resources. Providers of computing resources submit advertisements describing their capabilities and declaring constraints and preferences for jobs they are willing to run. Consumers submit ads describing their jobs. Like the provider ads, these ads may impose constraints and express preferences regarding execution sites. A *matchmaker* process matches the producer and consumer ads. All policy considerations are contained in the ads themselves; the matchmaker merely supplies the mechanism for interpreting the policies expressed in the ClassAd language. The present document only defines the language. This introduction hints at how the language can be used to express a variety of scheduling policies, but a full discussion of this issue is beyond the scope of this document.

The ClassAd language is a functional language. The basic unit is the *expression*, and execution entails *evaluation* of expressions, replacing an expression with its *normal form* or *value*. There are no side-effects: Evaluation has no effect other than calculating the value of an expression. The language is carefully designed to allow efficient evaluation. In particular, an expression can be evaluated in time proportional to the size of the expression.[1]

The most important type of expression is a *record expression* (sometimes called a "classad"), which is a set of name/value pairs. The "value" in each pair may be an arbitrarily complex expression, including nested record expressions and lists of expressions. An example is

```
[
    type = "gizmo";
    components = {
        [ type = "widget"; part_number = 12394 ],
        [ type = "widget"; part_number = 92348 ]
    };
```

[1] Actually, this statement is only true if each function call completes within a time bound proportional to the length of its arguments. All "built-in" functions have this property.

```
    main_component = components[0];
    main_part = main_component.part_number;
]
```

ClassAd expressions are strongly but dynamically typed. Supported types include integer and floating-point numbers, Boolean values (**true** and **false**), character strings, timestamps, and time intervals. During the evaluation, invalid sub-expressions evaluate to the value **error**. For example, `1/0`, `3 * "abc"`, `{1, 2, 3}[5]`, and `27[5]` all evaluate to **error**. Attribute references (occurrences of identifiers on the right-hand side of attribute definitions) are replaced by their definitions. Attribute names with no definition or circular definitions evaluate to **undefined**. Attribute lookup is "block structured": An attribute reference is resolved by searching all record expressions containing the reference, from innermost outward, for a matching definition (this matching is case-insensitive). For example, the expression

```
[
    a = 1; b = c;
    d = [ f = g; i = a; j = c; k = l; a = 2; ]
    l = d.k; c = 3;
]
```

evaluates to

```
[
    a = 1; b = 3;
    d = [ f = undefined; i = 2; j = 3; k = undefined; a = 2; ]
    l = undefined; c = 3
]
```

There are currently two implementations of the ClassAd language, one in C++ and one in Java. Both implementations should have identical external behavior conforming to this specification.[2] In both implementations, ClassAd expressions are tree-structured objects. Each implementation provides methods for constructing expressions, navigating through them, and evaluating them, as well as methods for translating between the internal data structures and character strings using either of two *concrete syntaxes*, the "native representation" used in the examples above and XML [4].

An important application of ClassAds is *matchmaking*. Matchmaking is applied to a pair A, B of record expressions, each of which is expected to have a "top-level" definition of the attribute `Requirements`. The `Requirements` attribute of A is evaluated in an environment in which the attribute reference `other` evaluates to B, and `B.Requirements` is evaluated in an environment in which `other` evaluates to A, If both `Requirements` attributes evaluate to the specific value **true** (not **undefined**, **error**, or a value of some non-Boolean type), the expressions A and B are said to *match*. In Condor, matchmaking is used to match *job* and *machine* ads. A machine ad describes the characteristics and current state of a machine. It also has a `Requirements` attribute to restrict the set of jobs it is willing to run and a `Rank` attribute to indicate how much it "likes" an individual job. `Rank` should evaluate to a non-negative integer, with higher values indicating preferred jobs. Similarly, a job ad has descriptive attributes, a `Requirements` attribute to constrain the set of acceptable machines, and a `Rank` attribute to express preferences. The Condor matchmaker uses these attributes to match jobs with machines.

[2] Of course, this is an idealized goal. As of this writing, there are several ways in which the two implementations differ from each other and from this specification. An effort is underway to improve conformance.

2 Basic Concepts

An expression is either *atomic*, consisting of a *literal constant* or *attribute reference*, or it is a *composite* expression, composed of an *operator* applied to one or more *operands*. A literal constant is an *integer*, *real*, *string*, *absolute time*, or *relative time* literal, or one of the values **true**, **false**, **error**, or **undefined**. An attribute reference has an associated character string, such as `size` or `n123`, called an *attribute name*. Operators include the familiar arithmetic, logical, and comparison operators of C, C++, or Java (such as `+`, `&&`, and `<=`), calls to functions (such as `substring(x, 1, 3)` or `member(x,l)`), operators for constructing lists: $\{expr_1, ..., expr_n\}$, and records: $[name_1 = expr_1; ...; name_n = expr_n]$, the subscript operator: $expr_1[expr_2]$, and the attribute-selection operator: *expr.name*.

3 Syntax

The ClassAd language has both an *abstract* syntax and several *concrete* syntaxes. The abstract syntax describes each expression as an abstract data structure called its *internal form*. Each concrete syntax maps an expression to a string of ASCII characters called its *external form* or *representation*. There are currently three concrete syntaxes defined: the *native* syntax, the *old classad* syntax, and the *xml* syntax. Each implementation should provide an application programming-language interface (API) for manipulating internal-form expressions as data structures in the host programming language and converting between representations: *parsing* to convert from external to internal form, and *unparsing* to convert from internal to external form.

Figure 1 shows a sample expression in abstract, native, and XML syntaxes.

3.1 Abstract Syntax

An internal-form expression is an ordered tree: Each node is either a *leaf* or an *internal node* with a sequence of *child* nodes. There is a unique *root* node; every other node has a unique *parent*. The leaves of an expression tree are *literal constants* and *references*. Each internal node contains an *operator*. Its children are called *operands*.

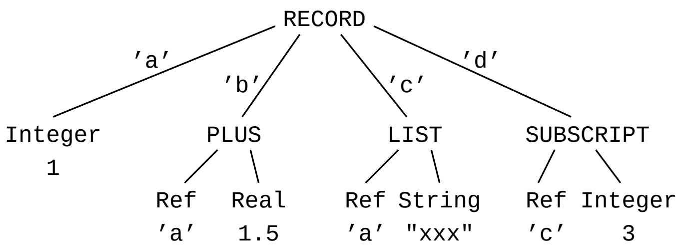
A literal constant is an *integer*, *real*, *string*, *boolean*, *absolute time*, *relative time*, *error*, or *undefined*. All literal constants have *values* as indicated in Table 1. The value of an error literal is the unique value **error** of type Error. An error literal also has an *annotation*, which is a sequence of zero or more non-null ASCII characters meant for human consumption. Similarly, an undefined literal has the value **undefined** and an annotation. Two literals are *equivalent* if and only if they have the same value.

A reference contains an *attribute name*, which is a sequence of zero or more non-null characters, or the reserved word `parent`. Two attribute names *match* if the character sequences are the same length and corresponding characters are the same except for differences in case. The following words are *reserved*, meaning that they may not be used as attribute names.

[3] Character values are currently limited to the range 1 – 255 (decimal). Future versions of this specification may use Unicode for representing attribute names and string literals.

[4] This document does not specify the precision or range of values supported by absolute time literals. The Java implementation currently stores the time as a signed, 64-bit offset in milliseconds from the epoch, while the C++ implementation uses an signed 32-bit offset in seconds (though this may be platform-dependent). Both implementations use an epoch of 00:00, January 1, 1970, Universal Coordinated Time. The time zone is recorded as an offset from UTC.

[5] This document does not specify the precision or range of values supported by relative time literals. The Java implementation currently stores relative times as signed, 64-bit integer indicating milliseconds, while the C++ implementation uses a signed 32-bit offset indicating seconds.



(a) Abstract Syntax

```
[ a = 1; b = a + 1.5; c = { a, "xxx" }; d = c[3] ]
```

(b) Native Syntax

```
<c><a name="a"><i>1</i></a>
    <a name="b"><e>a+1.5</e></a>
    <a name="c"><l><e>a<e><s>xxx</s></l></a>
    <a name="d"><e>c[3]</e></a></c>
```

(c) XML Syntax

Figure 1: Example ClassAd Expression

```
error false is isnt parent true undefined
```

Recognition of reserved words is independent of case. For example, `false`, `FALSE`, and `False` are all reserved words.

An internal node consists of an *operator* and a sequence of child nodes called its *operands*. Each operator has an *arity* constraining the number of operands it may have. Operators are classified as *unary* (one operand), *binary* (two operands), *ternary* (three operands), or *varargs* (any number of operands). The operators and their arities are listed in the Table 2. Table 2 also indicates the symbol used to represent each operator in the native syntax described in the following section.

With the exception of the binary operator SELECT and the varags operators RECORD and FUNCTION_CALL, the operands may be arbitrary expressions.[6] The second operand of the SELECT operator, called the *selector*, is an attribute name. The first operand of the FUNCTION_CALL operator, the *function name*, is a non-empty sequence of letters, digits, and underscores not starting with a digit; the remaining operands are called *actual parameters*. Section 4.3.9 lists a set of *built-in functions* that must be provided by all implementations.[7] The RECORD operator has zero or more operands, called *attribute definitions*. Each attribute definition is an ordered pair consisting of an attribute name and an arbitrary expression. The attribute names must be distinct (ignoring case)[8] We say that the attribute definition $N = E$ *defines* the

[6]The ClassAd language is a dynamically typed language. Mis-typed expressions such as `3/"foo"` are allowed by the syntax, but will evaluate to **error**.

[7]Other functions may be provided by an extension mechanism to be defined in a subsequent version of this specification.

[8]The set of attribute names in a RECORD expression is logically a set of distinct case-independent strings, but the original ordering and case may be retained by an implementation so that the "unparsing" of an parsed expression more closely resembles

Type	*Value or annotation*
Integer	32-bit signed, two's complement integer
Real	64-bit IEEE 754 double-precision floating point number
String	Zero or more non-null characters[3]
Boolean	The value **true** or **false**
AbsTime	An indication of an instant in time and the time zone where this time was measured[4]
RelTime	Interval between two absolute times[5]
Error	Zero or more non-null ASCII characters providing a human-readable explanation
Undefined	Zero or more non-null ASCII characters providing a human-readable explanation

Table 1: Literal Constants

attribute name N' if N and N' match.

3.2 Escaped Strings

Each of the concrete syntaxes represents an attribute name or string literal as a sequence of printable ASCII characters (characters with values in the range 32 – 126 inclusive).[9] The translation from concrete to abstract form (parsing) and from abstract to concrete form (unparsing) in each case shares the following two algorithms.

In the concrete representation, a backslash may be followed by one of the characters

```
b  t  n  f  r  "  '  \  0  1  2  3  4  5  6  7
```

In the value of the resulting String or attribute name, the backslash and one or more following characters are replaced by a single ASCII character as indicated by Table 3.

If the character following the backslash is an octal digit, the octal constant consists of three digits if the first digit is 0, 1, 2, or 3 and two digits otherwise. However, the constant is terminated by the first character that is not an octal digit. The string is ill-formed if a backslash is followed by any character other than those listed in Table 3, or if the value of an octal constant following a backslash is zero.

Multiple canonical representations may parse to the sequence of characters, any of which may be considered an "unparsing" of that sequence. The *canonical unparsing* of a sequence of characters is defined relative to an optional *delimiter* chararacter, which may be quote (34), apostrophe (39), or neither. Printable ASCII characters (in the range 32 – 126 inclusive) are represented as themselves, with the exception that each backslash (92) or delimiter character (if any) is preceded by an extra backslash. Each character with the value 8, 9, 10, 12, or 13 is replaced by a backslash followed by the character `b`, `t`, `n`, `f`, or `r`, respectively. Any other character is replaced by a backslash followed by a three-digit octal representation of its value.

For example, the strings

```
a'\n
a\'\n
```

the original.

[9] In this section, all characters codes are given in decimal.

<table>
<tr><th>Arity</th><th>Symbol</th><th>Name</th><th>Arity</th><th>Symbol</th><th>Name</th></tr>
<tr><td>1</td><td><code>+</code></td><td>UPLUS</td><td>2</td><td><code>>=</code></td><td>GREATER_EQ</td></tr>
<tr><td>1</td><td><code>-</code></td><td>UMINUS</td><td>2</td><td><code><<</code></td><td>LEFT_SHIFT</td></tr>
<tr><td>1</td><td><code>~</code></td><td>BIT_COMPLEMENT</td><td>2</td><td><code>>></code></td><td>RIGHT_SHIFT</td></tr>
<tr><td>1</td><td><code>!</code></td><td>NOT</td><td>2</td><td><code>>>></code></td><td>URIGHT_SHIFT</td></tr>
<tr><td>2</td><td><code>||</code></td><td>OR</td><td>2</td><td><code>+</code></td><td>PLUS</td></tr>
<tr><td>2</td><td><code>&&</code></td><td>AND</td><td>2</td><td><code>-</code></td><td>MINUS</td></tr>
<tr><td>2</td><td><code>|</code></td><td>BITOR</td><td>2</td><td><code>*</code></td><td>TIMES</td></tr>
<tr><td>2</td><td><code>^</code></td><td>BITXOR</td><td>2</td><td><code>/</code></td><td>DIV</td></tr>
<tr><td>2</td><td><code>&</code></td><td>BITAND</td><td>2</td><td><code>%</code></td><td>MOD</td></tr>
<tr><td>2</td><td><code>==</code></td><td>EQUAL</td><td>2</td><td><code>[]</code></td><td>SUBSCRIPT</td></tr>
<tr><td>2</td><td><code>!=</code></td><td>NOT_EQUAL</td><td>2</td><td><code>.</code></td><td>SELECT</td></tr>
<tr><td>2</td><td><code>is</code></td><td>SAME</td><td>3</td><td><code>?:</code></td><td>COND</td></tr>
<tr><td>2</td><td><code>isnt</code></td><td>DIFFERENT</td><td>var</td><td><code>{ ... }</code></td><td>LIST</td></tr>
<tr><td>2</td><td><code><</code></td><td>LESS</td><td>var</td><td><code>[...]</code></td><td>RECORD</td></tr>
<tr><td>2</td><td><code>></code></td><td>GREATER</td><td>var</td><td><code>name(...)</code></td><td>FUNCTION_CALL</td></tr>
<tr><td>2</td><td><code><=</code></td><td>LESS_EQ</td><td></td><td></td><td></td></tr>
</table>

Table 2: Operators

Escape Sequence	*Value (decimal)*
`\b`	8 (Backspace)
`\t`	9 (Horizontal Tab)
`\n`	10 (Linefeed)
`\f`	12 (Formfeed)
`\r`	13 (Carriage Return)
`\\`	92 (Backslash)
`\"`	34 (Quote)
`\'`	39 (Apostrophe)
octal_digit$^+$	*value of octal constant*

Table 3: Backslash escapes

```
a\47\012
\141\047\012
```

all parse to the three-character sequence consisting of a lower-case `a`, an apostrophe, and a newline (97, 39, 10 decimal). The canonical unparsing is the second of these if apostrophe is the delimiter, and the first otherwise.

3.3 Native Syntax.

The native syntax of the ClassAd language is quite similar to C, C++, or Java. The external representation of an expression consists of a sequence of *tokens* separated by optional white space and/or comments. Comments are as in C++ or Java: The string `//` introduces a comment that continues until the end of the current line, and `/*` introduces a comment that continues until the next occurrence of `*/` (such comments do not nest).

3.3.1 Tokens.

Tokens include *reserved words*, *integer*, *floating-point*, and *string literals*, *attribute names*, *operators*, and the additional lexical tokens equal sign (`=`), parentheses (`( )`), braces (`{ }`), brackets (`[ ]`), comma (`,`), and semicolon (`;`). The operators are shown in Table 2 in the previous section. The syntax of literals and attribute names is shown in Table 4.

```
token                   ::=  attribute_name | literal | operator | punctuation | reserved_word
attribute_name          ::=  unquoted_name | quoted_name
literal                 ::=  integer_literal
                              | floating_point_literal
                              | string_literal
punctuation             ::=  '=' | '(' | ')' | '{' | '}' | '[' | ']' | ',' | ';'
unquoted_name           ::=  ( letter | '_' ) ( letter | decimal_digit | '_' )*
quoted_name             ::=  '''( non_quote | '"' )+ '''
integer_literal         ::=  nonzero_digit decimal_digit*
                              | '0' octal_digit*
                              | '0' ( 'x' | 'X' ) hex_digit+
floating_point_literal  ::=  digit+ '.' digit* exponent?
                              | '.' digit+ exponent?
                              | digit+ exponent
exponent                ::=  ( 'e' | 'E' ) ( '+' | '-' )? decimal_digit+
string_literal          ::=  '"' ( non_quote | ''' )* '"'
letter                  ::=  'a'...'z' | 'A'...'Z'
non_quote               ::=  '\' escaped_char
                              | '\' octal_digit octal_digit?
                              | '\' ( '0'...'3' ) octal_digit octal_digit
                              | other_character
escaped_char            ::=  'n' | 't' | 'b' | 'r' | 'f' | '\' | '"' | '''
decimal_digit           ::=  '0'...'9'
nonzero_digit           ::=  '1'...'9'
octal_digit             ::=  '0'...'7'
hex_digit               ::=  decimal_digit | 'a'...'f' | 'A'...'F'
```

Table 4: Names and Literals

In Table 4, parentheses are used for grouping, literal characters are enclosed in single quotes, and the postfix meta-operators $*$, $+$, and ? are used to denote zero or more repetitions, one or more repetitions, and zero or one repetitions, respectively. The name `other_character` denotes any ASCII character other than Quote (34)[10] apostrophe (39), backslash (92), linefeed (10), carriage-return (13), or null (0). The name `reserved_word` denotes any of the strings

[10] In this section, all characters codes are given in decimal.

```
error false is isnt parent true undefined
```

in any combination of upper or lower case letters.

A ClassAd expression consists of a sequence of tokens, separated by whitespace. Whitespace consists of one or more occurrences of the ASCII characters Space (32), Horizontal Tab (9), Linefeed (10), Vertical Tab (11), Formfeed (12), or Carriage Return (13).[11] Whitespace is only required between two tokens if the first character of the second token could be construed as an extension of the first token according to the rules in Table 4. Two consecutive `string_literal`s separated by whitespace are equivalent to a single literal whose value is the concatenation of the values of the two literals.

A `string_literal` or `quoted_name` is parsed according to the common rules for Escaped Strings in Section 3.2. The `quoted_name` syntax allows an attribute name to be any sequence of non-null ASCII characters. If the sequence conforms to the syntax of `unquoted_name`, the attribute name may be represented using either syntax. For example, the following are three representations in the native syntax of the *same* attribute name:

```
_abc        '_abc'        '_ab\143'
```

3.3.2 Grammar.

The native syntax is defined by the context-free grammar in Table 5.

To simplify the grammar, productions enforcing the precedence of operators have been omitted. The actual precedences are indicated by the order in which operators are listed. Operators are listed in order of increasing precedence (most tightly binding last), with operators on the same line having equal precedence. All binary operators are left-associative. Unary operators have higher precedence than binary operators other than subscripting and attribute selection. The meta-syntactic symbols `()+?` are used as in the definition in the previous section. A complete YACC grammar for the ClassAd language is contained in Appendix A.

3.3.3 Unparsing.

Parsing associates with each *expression*, *binary_expression*, *prefix_expression*, or *suffix_expression* a node in the internal representation of an expression. For example, a *binary_expression* of the form $expr_1 + expr_2$ parses to an internal expression with operator PLUS and operands that are the results of parsing $expr_1$ and $expr_2$. Multiple distinct strings may parse to the same internal form, due to presence of optional comments, whitespace, and parentheses. To support certain built-in functions such as **string**, we define for each expression a *canonical unparsing*, which parses to that expression.

The canonical unparsing of an expression is completely parenthesized and has no comments and no whitespace outside of string literals or attribute names. For example, the strings "`-x + 3 * (y + 1)`" and "`((-x)+(3*(y+1)))`" both parse to the same internal form. The second string is the canonical unparsing.

The canonical unparsing of an attribute name is simply the name itself if it conforms to the syntax of *unquoted_name* in Table 4. Otherwise, an attribute name is unparsed using the *quoted_name* syntax. A string literal is unparsed as a *string*. Inside a *quoted_name* or *string*, the unparsing uses the rules defined in Section 3.2, with apostrophe as the delimiter for a *quoted_name* and quote as the delimter for a *string*.

[11] This definition of whitespace corresponds to the C and C++ programming languages. The Java Language Specification (Second Edition) defines whitespace similarly, but omits Vertical Tab. The latter definition also corresponds to the (deprecated) method `Character.isSpace` in the standard Java API. The API documentation recommends replacing `Character.isSpace` with `Character.isWhitespace`, which adds Vertical Tab as well as characters with codes 28–31 decimal (FS, GS, RS, and US) and several Unicode characters with codes greater than 256.

```
expression            ::= binary_expression
                          | binary_expression '?' expression ':' expression
binary_expression     ::= binary_expression binary_operator prefix_expression
                          | prefix_expression
binary_operator       ::= '||'
                          | '&&'
                          | '|'
                          | '^'
                          | '&'
                          | '==' | '!=' | 'is' | 'isnt'
                          | '<' | '>' | '<=' | '>='
                          | '<<' | '>>' | '>>>'
                          | '+' | '-'
                          | '*' | '/' | '%'
prefix_expression     ::= suffix_expression
                          | unary_operator suffix_expression
unary_operator        ::= '+' | '-' | '~' | '!'
suffix_expression     ::= atom
                          | suffix_expression '.' attribute_name
                          | suffix_expression '[' expression ']'
atom                  ::= attribute_name
                          | 'error' | 'false' | 'true' | 'undefined' | 'parent'
                          | integer_literal
                          | floating_point_literal
                          | string_literal+
                          | list_expression
                          | record_expression
                          | function_call
                          | '(' expression ')'
list_expression       ::= '{' ( expression ( ',' expression )* ','? )? '}'
record_expression     ::= '[' ( attribute_definition ( ';' attribute_definition )* ';'? )? ']'
attribute_definition  ::= attribute_name '=' expression
function_call         ::= unquoted_name '(' ( expression ( ',' expression )* )? ')'
```

Table 5: Native Expression Grammar

The literals **true**, **false**, **undefined**, and **error** unparse to `true`, `false`, `undefined`, and `error`, respectively. The annotation of an Error or Undefined literal is not included in the canonical unparsing.

The canonical unparsing of an integer literal is the decimal representation of its value, with no leading zeros unless the value is exactly zero and no sign unless the value is negative.

The canonical unparsing of a real literal is one of the strings `0.0`, `-0.0`, `real("INF")`, `real("-INF")`, or `real("NaN")`, or a normalized "scientific" representation such as `6.02E24` or `3.14159265E0`, with one non-zero digit before the decimal point and as many digits following the decimal point as necessary to represent the value exactly (but at least one digit must follow the decimal point). For details, see Section 3.5 on the

XML representation for details.

The canonical unparsing of an absolute time literal has the form `absTime("yyyy-mm-ddThh:mm:ss±zz:zz")` where the argument is a string representation of the time and date in ISO 8601 syntax, including the local time zone as hours and minutes from the prime meridian (negative for west). The canonical unparsing of a relative time has the form `relTime("d+hh:mm:ss.mmm")` where the argument is a string representation of the duration in days, hours, minutes, seconds, and milliseconds.[12] The suffix `.mmm` is omitted if the number of milliseconds is zero, and leading fields are omitted, together with the following punctuation, if their values are zero. Leading digits of the first non-zero field are also omitted. The string begins with a minus sign if the value is negative. Examples are `absTime("1949-03-11T08:17:00-06:00")`, `relTime("-5:00")`, and `relTime("0")`.

The canonical unparsing of a list or record expression omits the optional comma or semicolon following the last item. In particular, the empty record unparses as `[]` and the empty list unparses as `{}`.

3.4 Old ClassAd Syntax

The so-called "old" classad syntax is used in some versions of Condor to transmit classads (Record Expressions) over a network connection. It is deprecated. Not all expressions can be represented in the old syntax. To be so represented, an expression must satisfy the following conditions.[13]

- The operator at the root node of the tree is RECORD.
- The root node has two attribute definitions that define attribute names `MyType` and `TargetType` (ignoring case).
- All attribute names conform to the syntax of *unquoted_name* in Table 4.

If the root RECORD node has N children, the representation consists of 4 bytes representing the binary value N, most significant byte first, followed by N strings of ASCII characters, each terminated by a null character. The last two strings are representations, in the native syntax, of the expressions in the attribute definitions corresponding to attribute names `MyType` and `TargetType`, in that order. The remaining $N-2$ strings correspond to the other $N-2$ operands of the RECORD operator. Each of these strings has the form "*name* = *expr*", where *name* is the attribute name of the definition, and *expr* is a representation, using the native syntax, of the corresponding expression.

3.5 XML Syntax

A Document Type Definition (DTD) for the XML representation may be found in Appendix B, and a schema definition compliant with the XML Schema standarad [5] is in Appendix C.

In the following paragraphs, *xml_escape(string)* denotes a string of printing ASCII characters that results from the following two steps, excuted in order. First, the *string* is replaced by its canonical unparsing, as defined in Section 3.2. Then, each occurrence of `<`, `&`, or `>` is replaced by `<`, `&`, or `>`, respectively. The function *xml_escape_attr* is defined similarly, except that in the second step, `"` is also replaced by `"`.

[12] For this conversion, leap seconds and daylight savings time transitions are not taken into account. Every minute is 60 seconds, and every day is 24 hours.

[13] The "old"" classad language had many more restrictions than those listed here. An expression that meets these conditions can be represented in the Old syntax for transmission over a connection, but it may not be intelligible to a legacy application receiving the transmission.

The canonical XML representation of an expression depends on the operator at the root of the expression. All "simple" (atomic) values conform to types defined in *XML Schema Part 2: Datatypes* [3]. In this section, these types are referenced as if they were in the namespace xsd, e.g., xsd:string, where xmlns:xsd=http://www.w3.org/2001/XMLSchema. A complete XML Schema is provided in Appendix C.

A string literal is represented as <s>*xml_escape(s)*</s>, where s is the value of the literal. The content of an <s> element has type xsd:string.

An integer literal is represented as <i>*dddd*</i>, where *dddd* is the decimal representation of its value, with no excess leading zeros (that is, the first digit is zero if and only if the value is zero), preceded by a minus sign if the value is negative. The content of an <i> element has type xsd:int.

A real literal is represented as <r>*s*</r>, where s is the representation of the value of the literal in "scientific" notation, for example, <r>3.141592653589793E+00</r>. More specifically

- The values positive and negative infinity and not-a-number are represented by INF, -INF, and NaN, respectively.
- Any other value is represented by the concatenation of the the following strings, with no characters between them:
  - an optional leading minus sign (omitted if the value is positive),
  - a "mantissa" of the form d.f, where d is a digit and f is a sequence of 15 digits,
  - an upper-case letter E,
  - a plus or minus sign, and
  - an integer exponent of two or three digits.

  The first digit of the mantissa must be non-zero unless the value being represented is zero. If the exponent is three digits, the first digit must not be zero. Note that a sign is required in the exponent.

The content of an <i> element has type xsd:double. Except for the representations for positive and negative infinity and not-a-number, which conform to the specification of xsd:double, this representation is meant to conform to the printf format %1.15E.

A true or false boolean literal is represented by <b v="t"/> or <b v="f"/>, respectively.

An Error literal with non-empty annotation a is represented by the string <er a="*annot*"/>, where *annot* is *xml_escape_attr(a)*. If the annotation is empty, the representation is <er/>.

An Undefined literal with non-empty annotation a is represented by the string <un a="*annot*"/>, where *annot* is *xml_escape_attr(a)*. If the annotation is empty, the representation is <un/>.

An absolute time literal is represented as <at>yyyy-mm-ddThh:mm:ss±zz:zz</at>, where the element content is a string representation of the time and date in a restricted ISO 8601 syntax, including the local time zone as hours and minutes from the prime meridian (negative for west). The content of an <at> element has type xsd:dateTime. An example is <at>2003-01-25T09:00:00-06:00</at>.

A relative time literal is represented as <rt>sPnDTnHnMn.mmmS</rt>, where s is - for negative values and omitted for positive values, each n is a non-negative integer, representing days, hours, minutes, and seconds, mmm represents a number of milliseconds, and the letters PDTHMS are included verbatim. The suffixes represent scale factors: DT for days (24*60*60), H for hours (60*60), M for minutes (60) and S for seconds (1). A field (including its following scale factor) may be omitted if its value is zero. If the value is an integral number of seconds, the number of milliseconds, along with the preceding decimal point, may be omitted, but if present, it must be represented as exactly three digits. Finally, the letter T is omitted if the

H, M, and S fields are all omitted. Note that these rules permit multiple representation of the same value. For example, <rt>PT1H2S</rt>, <rt>PT60M2S</rt>, and <rt>PT3602.000S</rt> represent the same value. The *canonical* representation is defined by the limitation that the number of hours is less than 24, and the numbers of minutes and seconds are each less than 60. Moreover, each field (including milliseonds) is omitted if its value is zero. However as a special case, the value zero is represented as <rt>PT0S</rt>. Thus the first of the three examples above is the canonical representation of a duration of 3602 seconds. The content of an <rt> element has type xsd:duration.

If the root operator is LIST, the XML representation consists of the string <l> followed by the concatenation of the representations of its operands (in order), followed by the string </l>. The content of an <l> element is a sequence of elements of any the types listed in this section except <a>.

If the root operator is RECORD, the XML representation consists of the string <c> followed by the concatenation of the representations of its constituent definitions, followed by the string </c>. The representation of an attribute definition *name* = *expr* consists of the string followed by the XML representation of *expr* followed by the string . The content of a <c> element is a sequence of <a> elements. The content of an <a> element is an element of any of the types listed in this section except <a>.

In all other cases, the XML representation of an expression consists of <e>*xml_escape(s)*</e>, where *s* is the canonical unparsing of the expression in the native syntax. The content of an <e> element has type xsd:string.

Non-canonical XML representations may differ from the canonical representation in the following ways:

- Additional whitespace may be added to all elements other <s> or <r> elements, provided the results are syntactically valid. (For example, spaces may not be placed between the digits of an <i> element.)
- Additional whitespace may be added to tags, but not inside attributes in tags.
- The content of an <r> element may use an unnormalized representation (more or fewer digits before or after the decimal point), the E and the following exponent may be omitted (implying an exponent of zero), the decimal point may be omitted if the fractional part of the mantissa is empty, and any combination of upper and lower case letters may be used for the strings E, INF, and NaN.
- The content of an <at> element may be any string which is a valid argument to the **absTime** built-in function.
- The content of an <rt> element may be any string which is a valid argument to the **relTime** built-in function.
- Any expression *e* may be represented by <e>*xml_escape(s)* </e>, where *s* is any string that parses to *e* according to the native syntax.

For example,

```
<c><a n="the value"><e>b</e></a><a n="b"><r>3.14E0<\r></a><c>
<c>
    <a n="the value"> <e>   b</e> </a>
    <a         n="b"> <e>3.14</e> </a>
<c>
<e>[ 'the value' = b; b = 3.14 ]</e>
```

are three XML representations of the same expression; the first is the canonical representation.

4 Evaluation

This section defines the semantics of the ClassAd language by explaining how to *evaluate* an expression.In this section, "expression" means an internal-form expression tree. In general, a composite expression is evaluated by recursively evaluating its component sub-expressions and then using its top-level operator to combine the results. However, there are situations in which evaluation of an expression E depends on parts of a *context*, which is an expression containing E as a sub-expression. For example, in the expression

```
[ a = 3;  b = [ c = a ] ],
```

the second occurrence of `a` (an attribute reference) is evaluated by searching the two containing Record expressions for a definition of `a`, yielding the constant `3`.

More formally, an *expression in context* (EIC) is a pair (E, C) consisting of an expression C (the context) and a designated occurrence of a sub-expression E of C. The semantics of the ClassAd language is defined by a recursive function *eval* from EICs to EICs. A *top-level* EIC is an EIC of the form (E, E). For brevity, we will occasionally abbreviate the top-level EIC (E, E) as E, particularly when E is a literal constant. For example, the EIC (**error**, **error**) may be written as **error**. An expression E is evaluated by computing *eval(E, E)* and extracting the sub-expression from the resulting EIC.

The set of EICs with context C is partially ordered by the relation $\sqsubseteq$, defined by $(E, C) \sqsubseteq (E', C)$ iff E is a sub-expression of E'. When we speak of the "minimal" EIC with a given property, we mean the one that is minimal with respect to $\sqsubseteq$. An EIC (E, C) is called a *scope* if the top-level operator of E is RECORD.

Define *lookup(s, (E, C))*, where s is a string and (E, C) is an EIC, to be the EIC (E', C'), where

- If s matches[14] the string `"parent"` and there is a scope (E_p, C) such that $(E, C) \sqsubseteq (E_p, C)$, then $(E', C') = (E_p, C)$, where (E_p, C) is the minimal such scope.
- Otherwise, if there is a scope (E_d, C) such that $(E, C) \sqsubseteq (E_d, C)$ and E_d contains a definition $t = E_t$ such that t matches s, let $t = E_t$ be the definition in the minimal such scope. Then $(E', C') = (E_t, C)$.
- Otherwise, $(E', C') = (\textbf{undefined}, \textbf{undefined})$.

For example, let C be the expression

```
[ a = x;  b = [ a = y; c = a]; d = a ],
```

and let R denote the inner Record expression. C contains two occurrences of the attribute-reference expression `a`. Let E_1 denote the occurrence inside R and E_2 the other occurrence. Then $(E_1, C) \sqsubseteq (R, C) \sqsubseteq (C, C)$, $(E_2, C) \sqsubseteq (C, C)$, $\mathit{lookup}(\mathtt{a}, (E_1, C)) = (y, C)$, $\mathit{lookup}(\mathtt{a}, (E_2, C)) = (x, C)$, $\mathit{lookup}(\mathtt{c}, (E_1, C)) = (E_1, C)$, and $\mathit{lookup}(\mathtt{c}, (E_2, C)) = (\textbf{undefined}, \textbf{undefined})$.

4.1 Types, Undefined, and Error

Each expression has a *type*, which is one of Integer, Real, String, Boolean, AbsTime, RelTime, Undefined, Error, List, or Record. The types Integer and Real are collectively called numeric types. The types AbsTime and RelTime are collectively called timestamp types. Each operator imposes constraints on the types of its operands. If these constraints are not met, the value returned by the operator is **error**.

An attribute reference with attribute name N evaluates to **undefined** if the reference is not contained in any scope that defines N. It may also evaluate to **undefined** in the presence of loops, as in

[14] We are using the term "match" here as defined in Section 3.1: Two strings match if they are identical except for differences in case.

```
[ a = b; b = a ].
```

Most operators are "strict" with respect to **undefined** and **error**. The only exceptions are the Boolean operators described in Section 4.3.1, the operators **is** and **isnt** described in Section 4.3.2, and the LIST and RECORD constructors described in Section 4.3.8. Strict evaluation obeys the following ordered sequence of rules.

- If the operands do not obey the type restrictions imposed by the operator, the result of the evaluation is **error**. The following sections list all combinations of types accepted by each operator. None of the strict operators accept operands of type Error, so this rule implies that if any sub-expression evaluates to **error**, the expression evaluates to **error**. This rule also catches "type errors" such as `"foo" / 3`.
- Otherwise, if any operand of a strict operator is **undefined**, the result is **undefined**.
- Otherwise, the result is computed from the operands as described in the following sections.

4.2 Atomic Expressions

A literal constant evaluates to itself. More precisely, if c is an occurrence of a literal constant, then *eval(c, C) = (c, C)*.

If x is an attribute reference with attribute name N, then *eval(x, C) = eval(lookup(N, (x, C)))*. In particular, (x, C) evaluates to **undefined** if there is no scope (R, C) containing the indicated occurrence of x such that R defines N. If this recursive definition leads directly or indirectly to a call *eval(x, C)*, the result is **undefined**.

4.3 Composite Expressions

List and Record expressions evaluate to themselves. More precisely, if E is an expression whose root operator is LIST or RECORD, *eval(E, C) = (E, C)*. The operators SELECT and SUBSCRIPT are discussed below. For all other operators, evaluation is "bottom-up" and the result is a "pure value". More precisely, if $\odot$ is a binary operator other than SELECT, or SUBSCRIPT, then

$$eval(E_1 \odot E_2, C) = (c, c)$$

where

$$eval(E_1, C) = (E_1', C_1),$$
$$eval(E_2, C) = (E_2', C_2),$$

and c is the (literal constant) result of applying operator $\odot$ to the expressions E_1' and E_2', as defined in the following sections. Similar rules apply to unary and ternary operators.

The operators found in C, C++, or Java are generally evaluated according to the rules of those languages. In cases where the specifications of those languages differ, the ClassAd language follows the Java semantics because it is more precise (the C and C++ specifications occasionally say the results are "undefined" or "implementation defined" in unusual situations). The only deviations from Java semantics involve exceptions. In cases where Java specifies that evaluation throws an exception, the ClassAd language returns the constant **error**. The constants **error** and **undefined** also require special treatment when supplied as arguments to operators.

4.3.1 Boolean Operators

The Boolean operators `&&` and `||` and the ternary operator `_?_:_` are evaluated "left to right" with respect to **error**, and "optimistically" with respect to **undefined**. For example,

> **true** `||` x = **true**
> **false** && x = **false**
> **undefined** `||` **true** = **true**
> **true** `?` *val* `:` x = *val*
> **false** `?` x `:` *val* = *val*

even if x evaluates to **error** or **undefined**.

The Boolean operators treat Boolean **true**, **false**, and **undefined** as a three-element lattice with

> **false** `<` **undefined** `<` **true**.

With respect to this lattice, `&&` returns the minimum of its operands, `||` returns the maximum, and `!` interchanges **true** and **false**.

The complete definition of the operators `&&`, `&&`, `!`, and `_?_:_` is given by the tables

```
&& | F U T O    || | F U T O    ! |      ?:|
---+--------    ---+--------    --+--    ---+---
 F | F F F F     F | F U T E    F | T     F | expr3
 U | F U U E     U | U U T E    U | U     U |  U
 T | F U T E     T | T T T T    T | F     T | expr2
 O | E E E E     O | E E E E    O | E     O |  E
```

In these tables, the letters `T`, `F`, `U`, and `E` stand for the constants **true**, **false**, **undefined**, and **error**, respectively; `O` stands for any expression other than **true**, **false**, or **undefined** (including **error**); and `expr2` and `expr3` represent the second and third operands of the expression $expr_1$ `?` $expr_2$ `:` $expr_3$.

4.3.2 is and isnt

The expression $expr_1$ **is** $expr_2$ evaluates to **true** if $expr_1$ and $expr_2$ evaluate to "identical" values and **false** otherwise. The expression $expr_1$ **isnt** $expr_2$ evaluates to the negation of $expr_1$ **is** $expr_2$. These operators are most commonly used to test for **undefined** or **error** as in

```
result = (expr is undefined) ? 0 : (expr + 1);
```

but they can be used to compare arbitrary values.

For the purposes of this section, the relationship "identical" is defined as follows.

- Expressions of different types are never identical. Thus `(3 is 3.0) (3 is "3")` and `(undefined is error)` all evaluate to **false**.
- Two values of type Integer, Real, or RelTime are identical if and only if they are numerically equal constants.
- Two values of type AbsTime are identical if they represent the same instant in time and the same offset from UTC.

- Two values of type Boolean are identical if and only if they are both **true** or both **false**.
- Two values of type Undefined or two values of type Error are identical. Thus `((3 * "x") is error)` evaluates to **true**.
- Two values of type String are identical if and only if they are identical character by character. Case *is* significant. Thus `("One" == "one")` and `("One" isnt "one")` both evaluate to **true**.
- Two values of type List or Record are identical if and only if they are created by the same instance of a LIST or RECORD constructor operator. For example, if `R` is the expression

  ```
  [ a = { 1, 2 }; b = { 1, 2 }; c = a is b; d = a is a ],
  ```

  then `R.c` evaluates to `false`, while `R.d` evaluates to `true`.

Note that the **is** and **isnt** operators always evaluate to **true** or **false**, never **undefined** or **error**.

4.3.3 Comparison Operators

For the six comparison operators `<`, `<=`, `==`, `!=`, `>=`, and `>`, both operands must be numeric (Integer or Real), both String, both AbsTime, or both RelTime. Otherwise, the result is **error**. If one operand is Integer and the other is Real, the Integer argument is first converted to Real. The results are calculated as in Java [6].

If the operands are Strings, they are converted to lower case and compared lexicographically.

If the operands are AbsTimes, they are equal if they correspond to the same instant (according to UTC). Otherwise, the earlier time is less than the later one. If the operands are RelTimes, they are compared as signed integers.

4.3.4 Arithmetic Operators

The unary operators `+`, `-`, and binary operators `+`, `-`, `*`, `/`, `%`, take numeric operands.[15] The results are calculated as in Java [6],[16] with one exception: Integer division or remainder when the second operand is zero throws an ArithmeticException in Java, but returns **error** in the ClassAd language. In particular, if operands are Integers, the result is an Integer, and if one operand of a binary operation is an Integer and the other is a Real, the Integer operand is converted to a Real and the result is computed using 64-bit floating point arithmetic. The integral `/` operation truncates the result towards zero, and the integral `%` operation generally returns a result with the same sign as the dividend (the left operand). See the Java language specification [6] for details.

The unary and binary operators `+` and `-` are also defined for certain timestamp operands. The unary `+` operator is applicable to both AbsTime and RelTime operands and returns the value of its operand unchanged. The unary `-` operator is applicable only to RelTime operands and returns the RelTime value with the same magnitude and opposite sign.

The rules for binary operators are summarized in Table 6. If the result of an expression is an AbsTime, its time zone is the same as the time zone of the AbsTime argument.

[15] Unlike Java, the `+` operator is not overloaded to accept String operands.

[16] The `%` operator is defined for floating point operands according to the Java programming language specification, not according C. Some C implementations may not support `%` with floating point operands, so users concerned with portability should avoid this special case until all implementations are brought into compliance.

Expression	*Result type*	*Result value*
AbsTime + AbsTime	**error**	
AbsTime + RelTime	AbsTime	The AbsTime operand offset by the amount of the RelTime operand
RelTime + AbsTime	AbsTime	The AbsTime operand offset by the amount of the RelTime operand
RelTime + RelTime	RelTime	The numeric sum of the two operands
AbsTime - AbsTime	RelTime	The numeric difference of the two operands
AbsTime - RelTime	AbsTime	The AbsTime operand offset by the negative of the RelTime operand
RelTime - AbsTime	**error**	
RelTime - RelTime	RelTime	The numeric difference of the two operands

Table 6: Date and Time Arithmetic

4.3.5 Bitwise Boolean Operators

The bitwise logical unary operator `~` and binary operators `|`, `^`, and `&` are defined only for Integer and Boolean operands. They are defined to return the same results as the corresponding operators in Java [6].

4.3.6 Shift Operators

The shift operators `<<` (left shift), `>>` (right shift with sign extension), and `>>>` (right shift with zero fill) are defined only for Integer operands.[17] They are defined to return the same results as the corresponding operators in Java [6].

4.3.7 Select and Subscript

The SELECT operator has two operands, the *base* and the *selector*, where the selector is syntactically constrained to be an attribute name. In the native syntax, it is written *base*`.`*selector*. It is semantically equivalent to *base*`["`*selector*`"]`. That is, an instance of SUBSCRIPT operator where the *subscript* is the string value corresponding to the attribute name. For example,

```
[ rec = [ One = 1; Two = 2 ]; val = rec.one ].val
```

and

```
[ rec = [ One = 1; Two = 2 ]; val = rec["one"] ].val
```

both evaluate to `1`. The SELECT syntax is more concise, but the SUBSCRIPT syntax is more flexible, because it allows the selector to be computed rather than requiring a literal string.

The SUBSCRIPT operator has two operands, the *base* and the *subscript*. In the native syntax, it is written *base*`[`*subscript*`]`. The subscript expression must have type Integer or String. If the subscript is an Integer i, the base expression must have type List and the result is the i^{th} element of the list, counting from zero. If the subscript is a String s, the base expression must be a Record or List. If the base expression

[17] Note that C and C++ have no `>>>` operator. These languages perform a similar operation when the operands are declared to be unsigned. There are no unsigned types in Java or the ClassAd language.

has type Record, the result is computed by searching the base and its containing scopes for an attribute definition matching the attribute name s. If the base expression is a List, the SUBSCRIPT operator is applied to each member of the list and the result is a new "top-level" list of the results. In all other cases, the result is **error**.

More precisely,

$$eval(E_b[E_s], C) = (E', C'),$$

where E' and C' are defined as follows. Let

$$eval(E_b, C) = (E'_b, C'_b)$$

and

$$eval(E_s, C) = (E'_s, C'_s).$$

- If E'_s is an Integer with value i and E'_b is a list $\{E_0, ..., E_{n-1}\}$ with $n > i$ members, then $(E', C') = (E_i, C'_b)$.
- If E'_s is a String with value s and E'_b is a Record expression, then $(E', C') = eval(lookup(s, (E'_b, C'_b)))$.
- If C_2 is a String with value s and E'_b is a list $\{E_0, ..., E_{n-1}\}$, then E' is the list $\{E'_0, ..., E'_{n-1}\}$, where $eval(E_i[s], C'_b) = (E'_i, C'_i)$ and $C' = E'$ (that is, the result is a "top-level" EIC). Note that the environments C'_i returned by the recursive calls to $eval$ are discarded.
- In all other cases, the result is **error**.

4.3.8 List and Record Constructors

The LIST operator takes as operands an arbitrary sequence of values of arbitrary types. The RECORD operator takes as operands a sequence of definitions of the form $name_i = value_i$, where the $value_i$ are arbitrary values. The result is the Record

$$[name_0 = value_0;\ ...\ ;\ name_{n-1} = value_{n-1}]$$

List and Record expressions evaluate to themselves. That is, *eval(E, C) = (E, C)* if E is of type List or Record.

4.3.9 Function Calls

The FUNC_CALL operator takes a function name and zero or more operands. Function names are matched regardless of case, so that `substr("abc",2)`, `SubStr("abc",2)`, and `SUBSTR("abc",2)` all invoke the same function.

Currently, all functions are *strict* with respect to **error** and **undefined**, unless otherwise specified. In other words, if any argument evaluates to **error** or **undefined**, the result is **error** or **undefined**, respectively. If arguments of both types are present, the result is **error**.

Currently, all functions return "top-level" values that are independent of the the context of the call. That is $eval(f(E_1, ..., E_n), C) = (V, V)$, where $eval(E_i, C) = E'_i$ for $i = 1, ..., n$ and V is a value computed from $E'_1, ..., E'_n$ as described in the following table.

The following table lists all functions required by the current version of this specification; others may be added in future versions. The description of each function is preceded by a prototype indicating restrictions on the number and types of arguments and indicating the type of the result returned. If the restrictions are violated, the result is **error**. In the prototypes, "const" stands for any literal constant of type Integer, Real, String, Boolean, AbsTime, or RelTime (but not Undefined, Error, List, or Record), and "any" means any expression. A type followed by an asterisk indicates any number of arguments of the indicated type, including none. Square brackets are used to indicate optional arguments.

isUndefined(*any a*) returns *boolean.* Returns **true** is a is the undefined value, otherwise returns **false**. This function is not strict.

isError(*any a*) returns *boolean.* Returns **true** is a is the error value, otherwise returns **false**. This function is not strict.

isString(*any a*) returns *boolean.* Returns **true** is a is a string value, otherwise returns **false**. This function is not strict.

isInteger(*any a*) returns *boolean.* Returns **true** is a is an integer value, otherwise returns **false**. This function is not strict.

isReal(*any a*) returns *boolean.* Returns **true** is a is a real value, otherwise returns **false**. This function is not strict.

isList(*any a*) returns *boolean.* Returns **true** is a is a list value, otherwise returns **false**. This function is not strict.

isClassad(*any a*) returns *boolean.* Returns **true** is a is a record value, otherwise returns **false**. This function is not strict.

isBoolean(*any a*) returns *boolean.* Returns **true** is a is a boolean value, otherwise returns **false**. This function is not strict.

isAbstime(*any a*) returns *boolean.* Returns **true** is a is an AbsTime value, otherwise returns **false**. This function is not strict.

isReltime(*any a*) returns *boolean.* Returns **true** is a is a RelTime value, otherwise returns **false**. This function is not strict.

int(*const x*) returns *int.* The result is x converted to an Integer. If x is an Integer, the result is x. If x is a Real, it is truncated (towards zero) to an integer. If x is **true** the result is 1. If x is **false** the result is 0. If x is an AbsTime, it it converted to the number of seconds since the epoch, UTC. If x is a RelTime, it it converted to a number of seconds. If x is a String, it is parsed according to the native syntax for *integer_literal* or *floating_point_literal* as in Table 4 and then converted to an Integer as above. If x is a String that does not represent a valid integer or floating-point literal, the result is **error**.

real(*const x*) returns *real.* The result is x converted to a Real. If x is a Real, the result is x. If x is an Integer, it is converted to Real. If x is **true** the result is 1.0. If x is **false** the result is 0.0. If x is an AbsTime, it it converted to the number of seconds since the epoch, UTC. If x is a RelTime, it it converted to a number of seconds. If x is a String, it is parsed according to the native syntax for *integer_literal* or *floating_point_literal* as in Table 4 and then converted to a Real as above. In addition, the strings **INF**, **-INF** and **NaN** (in any combination of upper and lower case) are recognized as representing the IEEE754 values for positive and negative infinity and not-a-number, respectively. If x is a String that does not represent a valid integer or floating-point literal, the result is **error**. For any other type, x is converted to an Integer as if by "int", and the result is converted to a Real (or **error** if the conversion to Integer fails).

string(*any x*) returns *string.* If x is a String, the result is x. Otherwise, the result is the canonical unparsing of x (see Section 3.3.3).

floor(*const x*) returns *int.* If x is an Integer, the result is x. Otherwise, x is converted to a real by the function "real" above, and the result is the largest integer not greater than that value (or **error** if the conversion fails).

ceiling(*const x*) returns *int.* If x is an integer, the result is x. Otherwise, x is converted to a real by the function "real" above, and the result is the smallest integer not less than that value (or **error** if the conversion fails).

round(*const x*) returns *int.* If x is an integer, the result is x. Otherwise, x is converted to a real y by the function "real" above, and the result is the nearest integer to y. If y is midway between two integers, the even integer is returned. The result is **error** if the conversion fails or the resulting integer does not fit in 32 bits.

random(*number x*) returns *int.* If x is an integer, the result is an integer random number r in the range $0 <= r < x$. If x is a real number, the result is a real random number in the same range.If x is anything else, the result is an error.

strcat(*any) returns *string.*** Each argument is converted to a string by the function "string" above. The result is the concatenation of the strings.

substr(*string s*, *int offset* [, *int length*]) returns *string.* The result is the substring of s starting at the position indicated by *offset* with the length indicated by *length*. The first character of s is at offset 0. If *offset* is negative, it is replaced by length(s) − *offset*. If *length* is omitted, the substring extends to the end of s. If *length* is negative, an intermediate result is computed as if length were omitted, and then –*length* characters are deleted from the right end of the result. If the resulting substring lies partially outside the limits of s, the part that lies withing s is returned. If the substring lies entirely outside s or has negative length (because of a negative *length* argument), the result is the null string. [**Note**: This function is the same as the `substr` function of Perl.]

strcmp(*any a*, *any b*) returns *int.* The operands are converted to strings by the "string" function above. The result is an integer less than, equal to, or greater than zero according to whether a is lexicographically less than, equal to, or greater than b. Note that case *is* significant in the comparison.

stricmp(*any a*, *any b*) returns *int.* The same as **strcmp** except that upper and lower case letters are considered equivalent.

toUpper(*string s*) returns *string.* The operands are converted to strings by the "string" function above. The result is a string that is identical to s except that all lowercase letters in s will be converted to uppercase.

toLower(*string s*) returns *string*. The operands are converted to strings by the "string" function above. The result is a string that is identical to s except that all uppercase letters in s will be converted to lowercase.

member(*const x*, *string l*) returns *boolean*. If x is not a constant or l is not a list, then the result is an error. Otherwise, if any of the elements is equal to x in the sense of the `==` operator, then the result is true, otherwise it is false.

regexp(*string pattern*, *string target*, *string options*) returns *boolean*. If the regular expression *pattern* matches the *target*, this function returns true, but otherwise returns false.

Unfortunately, the allowed patterns and options cannot be precisely defined at this time because different ClassAd implementations use different underlying libraries to implement regular expression matching. The Java implementation uses Sun's regular expression implementation, which is Perl-like. The C++ implementation uses either POSIX or Perl-compatible regular expressions, depending on what is available at compilation time. This dichotomy is unfortunate, and we hope to resolve it in the future. For now, you must either know details about your ClassAd implementation, or you must use a subset of regular expressions that are POSIX- and Perl- compatible.

The options are specified as a string of letters in any order. Each letter indicates a single option. For POSIX regular expression matching, the only option is "i" for case-insensitive matching. For Perl-compatible regular expressions, there are four options: "i" for case-insensitive matching, "m" for multiline matching, "x" for extended syntax, and "s" to cause the dot character (.) to match all characters, including newlines.

identicalMember(*const x*, *string l*) returns *boolean*. If x is not a constant or l is not a list, then the result is an error. Otherwise, if any of the elements is equal to x in the sense of the `is` operator, then the result is true, otherwise it is false.

time() returns *int*. Returns the current Coordinated Universal Time, in seconds since midnight January 1, 1970.

interval(*int t*) returns *string*. The operand t is treated as a number of seconds. The result is a string of the form `days+hh:mm:ss`. Leading components are omitted if they are zero. For example, if the operand is $1472523 = 17 * 24 * 60 * 60 + 1 * 60 * 60 + 2 * 60 + 3$ (seventeen days, one hour, two minutes, and three seconds), the result is `"17+1:02:03"`; if the operand is 67, the result is `"1:07"`.

absTime(*string s*) returns *AbsTime*. The operand s is parsed as a specification of an instant in time (date and time). This function accepts the canonical native representation of AbsTime values, but minor variations in format are allowed.

The default format is `yyyy-mm-ddThh:mm:sszzzzz` where zzzzz is a time zone in the format `+hh:mm` or `-hh:mm`, but variations are allowed.

- Each separator character `-`, `:`, or `T` may be omitted or replaced by any sequence of non-digits. Note, however, that the `-` in a time zone of the form `-hh:mm` may not be omitted.
- The colon between `hh` and `mm` in the time zone may be omitted.
- An arbitrary sequence of non-digit characters may precede `zzzzz` or `yyyy`.
- The zone may be replaced by the character `z` or `Z`, which is equivalent to `-00:00`.

- The zone may be omitted, in which case the local time zone is used. If the string ends with `+dddd`, `-dddd`, `z`, or `Z`, where each `d` is a digit, this suffix is considered to be the time zone indication. For example, in `2003+1030`, the suffix `1030` is interpreted as a time zone 10 hours and 30 minutes east, rather than as October 30.
- The fields `ss`, `mm`, `hh`, etc. may be omitted (from right to left), in which case the omitted fields are assumed to be zero.

More precisely, the string must match the regular expression

```
D* dddd [D* dd [D* dd [D* dd [D* dd [D* dd D*]]]]] [-dd[:]dd|+dd[:]dd|z|Z]
```

Where `d` stands for a digit and `D` stands for a non-digit.

For example, in the United States central time zone, an AbsTime corresponding to "9 am Jan 25, 2003 CST" may be created by any of the function calls

```
2003-01-25T09:00:00-06:00     // canonical
2003-01-25   09:00:00 -0600   // different separators
20030125090000-0600           // compact format
2003-01-25 16:00:00 +01:00    // different time zone
2003-01-25 15:00Z             // omitted seconds, UTC time zone
2003-01-25 09:00:00           // default time zone (local)
2003-01-25 09                 // omitted minutes and seconds
```

and AbsTimes corresponding to "Jan 25, 2003" (implicitly midnight, UTC) may be written

```
2003-01-24T18:00:00-06:00   // canonical
2003-01-25T00:00:00         // default time zone: UTC
2003-01-25                  // omitted time of day
2003/01/25                  // different separators
20030125                    // compact format
```

The strings `2003-01-25T09:00:00-06:00` and `2003-01-25 15:00Z` represent the same instant in time, but measured in different time zones.

The following strings are invalid.

```
2003-01-25T09:00:00-06      // incomplete time zone
2003-01-25T09:00:00- 0600   // space in time zone
2003-1-25                   // missing digit in dd field
```

absTime([*const t[, int z]*) returns *AbsTime.*]

Creates an AbsTime value corresponding to time t an time-zone offset z. If t is a String, then z must be omitted, and t is parsed as a specification as described above. If t and z are both omitted, the result is an AbsTime value representing the time and place where the function call is evaluated. Otherwise, t is converted to a real by the function "real" above, and treated as a number of seconds from the epoch, Midnight January 1, 1970 UTC. If z is specified, it is treated as a number of seconds *east* of Greenwich. Otherwise, the offset is calculated from t according to the local rules for the place where the function is evaluated.

relTime(*const t*) returns *RelTime*. If the operand t is a String, it is parsed as a specification of a time interval. This function accepts the canonical native representation of RelTime values, but minor variations in format are allowed.

Otherwise, t is converted to a real by the function "real" above, and treated as a number of seconds.

The default string format is `[-]days+hh:mm:ss.fff`, where leading components and the fraction `.fff` are omitted if they are zero. In the default syntax, days is a sequence of digits starting with a non-zero digit, `hh`, `mm`, and `ss` are strings of exactly two digits (padded on the left with zeros if necessary) with values less than 24, 60, and 60, respectively and `fff` is a string of exactly three digits. In the relaxed syntax,

- Whitespace may be added anywhere except inside the numeric fields `days`, `hh`, etc.
- Numeric fields may have any number of digits and any non-negative value.
- The `+` may be replaced by `d` or `D`.
- The first `:` may be replaced by `h` or `H`.
- The second `:` may be replaced by `m` or `M`.
- The letter `s` or `S` may follow the last numeric field.
- If field i is terminated with one of the letters `dDhHmMsS` and the value of field $i-1$ is zero, field $i-1$, together with its terminating field name (`+`, `:`, `h`, etc.) may be omitted even if field $i-2$ is not omitted.
- The fraction `.fff` may have any number of digits. If it has no digits, the preceding decimal point may be omitted.

For example, one day, two minutes and three milliseconds may have any of the forms

```
1+00:02:00.003     // the result of relTimeToString
1d0h2m0.003s       // similar to ISO 8601
1d 2m 0.003s       // add spaces, omit hours field
1d 00:02:00.003    // mixed representations
1d 00:00:120.003   // number of seconds greater than 59
86520.002991       // seconds, excess precision in fraction
```

splitTime(*RelTime*) returns *ClassAd*. Creates a ClassAd with each component of the time as an element of the ClassAd. The ClassAd has five attributes:

```
Type      // "RelativeTime"
Days      // the number of days
Hours     // the number hours
Minutes   // the number of minutes
Seconds   // the number of seconds
```

splitTime(*AbsTime*) returns *ClassAd*. Creates a ClassAd with each component of the time as an element of the ClassAd. The ClassAd has five attributes:

`Type` // "AbsoluteTime"
`Year` // the year
`Month` // the month, from 1 (January) through 12 (December)
`Day` // the day, from 1 through 31
`Hours` // the number of hours
`Minutes` // the number of minutes
`Seconds` // the number of seconds
`Offset` // the timezone offset in seconds

formatTime(*AbsTime t*[, *string s*) returns *string.*]

This function creates a formatted string that is a representation of the absolute time *t*.

The string is similar to the ANSI C strftime function. It consists of arbitary text plus placeholders for elements of the time. These placeholders are percent signs (%) followed by a single letter. To have a percent sign in your output, you must use a double percent sign (%%).

Because an implementation may use strftime() to implement this, and some versions implement extra, non-ANSI C options, the exact options available to an implementation may vary. An implementation is only required to implement the ANSI C options, which are:

%a // abbreviated weekday name
%A // full weekday name
%b // abbreviated month name
%B // full month name
%c // local date and time representation
%d // day of the month (01-31)
%H // hour in the 24-hour clock (0-23)
%I // hour in the 12-hour clock (01-12)
%j // day of the year (001-366)
%m // month (01-12)
%M // minute (00-59)
%p // local equivalent of AM or PM
%S // second (00-59)
%U // week number of the year (Sunday as first day of week) (00-53)
%w // weekday (0-6, Sunday is 0)
%W // week number of the year (Monday as first day of week) (00-53)
%x // local date representation
%X // local time representation
%y // year without century (00-99)
%Y // year with century
%Z // time zone name, if any
% // %

Note that names may be locale-dependent, if the underlying operating system supports locales. Also note that some ClassAd implementations may have difficulty with time zone names for non-local time zones, since the names may vary.

formatTime(*int i*[, *string s*) returns *string.*]

This version of formatTime converts i to an absolute time, then behaves identically to the other version of formatTime.

5 Acknowledgments

The ClassAd language was originally developed by Wieru Cai, Miron Livny, and James Pryne as part of the Condor [2, 7, 8] project at the University of Wisconsin—Madison [10]. Under the leadership of Rajesh Raman, the language was made substantially more powerful (including such features as lists, arbitrary nesting of lists and records, and the Boolean, AbsTime and RelTime data types), while at the same time, the specification was made substantially simpler and more regular. In particular, the specification of the language was separated from details of its use in Condor. See Raman's Ph. D. dissertation [9] for a more readable introduction to the language and its design goals and a discussion of implementation issues, and a related conference paper [11] for a discussion of the use of ClassAds and Matchmaking in resource allocation. Raman is also the author of the current C++ implementation. Marvin Solomon is the author of the Java implementation.

Currently, the language is being maintained and developed by Alain Roy and Marvin Solomon, with input from other members of the Condor team. For the latest information about ClassAds, please see the ClassAd home page [1].

References

[1] Classified Advertisements. Available online at http://www.cs.wisc.edu/condor/classad.

[2] Condor: High Throughput Computing. Available online at http://www.cs.wisc.edu/condor.

[3] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes, 2001. Available online at http://www.w3.org/TR/xmlschema-2.

[4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (second edition), 2000. Available online at http://www.w3.org/TR/REC-xml.

[5] David C. Fallside. XML Schema Part 0: Primer, 2001. Available online at http://www.w3.org/TR/xmlschema-0.

[6] Bill Joy (Editor), Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification (Second Edition).* Addison–Wesley Pub Co., 2000. Available online at http://www.java.sun.com/docs/books/jls/index.html.

[7] M. J. Litzkow and M. Livny. Experience with the Condor Distributed Batch System. *IEEE Workshop on Experimental Distributed Systems*, 1990.

[8] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—A Hunter of Idle Workstations. In *Proc. of the 8th Int'l Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[9] Rajesh Raman. *Matchmaking Frameworks for Distributed Resource Management.* PhD thesis, University of Wisconsin—Madision, October 2000. Available online at http://www.cs.wisc.edu/condor/doc/rajesh.dissert.pdf.

[10] Rajesh Raman, Miron Livny, James Pruyne, and Wieru Cai. Classified Advertisements: Official Specification Version Alpha, April 1997. Unpublished Manuscript.

[11] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998. Available online at http://www.cs.wisc.edu/condor/doc/hpdc98.ps.

A YACC Grammar

```
%token ERROR
%token FALSE
%token IDENTIFIER
%token INTEGER
%token LEX_ERROR
%token REAL
%token STRING
%token TRUE
%token UNDEFINED

%left OP_LOR
%left OP_LAND
%left '|'
%left '^'
%left '&'
%left OP_EQ OP_NE OP_IS OP_ISNT
%left '<' '>' OP_LE OP_GE
%left OP_SHL OP_SHR OP_SHRR
%left '+' '-'
%left '*' '/' '%'
%left UNARY

%%


Expression
    : Expr
    | Expr '?' Expression ':' Expression
    ;
Expr
    : Expr OP_LOR Expr
    | Expr OP_LAND Expr
    | Expr '|' Expr
    | Expr '^' Expr
    | Expr '&' Expr
    | Expr OP_EQ Expr
```

```
    | Expr OP_NE Expr
    | Expr OP_IS Expr
    | Expr OP_ISNT Expr
    | Expr '<' Expr
    | Expr '>' Expr
    | Expr OP_LE Expr
    | Expr OP_GE Expr
    | Expr OP_SHL Expr
    | Expr OP_SHR Expr
    | Expr OP_SHRR Expr
    | Expr '+' Expr
    | Expr '-' Expr
    | Expr '*' Expr
    | Expr '/' Expr
    | Expr '%' Expr
    | PrefixExpr
    ;
PrefixExpr
    : SuffixExpr
    | '+' PrefixExpr %prec UNARY
    | '-' PrefixExpr %prec UNARY
    | '!' PrefixExpr %prec UNARY
    | '~' PrefixExpr %prec UNARY
    ;
SuffixExpr
    : Atom
    | SuffixExpr '.' Identifier
    | SuffixExpr '[' Expression ']'
    ;
Atom
    : Identifier
    | Literal
    | List
    | Record
    | Call
    | '(' Expression ')'
    ;
List
    : '{' ListBody ListEnd
    | '{' ListEnd
    ;
ListEnd
    : '}'
    | ',' '}'
    ;
ListBody
```

```
    : Expression
    | ListBody ',' Expression
    ;
Record
    : '[' DefinitionList RecordEnd
            Expr e = (Expr) i.next();
            if (r.lookup(n) != null)
    | '[' RecordEnd
    ;
RecordEnd
    : ']'
    | ';' ']'
    ;
DefinitionList
    : Identifier '=' Expression
    | DefinitionList ';' Identifier '=' Expression
    ;
Call
    : Identifier '(' ListBody ')'
    | Identifier '(' ')'
    ;
Literal
    : REAL
    | Strings
    | INTEGER
    | TRUE
    | FALSE
    | UNDEFINED
    | ERROR
    ;
Strings
    : STRING
    | Strings STRING
    ;
Identifier
    : IDENTIFIER
    ;
```

B XML Document Type Definition (DTD)

```
<!-- Document Type Definition for ClassAds (Classified Advertisements, as used
     in Condor.  See http://www.cs.wisc.edu/condor for more information. -->

<!-- Root element: a sequence of ClassAd elements -->
<!ELEMENT classads (i | r | s | e | b | at | rt | un | er | l | c)*>
```

```
<!-- A ClassAd (Classified Advertisement):  A set of "attributes". -->
<!ELEMENT c     (a)* >

<!-- An "attribute" contains a named element -->
<!ELEMENT a     (i | r | s | e | b | at | rt | un | er | l | c)>
<!ATTLIST a n   CDATA #REQUIRED>

<!-- Numbers -->
<!ELEMENT i     (#PCDATA)> <!-- An integer (sequence of digits) -->
<!ELEMENT r     (#PCDATA)> <!-- A real number in "scientific" notation -->

<!-- Strings -->
<!ELEMENT s     (#PCDATA)>

<!-- Sub-expressions in native syntax -->
<!ELEMENT e     (#PCDATA)>

<!-- Booleans: no content, but one mandatory attribute v=t or v=f -->
<!ELEMENT b     EMPTY>
<!ATTLIST b v   (t | f) #REQUIRED >

<!-- Timestamps -->
<!ELEMENT at    (#PCDATA)> <!-- Absoute time, e.g. 2003-01-25T09:00:00-06:00 -->
<!ELEMENT rt    (#PCDATA)> <!-- Relative time, e.g. P1H15M13.257S -->

<!-- Undefined and error -->
<!ELEMENT un    (#PCDATA)> <!-- Undefined; content, if any, is "annotation" -->
<!ELEMENT er    (#PCDATA)> <!-- Error; content, if any, is "annotation" -->

<!-- A List: a sequence of ClassAd elements -->
<!ELEMENT l     (i | r | s | e | b | at | rt | un | er | l | c)*>
```

C XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation><xsd:documentation xml:lang="en">
    Schema for Classified Advertisements (classads) as used in the
    Condor High Throughput Computing progject.
  </xsd:documentation></xsd:annotation>

  <xsd:element name="classads">
    <xsd:annotation><xsd:documentation xml:lang="en">
      The root document element.  It is identical to an "l" element
      except for the tag: Its content is simply a sequence of
```

```
    expressions.
  </xsd:documentation></xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="c"/>
        <xsd:element ref="l"/>
        <xsd:element ref="e"/>
        <xsd:element ref="s"/>
        <xsd:element ref="i"/>
        <xsd:element ref="r"/>
        <xsd:element ref="b"/>
        <xsd:element ref="er"/>
        <xsd:element ref="un"/>
        <xsd:element ref="at"/>
        <xsd:element ref="rt"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="s" type="xsd:string">
  <xsd:annotation><xsd:documentation xml:lang="en">
    A string literal.
    Content is the string itself.  It may contain character entities,
    such as &amp;lt;.  It may also contain backslash escapes, which
    are interpreted by the application.
  </xsd:documentation></xsd:annotation>
</xsd:element>

<xsd:element name="i" type="xsd:int">
  <xsd:annotation><xsd:documentation xml:lang="en">
    An integer literal.
    Content is the value, as a sequence of digits.
  </xsd:documentation></xsd:annotation>
</xsd:element>

<xsd:element name="r">
  <xsd:annotation><xsd:documentation xml:lang="en">
    A real (double-precision floating point) literal.
    Content is the value in "scientific" notation corresponding to
    the printf %1.5E format, or one of the special values INF, -INF, or NaN.
  </xsd:documentation></xsd:annotation>
  <xsd:simpleType>
    <xsd:restriction base="xsd:double">
      <xsd:pattern value="-?INF|NaN|-?\d\.\d{15}E(\+|\-)\d{2,3}"/>
```

```
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>

<xsd:element name="b">
  <xsd:annotation><xsd:documentation xml:lang="en">
    A boolean literal.  Represented as an element with empty content
    and required attribute v="t" or v="f".
    Content must be empty.
  </xsd:documentation></xsd:annotation>
  <xsd:complexType>
    <xsd:attribute name="v">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="t"/>
          <xsd:enumeration value="f"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
</xsd:element>

<xsd:element name="er">
  <xsd:annotation><xsd:documentation xml:lang="en">
    An error value.  The "a" attribute, if present, is an annotation
    (reason for the error).
    Content must be empty.
  </xsd:documentation></xsd:annotation>
  <xsd:complexType>
    <xsd:attribute name="a" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="un">
  <xsd:annotation><xsd:documentation xml:lang="en">
    An undefined value.  The "a" attribute, if present, is an
    annotation (perhaps the name of the classad attribute that was
    undefined).
    Content must be empty.
  </xsd:documentation></xsd:annotation>
  <xsd:complexType>
    <xsd:attribute name="a" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="at">
```

```
  <xsd:annotation><xsd:documentation xml:lang="en">
    An absolute time literal.
    Content is yyyy:mm:ddThh:mm:ssphh:mm where p is + or -, other
    lower-case letter represent arbitrary digits, and the remaining
    characters must appear exactly as shown.
    Note that the time zone is required and it contains a colon.
  </xsd:documentation></xsd:annotation>
  <xsd:simpleType>
    <xsd:restriction base="xsd:dateTime">
      <xsd:pattern value=
        "\d{4}-\d\d-\d\dT\d\d:\d\d:\d\d[+\-]\d\d:\d\d"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>

<xsd:element name="rt">
  <xsd:annotation><xsd:documentation xml:lang="en">
    A relative time literal.
    Content is -PnDTnHnMn.dddS where "n" represents one or more digits
    and "d" represents one digit.  The leading sign and individual
    components (nD, nH, etc.) may be omitted, but if present, they
    must appear in the order indicated. The 'T' is omitted if nD is the
    only field. The fractional seconds, together with the decimal
    point, may be omitted, but if present there must be exactly three
    digits.  Note, the pattern below does not completely specify all the
    constraints.  For example, the T must be omitted if and only if the
    H, M, and S fields are all omitted, and the attribute may not be
    completely empty.
  </xsd:documentation></xsd:annotation>
  <xsd:simpleType>
    <xsd:restriction base="xsd:duration">
      <xsd:pattern value="-?P(\d+D)?T?(\d+H)?(\d+M)?(\d+(\.\d\d\d)?S)?"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>

<xsd:element name="e" type="xsd:string">
  <xsd:annotation><xsd:documentation xml:lang="en">
    An expression.
    Content is an arbitrary classad expression in the "native syntax".
    This element should not be used for literals or if the "top-level
    operator" is list or classad.
  </xsd:documentation></xsd:annotation>
</xsd:element>

<xsd:element name="l">
```

```
  <xsd:annotation><xsd:documentation xml:lang="en">
    A list. Content is a sequence of arbitrary expressions.
  </xsd:documentation></xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="c"/>
        <xsd:element ref="l"/>
        <xsd:element ref="e"/>
        <xsd:element ref="s"/>
        <xsd:element ref="i"/>
        <xsd:element ref="r"/>
        <xsd:element ref="b"/>
        <xsd:element ref="er"/>
        <xsd:element ref="un"/>
        <xsd:element ref="at"/>
        <xsd:element ref="rt"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="c">
  <xsd:annotation><xsd:documentation xml:lang="en">
    A "classad" (a record). Content is a sequence of "a" elements.
  </xsd:documentation></xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="a" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:choice>
              <xsd:element ref="c"/>
              <xsd:element ref="l"/>
              <xsd:element ref="e"/>
              <xsd:element ref="s"/>
              <xsd:element ref="i"/>
              <xsd:element ref="r"/>
              <xsd:element ref="b"/>
              <xsd:element ref="er"/>
              <xsd:element ref="un"/>
              <xsd:element ref="at"/>
              <xsd:element ref="rt"/>
            </xsd:choice>
          </xsd:sequence>
          <xsd:attribute name="n" type="xsd:string"/>
```

```
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:unique name="dummy">
    <xsd:selector xpath="a"/>
    <xsd:field xpath="@n"/>
  </xsd:unique>
</xsd:element>

</xsd:schema>
```