

April for the not so foolish

A Tutorial and User's Guide

Alan H. Bond and Frank G. McCabe

December 1, 1999

This is a tutorial user guide for the programming language April. It is intended to give readers an introduction to the language, with sufficient detail to allow them to write April programs.

It was obtained mainly by editing a shorter version from the full April reference manual written by Frank G. McCabe and Keith L. Clark. For further details, simply consult the full version of that manual. In particular, there is an html form which is indexed from a contents page.

1 Introducing April

April is an Agent Programming and Implementation Language. It was developed by Frank McCabe and Keith Clark, during the period 1992-1998, at Imperial College, London and at Fujitsu's Network Agents Research Group, in Tokyo and in San Jose, California

April is a strongly-typed functional language like ML, with additional mechanisms for multi-agent programming. Being a functional language means that the fundamental expression is a functional application,

```
FUNCTION ARG
```

and that a program is a compound expression made up of nested subexpressions. Being strongly typed means that every subexpression has a type that can be inferred at compile time. Computation consists of evaluating these expressions, so every subexpression will evaluate to a value whose type was determined at compile time.

In addition to functions and procedures, April provides processes which can be created with an explicit spawn statement

```
HANDLE = spawn STATEMENT
```

where STATEMENT is any statement, usually a procedure call, and HANDLE is a process handle, allowing the resulting process to be referred to. Processes can send each other messages

```
MESSAGE >> HANDLE
```

where MESSAGE can be any April expression.

Messages are held in a single message queue for each process. A process reads messages from its message queue using a `receive` procedure

```
receive{ PATTERN1 - >> STATEMENT1 | PATTERN2 - >> STATEMENT2 | .. }
```

A PATTERN being an expression containing initially unbound variables.

Processes can be run on a set of computers forming a network, and their distribution over the network is transparent to the user.

April is a higher-order language, meaning that functions and procedures are first class objects, and thus can be bound as values of variables, can be members of compound data structures such as lists, can be passed as arguments and returned as values. They can therefore also be sent as messages.

An April agent is a group of processes, including one publicly named process and all the processes spawned from that process. Mobile agents can be obtained by sending messages containing procedures.

April is extensible using its macro language, and April's syntax is extensible, being an operator precedence grammar.

2 April data values and types

April data values fall into three main categories: primitive values, complex or compound values, and values of user-defined types.

All data values in April have a standard written representation - that is, it is possible to write an explicit literal value of any particular type. For example, the expression: 10.34 refers to a number, and ["alpha","beta"] to a list of strings.

2.1 Primitive types

A primitive or scalar type is one whose values have no internal structure - they can be manipulated and processed but they cannot be inspected.

(i) April numbers are either integers or floating points.

(ii) The `symbol` type refers to symbols in the program. Symbols are identified by prefixing an identifier with a quote character: `'foo`

(iii) `string` values are written as sequences of characters surrounded by double-quote characters: `"This is a string of 33 characters"`.

2.2 Compound types

April has three built-in methods for composing data values - `lists`, `tuples` and `records`.

In addition, a `labelled record` is a form of tuple/record which is introduced by means of a user-defined type declaration.

Records are similar to tuples where individual components of the tuple are identified by name rather than solely by position. In addition, there are other styles of compound type - such as `function` and `procedure` types. Also, the generic type `any` provides a mechanism for escaping the type system.

2.2.1 The list type

A `list` is an ordered sequence of values, all of which have the same type. They are written as a comma-separated sequence surrounded by `[]`s. e.g. `[2, 3, 5, 7, 11]`.

Lists are formed by using the list constructor `“,..”` which constructs a list from a given head and tail.

2.2.2 The tuple type

A `tuple` is a ordered sequence of values, each of which is potentially of a different type. Tuples are written as a comma-separated sequence surrounded by `()`s, e.g. `("Fred. U. Are", 40)` The components of a tuple such as this can only be accessed via a pattern matching expression.

2.2.3 The record type

A `record` is a tuple in which the elements are identified by a field name. A record type is similarly a tuple type in which elements of the type are identified by field name. For example, `(name="Fred. U. Are", age=40)` whose type description is `(string?name, number?age)`, where `name` and `age` are symbolic field names identifying the first and second elements of the record.

In most cases April requires that elements of record literals follow the same order as in the appropriate type description. For example, the nearly equivalent record: `(age=40, name="Fred. U. Are")` has a different, incompatible, type.

A major example of a record is the `theta` expression which are used to contain functions and procedures.

Field names are most often used in conjunction with the dot operator for example we can access the name field using `R.name`.

On the other hand, April does not permit record patterns to match and access elements of a record.

2.3 Standard April Types

The complete April language system includes a number of standard types

(i) The `logical` type, `logical ::= true | false`

(ii) The `handle` type handles identify processes. Usually, handle values are not constructed directly, instead they are constructed via special operators that spawn new processes or via typecast expressions. For example, the expression:

```
handle%"es@smart.systems.com"
```

identifies a public April process - called `es` - which is residing on the host `smart.systems.com` This mechanism allows an April-based service to publish itself on the Internet without needing to communicate its handle beforehand.

(iii) the `error` type. Values of the error type is used by many built-in functions and procedures to encapsulate critical data when an exception is raised. It has value `failed` when an equation or statement clause fails to apply to a set of arguments.

3 April expressions

The legal expressions in a programming language can be classified into value expressions - such as arithmetic or list expressions - and syntax expressions - which reflect syntactic constructs in the language.

3.1 Function application expressions

Many of the value expression forms are described previously as data values and types. In this section we describe one more form of value expression, namely function application expressions.

A function application expression takes the form: `FUN ARG`, where `FUN` and `ARG` are also all expressions. In the case of the `FUN` expression, it should evaluate to a function value. In the case of `ARG`, it will often evaluate to a tuple of arguments; the value of the expression is the result of applying the function to the arguments `ARG1, . . . , ARGK`.

3.2 Syntax expressions

April has a relatively rich collection of syntax expressions.

3.2.1 Valof expressions

The `valof` expression allows the value of an expression to be determined as a result of executing one or more statements. The form of the `valof` expression is: `valof STATEMENT`, where `STATEMENT` should include an `valis` statement: `valis EXPRESSION`.

3.2.2 Try...onerror expressions

A `try...onerror` expression allows a programmer to handle failure within an expression. The form of a `try...onerror` expression is: `try EXP onerror FUN`, where `EXP` is evaluated, and if successful, becomes the value of the entire expression. `FUN` is a single-argument function, whose argument is of type `error`. It is applied to the error code generated as a result of the run-time exception. The value returned by `FUN` becomes the value returned by the entire `try...onerror` expression.

3.2.3 Exception expressions

The `exception` expression does not have a value, although it does have an argument! Instead it forces the termination of current evaluation, and causes the generation of a run-time exception. The exception generated is based on the argument of the exception expression: `exception EXP`.

3.2.4 Collect expressions

The `collect` expression allows a list of values to be computed - as the result of a statement. The form of a collect expression is: `collect STATEMENT`, where `STATEMENT` is expected to contain at least one `elemis` statement. The `elemis` statement(s) represent the elements of the resulting list - each time an `elemis` statement is executed, the value associated with it is added to the end of the list being constructed.

3.2.5 Setof expressions

The `setof` expression is similar to the `collect` expression, except that the resulting list is sorted with duplicates removed - i.e., a set is returned rather than a bag. The form of a `setof` expression is: `setof STATEMENT`, where `STATEMENT` is expected to contain at least one `elemis` statement. The `elemis` statement(s) represent the elements of the set.

3.2.6 Case expressions

The `case` expression allows the value of an expression to be determined by cases in the value of a governing expression. The form of the `case` expression is:

```
case EXP in { PTN1 => E1 | ... | PTNK => EK }
```

where `PTNI` are all `PATTERNS` compatible with `EXP`, and expressions `EI` are all expressions of the same type. The value of a case expression depends on the first pattern `PTNI` that matches `EXP`. This pattern selects a corresponding expression - `EI` - which is the value returned by the case.

Note that the body of the case expression takes the form of a disjunction of equations - i.e., it takes the form of a function, and indeed it can also be written: `case EXP in FUN_VAR`.

3.2.7 Conditional expressions

A conditional expression has the form `if TEST then EXP1 else EXP2`. The `TEST` predicate part of the expression is evaluated, and depending on whether it evaluates to true or false, the value of the complete expression is `EXP1` or `EXP2` respectively.

Conditional expressions are useful in the definition of simple functions as well as in statements: `app(X,Y) => { if [?E1,..?Z] . = X then [E1,..app(Z,Y)] else Y }`

Conditional expressions are equivalent to a specific use of the `valof` construct. The above function definition is entirely equivalent to: `app(X,Y) => valof { if [E1,..Z] . = X then valis [E1,..app(Z,Y)] else valis Y }`

3.2.8 Typecast and type coercion expressions

`TYPE ?? E` denotes that the type of the expression `E` is `TYPE`.

Complimentary to typecast expressions are the type coercion expressions. A type coercion is similar to a typecast except that some transformation of the input is often implied. The form of a type coercion expression is `TYPE %% E`.

3.2.9 Record expressions

There are various kinds of record expressions, namely, anonymous record expressions, labelled tuple expressions and theta expressions.

Anonymous record expressions. A basic anonymous record expression is really a tuple in which the elements are a sequence of field expressions: `(FIELD1, ... FIELDK)`

A `FIELD` is written using the '=' operator: `NAME=EXPRESSION`.

For example, a personnel record may look like: `(name="Foo",dept="Bar")`

Labelled tuple expressions In the core form of a labelled tuple, it is written as the `label` followed by the argument `tuple`. The elements of the `tuple` contain, in order, appropriate expressions which are consistent with the type declaration.

For example, the following type definition introduces a type of binary labelled trees, with the node labelled record used to describe nodes in the tree:

```
tree(%a) ::= empty | node(tree(%a)?left,%a?lbl,tree(%a)?right)
```

Given such a type declaration, providing that we are within the scope of the type declaration, we can write down a tree expression directly using this data type:

```
node(empty,10,node(empty,12,empty))
```

In some situations, a record can be viewed as a kind of binding environment. The fields of the record introduce variables which have a certain scope. In the case of regular anonymous records and labelled tuples the scope of the field variables does not extend to the values defined in a given record literal itself. In the case of theta expressions, the scope of fields includes the fields themselves.

Theta expressions A **theta** expression is written using semi-colon to separate a sequence of fields; each of which is denoted either by an = - for a read-only definition - or by a : for a read/write definition.

```
{ FIELD1; ... FIELDK }
```

3.3 Lists and set abstractions

April has a higher-level way of specifying lists and sets based on a set-abstraction notation. These allow lists and sets to be specified in terms of logical conditions on the elements of the lists and sets, as opposed to low-level iterations on lists and sets.

The general form of such an expression is:

```
SETOF { TEMPLATE : GENERATING_EXPRESSION },
```

where the SETOF is a specifier denoting the nature of the list being built, TEMPLATE is an expression which denotes the shape of the elements of the list and GENERATING_EXPRESSION is a logical expression that governs the conditions for elements in the set.

The form of a **setof** abstraction is: **setof** {T : CONDITION}

The value of this expression is a list of T values, generated by means of CONDITION. For example, given a list of the form

```
ages = [('a',10),('b',20),('c',10),('d',20)];
```

we can construct a set of age values in the range 10-15 by means of:

```
S = setof {A : (_,A) in ages && A>=10 && A<=15 }
```

The value of S would be: [10]

bagof is similar to the **setof** abstraction, except that the result is not sorted, and there may be duplications.

The **counting** abstraction is similar to the **bagof** abstraction, except that a limit is placed on the size of the resulting list. For example, to find at most 2 males, whose ages are greater than 10, we might use the expression:

```
2 of { M : M in male && (M,A) in ages && A>10 }
```

The core method for generating elements of abstractions is the **in** test. Within a **setof**/**bagof**/N of abstraction, a condition of the form:

PTN in EXP

has the effect of generating instances of PTN that match elements of the list or set EXP.

Within a generating expression it is possible to combine two or more conditions using the `&&` conjunction. The meaning of a generating expression such as: `C1 && C2` is simply the conjunction of C1 and C2. Note that either C1 or C2 or both may be generating expressions or simply testing expressions.

Within a generating expression the `||` conjunction may be used to specify alternatives. The meaning of a generating expression such as: `C1 || C2` is simply the disjunction of C1 and C2. Note that either C1 or C2 or both may be generating expressions.

The `!` negation condition may be used for negative tests within the generating expression. Note that a negated condition can never be used to generate elements of a setof/bagof/N of abstraction; it can only filter out potential elements.

For example, to show those males who have no children, we can use:

```
setof { X : X in male && ! (X,-) in parent }
```

The `istrue` abstraction condition is used to verify that a particular condition is satisfied. `istrue` is used within the `CONDITION` of a setof/bagof/N of abstraction when it is important that the test condition is only completed once. `istrue CONDITION`, where `CONDITION` is any logical condition, including element generating conditions. For example, in:

```
bagof { M : M in male && istrue (M,A) in parent }
```

The `forall` abstraction condition is used to verify that a particular condition is satisfied for every element of a list or set. or example, to list males who have only male children, we can use the expression:

```
setof { M : M in male && C in male forall (M,C) in parent }
```

4 Patterns and pattern matching

4.1 Primitive patterns

There are primitive patterns - `number patterns`, `symbol patterns` and `string patterns` - corresponding to the primitive data types.

The `variable declaration pattern`.

A variable is declared in a pattern by its first occurrence - there does not need to be any special marking identifying the variable. Note that subsequent occurrences of an identifier will refer to the same variable; including situations where the variable was first declared in an outer scope.

The `anonymous variable pattern`

A “`_`” symbol occurring on its own in a pattern signifies a wild card or anonymous variable pattern. “`_`” will match any value - and the value is ignored. “`_`” patterns are useful as placeholders in more complex patterns where only part of the structure needs to be matched against.

4.2 List and other compound patterns

1. **list patterns.** A `list pattern` mimics the form of the list expression; the empty list - `[]` - matches the empty list, and the list pattern: `[HEAD,..TAIL]` matches a non-empty list provided that the head of that list matches `HEAD` and the remainder of the list matches `TAIL`.
2. **tuple patterns.** A `tuple pattern` is written using the normal tuple notation.
3. **labelled tuple patterns.**
4. **handle patterns.** The `handle record pattern` can be used to filter out messages based on properties of the handle:

```
receive{ PTN::(handle(nme.=name)::nme in ["foo",...,"others"]).=replyto  
- >> ... }
```
5. **guarded patterns.** A pattern of the form: `PTN :: TEST` is a `guarded pattern`. This pattern applies if both `PTN` applies and the predicate `TEST` evaluates to true.
6. **types as patterns.**

```
tree(number)?X - >> ...
```

7. string patterns.

The string concatenation operator `++` can also be used in **string patterns**. For example to extract the substring between a front and tail string, we can use:

```
"front" ++ sub ++ "tail" .= STRING_VALUE
```

The `~` string pattern can be used to pinch off a fixed portion of a string in a string pattern match.

```
(F++"*"++R)~10++L .= STRING_VALUE
```

matches the first 10 characters of the `STRING_VALUE` against the pattern:

```
F++"*"++R
```

The variable `L` is bound to the remainder of the string value if a `"*"` is found in the first 10 characters.

The `in` test may be used in a string pattern to verify that a sub-string is also a member of a list of strings.

4.3 Value extraction

The `%%` operator can be used to read a value from a string. The form of this string pattern is: `TYPE %% VAR`, where `TYPE` is any type expression. This string pattern will parse the string to verify that the string contains a `TYPE` value and binds the variable `VAR` to that value.

5 April statements

5.1 Primitive statements

April has relatively few different kinds of **primitive statement**, although there is a rich collection of built-in procedures.

- **Variable declaration statement** - declare variables in a procedure. A variable declaration statement takes one of two forms:
 $V : E$
which introduces the re-assignable variable V , or
 $V = E$
which introduces a single-assignment or read-only variable. In both cases the type of the variable is determined from the type of E , and the initial value of V is the value of E .

It is possible to explicitly indicate the type of the variable being declared by using a $?$ operator:

$T ? V : E$.

- **Assignment statements**, which reassign a variable to a new value take two main forms in April. The first takes the form:
 $VARIABLE := EXPRESSION$
where $VARIABLE$ is a previously declared read-write variable, and $EXPRESSION$ is a type-compatible expression.
The second form of assignment is the tuple assignment.
 $(V_1, \dots, V_K) := K\text{-TUPLE } EXPRESSION$.
In this form, a tuple of variables can be assigned simultaneously. This allows, for example, a tuple-valued function's results to be unpacked into separate variables. Tuple assignment is used in two common situations, where the right-hand-side of the assignment is a tuple-valued function call, and where the right-hand side is an explicit tuple.
- The **match statement**, which applies a pattern to a value, takes the form of:
 $PTN . = EXPRESSION$
The $EXPRESSION$ is evaluated and matched with PTN ; as a result, any variables declared in PTN are given their initial value.
- The **null statement**, $\{\}$ does nothing.
- The **procedure call statement**, which invokes a procedure takes the form:
 $PROC(E_1, \dots, E_K)$

where PROC is a procedure-valued expression, and EI are expressions corresponding to the arguments of the procedure.

5.2 Compound statements

April has the usual set of statement types common to functional languages.

- the statement sequence - {S1;S2;...}
- the while statement - while TEST do STATEMENT
- the for statement - for PTN in LIST do S
- the conditional statement - if TEST then S1 else S2
- the case statement - case EXP in {PTN1 - > S1 | PTN2 - > S2 ... | PTNK - > SK }
- the dot statement - R . STATEMENT, where R is a record and STATEMENT is any statement.
- the onerror statement - try S1 onerror PROC
- the exception statement - exception ERROR, The value of ERROR is passed to the error handling procedure.
- the labelled statement - LABEL: -STATEMENT
- the leave statement - leave LABEL, forces early termination of the labelled statement whose label is LABEL.

6 Messages

6.1 Sending messages between processes

In April, sending a message is technically a primitive statement, whereas the message receive statement is a compound statement. This is because the message receive statement selects which statement to execute based on the messages received.

- `>>` - send a message
`MSG >> TO`
where `TO` is a handle and `MSG` is any expression. The message `MSG` is sent to the `TO` process.
- `^^` - modify message attributes. To send a message to another process while requiring that any reply to the message should go to a third process. In April,
`MSG ^^ [replyTo(FROM)] >> TO`
This has the effect of modifying the `replyTo` attribute of the `MSG` so that replies will go to `FROM`. This does not affect the sender attribute.
- `>>>` - forward a message
`any?MSG >>> handle?TO`
This statement is only permitted on the right hand side of a message receive - `-->>` - operator. The message `MSG` is forwarded to the new receiver. Any reply which the new recipient sends to this message will go directly to the process which sent the message in the first place.
- `self post` - post a message on own message queue
`MSG >> self().`
`MSG !>>` puts it on the front of the message queue.
- `_front_msg(MSG,REPLY,SENDER)` - post message on front of message queue.
- `MSG >>* RECIPIENT-LIST` - Send a message to a list of processes

6.2 Messages and their attributes

When a message is sent between processes, a number of attributes may be associated with the message. These attributes allow the sending process to control some policy aspects of the message delivery.

- `replyTo(handle)` - reply to handle - defaults to `replyTo(handle?who=self())`
- `leaseTime(number)` - lifetime of message, in seconds since Jan 1st 1970, default is `leaseTime(number?when=0)`, 0 denoting infinity.
- `receiptRequest(any?code)` when the message is received, a reply message of the form: `receiptRequest(any?code)` is sent to the sending process. If the message times out then `timedOut(code)` is sent.
- The `auditTrail` message attribute is used to request an audit trail indicating the route that a message goes through from the sending process to the process that receives the message. The form of the `auditTrail` attribute is: `auditTrail(handle[]?trail=[])`. The `trail` is bound to a list of the handles of the intermediate communications servers that were involved in delivering the message.

6.3 Receiving messages

A process receives a message by executing a `receive` statement. A basic message receive statement takes the form:

```
receive PROCEDURE
```

where `PROCEDURE` is a message handling procedure. The standard way of writing a message handling procedure is as a sequence of message handling clauses:

```
receive {PTN1 - >> S1 | PTN2 - >> S2 | ... | PTNK - >> SK }
```

This statement will filter incoming messages until one that matches one of the patterns `PTNI`. If there are no matching messages then the process will suspend and wait for more messages to arrive.

More accurately, the April engine will examine each message in the process's message queue in turn - in the order that they were received. For each message, each of the patterns `PI` are tried, in turn from first to last, to see if it matches. If it does then the corresponding statement `SI` is executed.

Messages which do not match any of the `PTNI` patterns are not lost - they are simply kept in an internal process message queue until the process executes another message receive statement which can accept them.

As soon as a matching message is found, then the `SI` on the right hand side of the `- >>` operator is executed. Note that the `PTNI` may declare additional local variables - the scope of these local variables extends to the `SI`; but not outside the message receive statement itself.

6.4 Receiving message attributes

Each message receive clause has additional components which are automatically attached to the message:

- the handle of the `sender` of the message. This is made available as the value of the sender variable whose scope extends to the action part of the message receive clause.
- the `replyto` handle of the process that any replies should go to. This is made available as the value of the `replyto` variable whose scope extends to the action part of the message receive clause.
- the list of `message attributes` associated with the message; as requested by the sender of the message. This is made available as the value of the `options` variable whose scope extends to the action part of the message receive clause.

If one of the message clauses in a message receive has the form:

```
timeout EXP - >> ACTION
```

where `EXP` is a relative time expression (i.e., of type `rtime`), then the process will only wait an `EXP` amount time for a matching message to arrive.

6.5 Message streams

Normally messages are not consumed one at a time, but rather are consumed as a stream of messages. In April, we can consume a stream of messages using the repeat statements. These statements are not primitive in April, but are in fact macro-processed into simpler statements.

The `repeat...until` statement is a message terminated message stream statement. It consumes a stream of messages until a terminating message is received. The form of the `repeat...until` statement is:

```
repeat { PTN1 - >> S1 | PTN2 - >> S2 ... | PTNK - >> SK } until PTNQ
```

where `PTNI` are message receive patterns as in the normal message receive statement and `PTNQ` is also pattern.

This statement continues to consume any messages of the form `PTNI` (and executing the corresponding statements `SI`) until a message that matches `PTNQ` is received.

Note that the messages received by a process are handled in order of arrival, thus any termination message is handled after other messages which arrived earlier at the process.

Note also that any messages which arrive at the process which match none of PTNI or PTNQ are held in the processes message buffer without otherwise affecting the execution of this statement.

There is another form, namely `repeat...until alarm`. The `repeat...until alarm` statement is a timed message stream statement. It consumes a stream of messages for a period of real-time.

7 Functions and procedures

7.1 Functions and procedures in April

In April, functions and procedures are first class values. That means that a function or procedure can not only be used - as in applied to arguments - but it can also be stored as the value of the variable or even sent in a message to another process. This last capability gives rise to many powerful applications - including mobile agents, automatic configuration over a network and so on.

Functions and procedures can be defined in one of two fundamental contexts: within a theta expression and as a normal value-related expression. Functions and procedures defined as fields in a theta expression may be mutually recursive, whereas other functions or procedures are not.

7.2 Functions

April uses `equations` to define functions, where equations are expressed as a pattern/expression pair. Unlike most programming languages, functions in April are properly considered to be partial functions rather than total: that is, it is possible for a function not to apply to a given set of arguments. This property of functions is important because it allows us to combine them using a function union operator.

In general, a function is either an equation or a union of functions.

7.3 Function equations

An `equation` is a rewrite rule for reducing function application expressions to simpler expressions:

`PTN => EXP`

where `PTN` is a pattern and `EXP` is an expression.

When a function is applied, the `PTN` is matched against the actual argument of the function, and if the match is successful, then the function application expression is reduced to `EXP`. Since the `PTN` can declare variables - whose scope extends to `EXP` - the body of the function can access the actual values as bindings of variables.

The equation body `EXP` may refer to other variables than those declared in `PTN` - these

variables are free in the function. All free variables are ultimately bound by some outer scope that the function is embedded within. The most common free variables in function bodies are actually references to other functions. Although a function may contain references to free variables, it may not modify their values, unless they are theta variables. The variables that are declared in the equation pattern - which are effectively the formal parameters of the function - are single assignment or read-only variables. This implies that such variables may not be re-assigned to in the body of the equation.

A function that has several arguments really only has one - the arguments are all encoded as a tuple which forms the single argument of the function.

7.4 Function union

A `function union` is the union of two functions, written with a `|` operator:

`F | G`

where `F` and `G` are functions - of compatible type. The meaning of such a function is analogous to the mathematical union of the two partial functions `F` and `G`. Pragmatically, this means that when a function union `{F|G}` is applied, if the function `F` fails to apply then the function `G` is attempted. If `G` also fails to apply then the function union fails - which may lead to the application itself failing.

Where the patterns for the two component functions overlap, the first function takes precedence over the second. A mathematical function union would require that where the patterns overlap, the values of the component functions must be the same.

7.5 Function application

A `function application expression` is of the form: `FUN ARGUMENT`, where `FUN` is a function valued expression of compatible type with `ARGUMENT`. Typically the argument of a function is a tuple of the form `(A1, ..., AK)`; but where the function is single-arity then `FUN ARG` is entirely equivalent to `FUN(ARG)`

April is a strict call-by-value programming language - arguments to functions are evaluated before entering the function body. All of `FUN`, `A1, ..., AK` are evaluated prior to the `FUN` body itself. However, the precise order of evaluation of `FUN`, `A1, ..., AK` is not defined and programmers should not rely on a particular order of evaluation of function parameters.

All the parameters to a function are input - a function only returns a single value. If it is desired that a function return more than one value then arrange to return a tuple or

record of values:

```
fun = { (P1,...,PK) => (EXP1,...,EXPJ)}
```

April allows multiple variables to be assigned simultaneously as a tuple; so the results of such a function could be unpacked using an assignment of the form:

```
(V1,...,VJ) := fun(E1,...,EK).
```

It is worth noting again that functions and procedures may be sent in messages to other processes, including to processes on different computers. In this case, the April message system will ensure that both the function itself and the values of any free variables referred to in the function will also be sent.

7.6 Built-in April functions and procedures

We have put descriptions on April's comprehensive collection of functions and procedures in appendices. Appendix A describes arithmetic and mathematical functions. Appendix B describes string manipulation and formatting functions. Appendix C describes list and set processing functions. Appendix D describes tests, i.e. predicates and logical functions. Appendix E describes I/O for strings, files and sockets.

7.7 Examples of April functions

```
fact={
  0 => 1
  |
  N::N>=0 => N*fact(N-1)
};
```

- (i) `fact` is defined as a union of two partial functions.
- (ii) The `0` pattern is a constant pattern.
- (iii) `::N>=` is a guard condition on the variables in the pattern, `N`.

```
rev = {
  [] => []
  |
  [H,..T] => rev(T)<>[H]
};
```

- (i) `[]` is a **constant pattern** - the empty list.
- (ii) `[H,..T]` is a **list pattern**, using the list constructor operator `,..`
- (iii) `<>` is a list concatenation operator.
- (iv) `[H]` is a list expression evaluating to a list of one element.

```
double_list = {
  [] => []
  |
  [H,..T] => [2*H,..double_list(T)]
};
```

`[2*H,..double_list(T)]` is a list expression.

```
double_list = {
  L => bagof {2*X: X in L}
};
```

A **bagof** expression has two components, the first being a **pattern** and the second a condition on variables in the pattern.

```
order_double_list = {
  L => setof {2*X: X in L}
};
```

A **setof** expression similarly.

8 Procedures

April uses **clauses** to define procedures, where a clause is a pattern/action pair - analogous to the pattern/expression pair for equations. Like functions, April procedures have a semantics based on the success or failure of the application of a pattern to a set of arguments: this means that April procedures also have a form of partial semantics.

In general, a procedure is either a clause or a union of procedures.

8.1 Procedure clauses

A **clause** is a rewrite rule for reducing procedure application statements to simpler statements:

`PTN -> STMT`

where `PTN` is a pattern and `STMT` is a statement.

Although a procedure may contain references to free variables, it may not modify their values - apart from so-called theta-variables that are in the same scope. This has the effect of preventing procedures from side-effecting their environment in other than strictly controlled ways - one common way is by sending messages between processes.

The variables that are declared in the equation pattern - which are effectively the formal parameters of the procedure - are single assignment or read-only variables. This implies that such variables may not be re-assigned to in the body of the procedure clause.

A *special notation* for a **procedure clause** is used within a message **receive statement**, here a double arrow `->>` is used.

```
receive { PTN1 ->> A1 | PTN2 ->> A2 | ... | defaultProc }
```

This is essentially because the patterns match to messages, which actually contain four components, namely, content, sender, replyto and message attributes. However, in the usual case, we only want to explicitly match the message content component.

8.2 Procedure union

A **procedure union** is the union of two procedures, written with a `|` operator:

```
P | Q
```

where `P` and `Q` are procedures - of compatible type. These can be names of procedures or procedure clauses.

8.3 Procedure application

A **procedure application statement** is of the form:

```
PROC ARGUMENT
```

where `PROC` is a procedure valued expression of compatible type to the type of `ARGUMENT`. Of course, the procedure will typically have several arguments, in which case these are encoded in a tuple as `(A1, ..., AK)`.

8.4 Procedure expressions

Like functions, a `procedure` is a first class value in April. This implies that a procedure - defined using a combination of clauses and procedure union operators - can be used as a value in an expression.

The compiler determines by context the situation where a procedure is being used as a value - for example, in the statement:

```
('doThis, {() -> "Hello world\n" >> stdout}) >> execServer}
the expression {() -> "Hello world\n" >> stdout} is a procedure value.
```

The type of a procedure value `PTN -> STMT` is an expression of the form `TP{}`, where `TP` is the type of `PTN`. Where a procedure is defined using the union operator, the type is found by unifying the types associated with the two arms of the procedure union.

There is a `fail` procedure.

8.5 Programs and theta expressions

A `theta expression` is a special form of `record expression` that provides a syntactic environment for declaring programs. The form of a theta record expression is:

```
{EL1; ... ELK }
```

where `ELI` are declarations, of types, functions, procedures or variables.

Semantically a `theta record` has two views: internally it is a binding environment in which recursive programs may be declared; and externally it is equivalent to a record - a record of all the program elements declared in it.

8.6 Function declarations in theta expressions

A `function declaration` in a `theta record` takes the form of a labelled set of equations:
{ ... `FOO = FUNCTION ...` }.

For example, the list `append` function may be defined using:

```
append = { ([], ?X) => X
           | ([?E, ..?R], ?X) => [E, ..append(R, X)] }
```

8.7 Procedure declarations in theta expressions

Within a **theta record**, a procedure can be defined as a labelled mutually recursive set of clauses or, where there is only one clause, with a notation that is more reminiscent of "C".

The clausal form of procedure takes the form

```
{ ... P = { CLAUSE1 | CLAUSE2 | ... } ... }
```

For example, an exec server process procedure may be defined using:

```
exec = { () -> { repeat{ ('dothis, P) ->> P() } until quit }
```

Procedures are only allowed to change the values of variables declared within the procedure itself and any theta variables that are in its scope.

8.8 Theta variables

An **environment variable** is a **variable declaration** that takes place at the level of other procedures and functions within a **theta record expression**. The initial value of a **theta variable** is determined by the right-hand side of its declaration. However, while the initializing expression of a **theta variable** can be any expression - possibly involving references to functions also defined within the same theta record expression - the programmer should be careful to avoid circular execution problems. The order of evaluation of variable initializations is not defined; therefore the programmer should not make any assumptions about the values of other **theta variables** - including the variable itself - when determining the initial value of a given theta variable.

A **theta expression** containing one or more **theta variables** may be sent in a message to other processes - either on the same host computer or on other host computers. However, the precise semantics of this differs in the two cases.

In the case that a **theta expression** is shared by two or more processes within a single April invocation; each of these processes can potentially invoke procedures within the **theta record** that will side-affect embedded variables. This can lead to severe complications with respect to simultaneous access to environment variables by the different processes.

In the case that a **theta expression** is sent to a process on a different host computer - or a different April invocation on the same computer - then the values of theta variables are duplicated and become, in effect, separate variables; although they are still internally linked as a unified theta record on the remote host.

8.9 Semantics of modules

April's `module` system does not rely on names of functions and procedures to link together exported and imported procedures. Instead we regard a `module` as a `theta expression` which is effectively a record of procedures, functions and types; accessing an imported procedure is achieved by accessing the correct member of the record. The principal benefit of not using names to link procedures and functions together is that it is very straightforward to embed programs in messages and data structures that are communicated between April applications.

Since April `modules` are based on `theta expressions`, they are actually first class values and as such can be assigned to variables and returned as values of functions. This would allow a module to be sent in a message for example.

9 April processes

Processes in April are a key abstraction. It is a process-oriented language that can be used to construct agent systems as well as other distributed applications.

A process is identified by its handle; a handle is a standard type in April which looks like:

```
handle ::= handle(string?tgt,string?name,  
string?home=host(),string[]?locs=location());
```

The handle of a process contains sufficient information for the April engine to be able to deliver messages to the process, as well as other internal attributes.

9.1 Spawning a new process

`spawn STATEMENT => handle` - normal spawn.

A new process is created to execute the STATEMENT concurrently with other processes. The handle of the new process is returned as the value of this expression.

```
spawn STATEMENT as handle?H=> handle - spawn with a public name  
spawn STATEMENT as AGENT
```

`spawn STATEMENT using handle?F=> handle` - spawn with a different file manager

```
H = spawn STATEMENT using sub_manager(DIRECTORY,[_allow_read])
```

The new process will be spawned with DIRECTORY as its current directory

`spawn STATEMENT error_link handle?E=> handle` - spawn process with non-standard error handler

```
H : spawn STATEMENT error_link HANDLE
```

`spawn STATEMENT managed_by handle?M=> handle` - spawning process with non-standard resource handler

```
H = spawn STATEMENT managed_by PROCESS_HANDLE
```

Combining different non-standard process managers

```
H = spawn STATEMENT using DIRECTORY link_error HANDLE
```

9.2 Example of an April process

```
fact_server()
{
repeat{
('arg_is, N)::N>=0 ->
    ('result_is,fact(N)) >> replyto
}
until 'quit
};
H= spawn fact_server();
```

The procedure `fact_server` consists of a `repeat` statement containing one procedure. This procedure has a pattern `('arg_is, N)` with a guard, `N>=0`. If this matches a message in the queue, it generates a reply message by evaluating the tuple expression `('result_is,fact(N))`. This calls function `fact` to compute the factorial of `N`, `facn`. The message is thus `('result_is,facn)`, and this is sent to the process identified by the reserved identifier `replyto`. This has been set, by the message matcher, to the handle of the sender of the message matched, A process is created by the `spawn` operator by evaluating the procedure call `fact_server()`, assigning the new handle to the variable `H`.

```
rev_fact_server()
{
repeat{
('fact_arg_is, ?N)::N>=0 ->
    ('fact_result_is,fact(N))>>replyto
|
('rev_arg_is,?L) ->
    ('rev_result_is,rev(L))>>replyto
}
until 'quit::sender==creator()
};

H= spawn rev_fact_server();
```

This server provides factorial and `rev` functions to other processes.

9.3 Determining process status

`state(handle?P) => _process_state`

This function returns the status of the identified process using one of the following symbols:

`dead` - if the process identified has completed execution - or if it never existed.

`quiescent` - between the process being created and its first instruction.

`runnable` - if the process is potentially (or actually) running.

`wait_io` - if the process is waiting for an I/O operation to complete or become available.

`wait_msg` - if the process is waiting for a message.

`wait_timer` - if the process is waiting for a timer event.

`wait_clicks` - if the process is waiting for an additional allocation of CPU time.

`wait_memory` - if the process is waiting for an additional allocation of CPU memory.

`done(handle?P) => logical` - test for process termination.

9.4 Process control functions

`kill(handle?P) {}` - kill local process.

`waitfor(handle?P) {}` - wait for process termination

This procedure suspends and waits for the process P to terminate.

`halt {}` - halt execution of April

Exits the April system. This is a privileged procedure - and may not be used by a non-privileged process.

`abort()` - abort current process

`exit(number?CODE) {}` - this procedure terminates the current process immediately.

`CODE` is an integer. Exits the April system with the given code. This is a privileged procedure - and may not be used by a non-privileged process.

`delay(number?TIME) {}` - suspend process for a fixed time

The process suspends execution for `TIME` seconds.

`sleep(number?TIME) {}` - sleep until some time

The process suspends execution until at least `TIME`; where `TIME` is an absolute time - which is typically some offset from the value returned by the `now()` built-in function.

9.5 Identification Functions

`host()` => `string` - identify host machine name.

`location()` => `string[]` - identify location of host computer. The location function returns a list which represents the location address of agents running on this machine. This facility is intended to allow processes to express handle values with complex routing requirements (such as for mobile agents, or agents executing on mobile computers).

`self()` => `handle` - handle of current process.

`creator()` => `handle` - creator of current process .

`commserver()` => `handle` - identity of the communications server.

9.6 Standard processes

When the April engine is initiated - by invoking the April command - a number of standard processes are also started automatically:

The **standard system monitor** - This process is responsible for monitoring system error conditions and for managing process termination.

The **standard file manager** - This process provides initial access to the file system. By default, the standard file manager gives the same access rights to the April program as the operating system itself. However, subsidiary file managers may give restricted access to the file system.

stdin, stdout and stderr - The processes corresponding to the standard input, output and error channels of the Unix process.

The **standard mail input reader** - This process maintains the connection to the ICM communications server. Whenever a message arrives from there, the standard mail reader forwards the message to the appropriate internal process - if it exists.

The **standard mail dispatcher** - When processes send messages the April engine delivers the message if the recipient process is in the same invocation as the sender. Otherwise, the engine delivers the message to the standard mail dispatcher which will (normally) upload the message to the communications server.

The **standard resource manager** - This process allocates memory and CPU cycles to processes. The standard resource manager simply gives additional memory and/or cycles to each process that requests them. A non-standard resource manager may be more thrifty: only giving resources to processes representing owners who have paid for them.

10 How April programs communicate

Interprocess communication is transparent in April - from the point of view of an April program, there is no difference between sending a message to a local process and sending a message to a process that resides on a different machine - or in a different network. In order to implement this transparent communications mechanism, April uses a communications server when sending messages between different machines; April relies on the InterAgent Communications Model (ICM) for this purpose.

In an April program, processes send messages to each other via a simple message send statement

```
('hello, X+34) >> yoohoo
```

The process of sending a message between processes within a single April invocation is very straightforward. If the message is destined for another process inside the same April invocation, the execution engine simply puts a pointer to the message in the new process's message queue. The situation is more complex when the message is sent to a process in a different host machine (or even on the same host machine but in a different invocation of April. In the latter case messages are routed using the InterAgent Communications Model which is separately documented in ICM Reference Manual.

10.1 The communicative process

Messages between April invocations are handled by a combination of April's socket library and the communications server. The process of sending such a message involves a number of steps:

1. When an April program sends a message, the underlying April machine attempts to deliver that message to a local process. If the recipient of the message is a local process, then the message is handed over to the recipient, and the message send terminates.
2. Each April process that is executing in the April machine has associated with it a mailer process. Normally, all the processes that form a particular application share a common mailer process, but this is not necessary.

If the recipient of a message is not a local process, then the message is given instead to the process's mailer process. The mailer process is the only process with the authority to establish connections to other host computers.

3. During the initialization phase, the main mailer process is started. Its first action is to establish a connection with a copy of the communications server; which normally runs as a separate Unix process on the same host as the April application itself. This connection is established using April's `tcp_connect` primitive.

When the mailer process is handed a message to deliver, it sends it to the communication server - using the `fencode` procedure

4. The communications server is typically shared by a number of different April invocations and other processes using the ICM's API. The communications server reads messages from all its clients and distributes the messages according to their intended destinations.

Like the main April engine itself, there are several possibilities for handling a message: either the message is intended for another process running on the same host machine, or it is intended for another host computer. The final possibility is that the message is intended for the communications server itself.

If the message is for another April process running on the same host computer, the communications server sends the message to that April engine.

5. The April mailer process is actually composed of two processes - one for sending messages to the communications server, and one for reading messages from the server. When the communication server sends the April engine a message, the reader half of the mailer process reads the message - using `fdecode` (*note `fdecode::`.) - and then hands over the message to the final recipient of the message using the same mechanism that is used for purely internal message passing.
6. If a message is destined for another host computer, then the communications server attempts to establish contact with the April communications server running on the remote host. Note that this could be anywhere on the Internet - the same mechanism that operates for a local area network also operates on the global scale.

Although this process may seem complicated, it has a number of important features:

- Messages within a single April invocation are handled very efficiently.
- Message traffic between invocations is controlled by both an output gate and an input gate - written in April. This allows certain security checks to be implemented on messages - such as verifying that any embedded code is safe to execute.

- The fact that different April processes may have different mailer processes associated with them means that it is possible to create a tailored environment for the execution of certain processes. It is possible, for example, to restrict certain processes (including any processes forked from the restricted processes) to communicate only with certain other hosts (or none)

10.2 The communications server

The communications server is used to allow messages to be sent between different April processes - either on the same computer but in different April invocations, or on different computers.

While a normal April application programmer does not need to be aware of the extra complexity involving this communications server, programmers who intend to use the April communications API will need to be aware of some of the complexities.

For inter-agent communication where agents are on different platforms, April uses the InterAgent Communications Model (ICM). This is an independent communications network and transport protocol that supports many important aspects of inter-agent communication. The ICM and how to use it is described more fully in a companion reference manual - The InterAgent Communications Reference Manual.

It is possible to query the communications server from an April program. In order to find out if a particular agent is present, a `icmPingAgent` message can be sent to the `commserver()` handle:

where `AGENT` is a `handle`. After a while (and it might take a while if the agent is located on a host that is far away), the `commserver` will respond with either:

```
receive{ ('icmPingAgent, 'icmOk, AGENT)::replyto==commserver() - >> ... |
... }
```

if the `commserver` can find the agent or, in the case that the `commserver` cannot find the agent it will respond with:

```
receive{ ... | (icmPingAgent, 'icmFailed, AGENT)::replyto==commserver()
->> ... }
```

If the communications server reports that an agent is alive, it is not guaranteed that a given sub-process of the agent is also alive. Furthermore, the communication's server cannot know if an agent is still alive even immediately after a successful ping of that agent.

11 Example programs in April

11.1 A simple value server

Our first complete example is a value server. A publicly named process called `value_server`, stores values for other processes. To store a value a process sends a store message of the form `('store,Name,Val)` where `Name` is a symbol and `Val` is a number. To retrieve a value, a process sends a retrieve message of the form `('retrieve,symbol?Name)`. The value server keeps all the values it receives in a list of associations between `Names` and `Values`. In the program, we have a main procedure which spawns the value server and two other processes, one which provides values to the server and one which retrieves them. We show a trace of a run of the program.

```
program
{
  main(){
    H = spawn value_server() as handle(name="value_server");

    P = spawn value_provider();
    ('value_store_is,H)>>P;

    Q = spawn value_receiver();
    ('value_store_is,H)>>Q;
    waitfor(P);
    waitfor(Q);

  };

value_server()
{
  AVPairs:[];
  repeat
  ('store,Name,Val) ->>
    { "store message received by value_server \n">>stdout;
  AVPairs:=
    [(Name,Val),..
  {AVPairs reject (Name,_)}]}
  |
  ('retrieve,symbol?Name)::
  (Name,?V) in AVPairs ->>
```

```

        { "retrieve symbol message received by value_server \n">>stdout;
('value,Name,V) >> replyto}
    |
('retrieve,Name) ->>
        { "retrieve name message received by value_server \n">>stdout;
('no_value_for,Name) >> replyto}
    until
        'quit::sender==creator()
};

```

```

value_provider()
{
repeat
('value_store_is,H) ->> {
    "\nstoring value for bill as 20 \n" >> stdout;
    ('store,'bill,20) >> H}
until
    'quit::sender==creator()
};

```

```

value_receiver()
{
repeat
    ('value_store_is,H) ->> {
        "sending retrieve message \n" >> stdout;
        ('retrieve,'bill)>>H}
    |
    ('value,Name,Value) ->> "value of " ++ Name^1 ++ " is " ++ Value^1 ++ "\n" >> stdout
    |
    ('no_value_for,Name) ->> "no value found for" ++ Name^1 ++ "\n" >> stdout
until
    'quit::sender==creator()
};

```

```

} execute main;

```

```

Script started on Sun Apr 25 15:12:11 1999
groat-1: pwd

```

```
/home/cs101/software/i386-linux2/April/April
groat-2: icm -L-
04/25/99 15:12:22 - Failed to set up commserver
groat-3: apc value_server.ap
April compiler - 4.2.5
groat-4: April value_server.aam

storing value for bill as 20
store message received by value_server
sending retrieve message
retrieve symbol message received by value_server
value of 'bill is 20
Terminating groat#20031 with code 1
groat-5:
Script done on Sun Apr 25 15:12:47 1999
```

Various comments:

1. The value server is defined as a procedure and then a statement which is a procedure call is spawned and given a public handle. The handle is bound as the value of the variable H and sent to the other processes in messages. However we could have used its public name. `EXPRESSION >> value_server.`
2. The April program will terminate when its main procedure terminates. Therefore we used a `waitfor` statement to allow the processes to finish before the main procedure finished. Otherwise, the program would terminate without all the messages being processed.
3. The output to `stdout` is a string value obtained from a string expression. This uses `++` which is the string concatenation operator, and `^` which converts a nonstring value into a string value.
4. Before running an April program, `icm` must be running. If it is already running, the `icm` command will give an error message. You can test if `icm` is running by using the command `ilist`.

12 Running April

You first should start the communications manager icm, by

```
icm -L-
```

If an icm is already running on your machine, you will get a message, e.g.

```
groat-2: icm -L-
```

```
04/25/99 15:12:22 - Failed to set up commserver
```

Also, you can always list all the April agents currently running on your machine, including the icm process, by using the command

```
ilist
```

e.g.

```
groat-39: ilist
```

```
05/01/99 13:46:12 - Agents registered at groat.ugcs.caltech.edu:
```

```
  groat#31871@groat.ugcs.caltech.edu/[131.215.43.183]
```

```
  icmCommServer@bolivar.ugcs.caltech.edu/[131.215.43.204,131.215.43.183]
```

```
  value_server@groat.ugcs.caltech.edu/[131.215.43.183]
```

```
  groat#28760@groat.ugcs.caltech.edu/[131.215.43.183]
```

```
  list@groat.ugcs.caltech.edu/[131.215.43.183]
```

```
groat-40:
```

The icm usually runs all the time. It uses very little resources.

Then the April compiler is evoked by

```
apc program.ap
```

then to run the compiled code

```
April program.aam
```

To run April on several machines, an icm communications server should be running on each machine.

You should run at least one publicly named process on each machine, then processes can be migrated among machines.

A Arithmetic and mathematical functions

These functions take numerical arguments. Giving non-numeric arguments will result in errors. All arithmetic functions will accept either integral or fractional arguments. Internally, results of arithmetic expressions are stored as integers where possible.

A.1 Standard arithmetic functions

These functions implement standard arithmetic operations such as addition and multiplication. Some of these functions are actually polymorphic - i.e., they can take different kinds of arguments.

- Addition - Addition
- Subtraction - Subtraction
- Negation - Arithmetic negation
- Multiplication - Multiplication
- Division - Division
- Integer quotient - Integer quotient
- Remainder - Remainder
- Bitwise and - Bitwise and
- Bitwise or -
- Bitwise xor - Bitwise exclusive or
- Ones complement - Bitwise negation
- Logical left shift -
- Logical right shift -
- Absolute value - Absolute value
- Truncate to Integer - Truncate to integer nearest 0
- Integer ceiling - Integer ceiling
- Integer floor - Integer floor

A.2 Mathematical Functions

These functions give support for complex mathematical computations such as trigonometry.

- sqrt - Square root
- cbrt - Cube root
- pow - Power
- exp - Exponentiation
- log - Natural logarithm
- log10 - Decimal logarithm
- sin - Trigonometric sin
- cos - Trigonometric cos
- tan -
- asin - Trigonometric arc sin
- acos - Trigonometric arc cos
- atan - Trigonometric arc tan
- sinh - Hyperbolic sinh
- cosh - Hyperbolic cosh
- tanh - Hyperbolic tanh
- asinh - Hyperbolic inverse sinh
- acosh - Hyperbolic inverse cosh
- atanh - Trigonometric inverse tanh

A.3 Miscellaneous arithmetic functions

- ldexp - Raise a number by a power of 2
- frexp - Compute mantissa and exponent
- modf - Compute integer and fractional part of number
- rand - random number generation
- irand - random integer generation
- seed - seeding the random number generator

B String Handling

April supports strings in two main ways - by providing a suite of string manipulation functions such as `substring` and `string append` - and by a set of string patterns which allow strings to be parsed as part of a match statement or expression.

B.1 Basic string functions

The following functions are basic operations on strings. They form a simple spanning set of operations on strings, allowing them to be dismantled as well as being constructed from smaller pieces.

- `strlen(string?S) => number` - the length of a string
- `string?STR1 ++ string?STR2 => string` - concatenate strings
- `expand(string?STR,string?KEY) => string[]` - analyze a string into components. This function is used to analyze a string into components. The `STR` is searched for occurrences of characters from `KEY`. The returned value is a list of strings, each element consisting of a segment from the `STR` between points where a character from `KEY` occurs. For example, the value of: `expand("peter@lg.doc.ic.ac", "@")` is the list: `["peter","lg","doc","ic","ac"]`
- `collapse(string[]?WORDS,string?GLUE) => string` - collapse a list of strings into one string

This function takes a list of strings, and a glue string, and concatenates them all together into a single string. A copy of the string `GLUE` is inserted between each word in `WORDS` as it is copied into the returned string.

B.2 String formatting

The string formatting functions allow the construction of strings from other values.

- String formatting
 - `EXP ~ W` means format the value `EXP` as a string in `W` characters. `EXP` can be a number value, a handle value or a string value.
 - `EXP ^ P` means format `EXP` to precision `P`.

- `strof(EXPRESSION,PRECISION) => STRING` - convert a basic value into a string, `PRECISION` is an `INTEGER`.

The `strof` function formats a basic value into a string. It does not cope with all April values; however, it can format `number`, `string` and `handle` values.

- `int2str(NUMBER,BASE) => STRING` - convert an integer into a string
- `strpad(string?X,number?WIDTH,string?PAD) => string` - reformat a string into another string
The `strpad` function reformats a string value into another string. The length of the output string is given by `WIDTH` - if `WIDTH` is 0 then the output is the same as the input, otherwise it specifies the length of the output string.

B.3 Accessing elements of strings

These functions provide a mechanism for accessing characters and bytes from string values

- `ascii(STRING) => LIST` - convert a string into a list of ASCII codes
- `nthascii(STRING,NUMBER) => NUMBER` - extract the `n`th char from a string
- `charof(LIST) => STRING` - convert list of ASCII codes into string

B.4 Basic symbol processing

April offers some string matching patterns; however, for many applications this is not powerful enough. For example, April's string patterns are not powerful enough to construct parsers.

For this reason, April offers some facilities to convert between strings and lists of numbers or lists of symbols. In addition, there are also some functions for converting between numbers and single character symbols.

- Type coercion can be used to convert from a string to a list of symbols,
`symbol[] %% STRING`,
e.g. `SS = symbol[] %% 'foo'`;
- Converting a single character symbol to a number -
`sym2ascii(SYMBOL) => NUMBER`

- Converting a number to a single character symbol -
`ascii2sym(INTEGER) => SYMBOL`

C Lists and Sets

One particular form of April list is the set. An April set is simply a list like other lists; however, April supports a number of set-style operations which allow programmers to use lists as sets. We refer to this as the set interpretation on lists - to distinguish April's collect of set-style operators from a true (and expensive) set type.

C.1 List Operations

- `LIST # INTEGER => ELEMENT` - index list elements The list-indexing function accesses individual elements of a list. e.g. `A#i:=23`;
- `listlen(LIST) => INTEGER` - length of a list
- `front(LIST,NUMBER) => LIST` - front portion of a list
- `back(LIST,NUMBER) => LIST` - the last NUMBER elements of a list
- `head(LIST) => ELEMENT` - extract first element of list
- `tail(LIST,NUMBER) => LIST` - extract remainder of list after counting a NUMBER of elements from the front.
- `LIST <> LIST => LIST` - list append
- `[HEAD ,.. TAIL] => LIST` - construct new list, e.g. `['fred, 'jim, 'harry ,.. NAMES]`
- `(INTEGER..INTEGER) => LIST of NUMBERS` - generate a list of integers, e.g. `[1..4]` has value `[1,2,3,4]`.
- `iota(INTEGER,INTEGER,STEP) => LIST of NUMBERS` - generate a list of integers using a STEP argument.
- `LIST // FUNCTION => LIST` - map function over elements of a list e.g. `[1,2,3,4,5] // double` gives the result `[2,4,6,8,10]`
- `LIST \\ FUNCTION => VALUE` - reduce list.
This higher order operator reduces a list of values to a single value by successively applying a function to each of the elements. e.g. `[1,2,3,4,5] \\ ADD` is equivalent to the expression: `ADD(ADD(ADD(ADD(1,2),3),4),5)` where `ADD(A,B)` is some appropriate binary function.
- `sort(LIST) => LIST` - sort a list
This function returns the list containing the elements of the given list sorted in ascending order. The ordering used is the standard April ordering.

C.2 Pattern based operations on lists

April has two functions where a pattern is used to select elements from lists. These functions allow lists to be searched for matching elements - i.e., they allow a form of database interpretation on lists.

- $LIST1 \hat{/} PATTERN \Rightarrow LIST2$ - select elements from a list. This function returns the list containing all elements of $LIST1$ which match $PATTERN$.
- $LIST1 \hat{\ } PATTERN \Rightarrow LIST2$ - remove elements from list. This function returns the list containing all elements of $LIST1$ which do not match $PATTERN$.

D Tests and Conditions

These functions return values of type `logical`.

D.1 Term Matching

`PATTERN . = TERM => logical` - match condition

`ELEMENT in LIST => logical` - list membership condition

D.2 Comparisons

Not all values in April are strictly comparable - for example procedures and functions are not comparable. In addition, elements of a user-defined type are not comparable. However, ordinary values such as strings, numbers and even lists of these are comparable. The type-checker will insist that when comparing two elements, they have the same type.

D.3 The standard ordering relation used in April

If two values are of the same type, then the relative ordering is as follows:

- One number is less than another if it is numerically less. Note that we can transparently compare integer values and floating point numbers; e.g. $1 < 1.5 < 2$
- Symbols and strings can be ordered by comparing the characters in their name using the underlying character encoding scheme (Usually based on ASCII). One symbol or string is less than another if:
 - its first character is less than the first character of the second symbol using the underlying character ordering - or,
 - if the first characters are the same then the next character in each symbol is compared in the same way. This is repeated until two different characters are found and a result is decided - or,
 - if one symbol runs out of characters before a result is decided, the shorter symbol is deemed to be smaller.
- One tuple is less than another if either:

- its arity is smaller (i.e. there are fewer elements in the smaller tuple) - or,
- if the arities are the same, the element in one tuple (in a left-to-right sequence) is less than the corresponding element in the other tuple.
- One record is less than another if either:
 - its arity is smaller (i.e. there are fewer elements in the smaller record) - or,
 - its label is smaller - in the symbol encoding described above - or,
 - if the arities are the same, an element in one record (in a left-to-right sequence) is less than the corresponding element in the other record.

D.4 Comparison functions

Using this ordering, we can define the following comparison functions:

- `==::` - equality test
- `! =::` - inequality test
- `<::` - less than test
- `>::` - greater than test
- `<=::` - less than or equal test
- `>=::` - greater than or equal test

D.5 Logical Connectives

The logical connectives allow boolean combinations to be formed. Note that `&&` and `||` are conditional combinations: if the first element evaluates to false or true respectively then the second argument is not evaluated. While this is not strict in the mathematical sense, it makes perfect sense for a programming language, since it allows optimizations to be made by the compiler.

- `&&` - conjunction
- `||` - disjunction
- `!` - logical negation

E Input and Output

string I/O, file I/O and socket I/O are provided.

Contents

1	Introducing April	2
2	April data values and types	3
2.1	Primitive types	3
2.2	Compound types	3
2.2.1	The list type	3
2.2.2	The tuple type	4
2.2.3	The record type	4
2.3	Standard April Types	4
3	April expressions	6
3.1	Function application expressions	6
3.2	Syntax expressions	6
3.2.1	Valof expressions	6
3.2.2	Try...onerror expressions	6
3.2.3	Exception expressions	7
3.2.4	Collect expressions	7
3.2.5	Setof expressions	7
3.2.6	Case expressions	7
3.2.7	Conditional expressions	8
3.2.8	Typecast and type coercion expressions	8
3.2.9	Record expressions	8
3.3	Lists and set abstractions	9

4	Patterns and pattern matching	11
4.1	Primitive patterns	11
4.2	List and other compound patterns	11
4.3	Value extraction	12
5	April statements	13
5.1	Primitive statements	13
5.2	Compound statements	14
6	Messages	15
6.1	Sending messages between processes	15
6.2	Messages and their attributes	15
6.3	Receiving messages	16
6.4	Receiving message attributes	17
6.5	Message streams	17
7	Functions and procedures	19
7.1	Functions and procedures in April	19
7.2	Functions	19
7.3	Function equations	19
7.4	Function union	20
7.5	Function application	20
7.6	Built-in April functions and procedures	21
7.7	Examples of April functions	21

8	Procedures	22
8.1	Procedure clauses	23
8.2	Procedure union	23
8.3	Procedure application	23
8.4	Procedure expressions	24
8.5	Programs and theta expressions	24
8.6	Function declarations in theta expressions	24
8.7	Procedure declarations in theta expressions	25
8.8	Theta variables	25
8.9	Semantics of modules	26
9	April processes	27
9.1	Spawning a new process	27
9.2	Example of an April process	28
9.3	Determining process status	29
9.4	Process control functions	29
9.5	Identification Functions	30
9.6	Standard processes	30
10	How April programs communicate	31
10.1	The communicative process	31
10.2	The communications server	33
11	Example programs in April	34
11.1	A simple value server	34

12 Running April	37
A Arithmetic and mathematical functions	38
A.1 Standard arithmetic functions	38
A.2 Mathematical Functions	39
A.3 Miscellaneous arithmetic functions	40
B String Handling	41
B.1 Basic string functions	41
B.2 String formatting	41
B.3 Accessing elements of strings	42
B.4 Basic symbol processing	42
C Lists and Sets	44
C.1 List Operations	44
C.2 Pattern based operations on lists	45
D Tests and Conditions	46
D.1 Term Matching	46
D.2 Comparisons	46
D.3 The standard ordering relation used in April	46
D.4 Comparison functions	47
D.5 Logical Connectives	47
E Input and Output	48