Computing Science Technical Report No. 143

Newsqueak: A Language for Communicating with Mice

Rob Pike

August 16, 1993

Newsqueak: A Language for Communicating with Mice

Rob Pike

ABSTRACT

This is the reference manual for the revised Squeak language, a concurrent language designed for writing interactive graphics programs. The language is, however, much more generally applicable. This manual defines the language. Separate documents will describe the libraries and give a rationale for the design.

August 16, 1993

Newsqueak: A Language for Communicating with Mice

Rob Pike

This is an informal reference manual for the concurrent language Newsqueak. Newsqueak's roots are in Squeak, a language designed a few years ago by Luca Cardelli and Rob Pike to illustrate concurrent solutions to problems in user interface design. Newsqueak addresses the same problems but in a broader context: Squeak was for designing devices such as menus and scroll bars; Newsqueak is for writing entire applications, and in particular a window system. Besides a wholesale redesign of Squeak's syntax, Newsqueak therefore has several major components absent in Squeak: a type system, dynamic process creation, and dynamic channel creation. Also, Squeak deferred most mundane programming details to the language it compiled into, C. Newsqueak is instead a self-contained language. An interpreter for it, called squint, has been implemented.

Newsqueak draws heavily from CSP and C, and the discussion that follows assumes modest familiarity with both these languages. Roughly, the syntax and basic semantics come from C, while the message-passing primitives come from CSP. The way these are put together is, however, unique to Newsqueak.

1. Text

Input is free-form. An identifier consists of alphanumeric characters, including underscore, and does not begin with a number. A sharp character # begins a comment, which continues until a new-line character. Files may be included; an occurrence of

`include "`*file name*`"`

is replaced by the contents of the file. (In this manual, italic text distinguishes syntactic constructions from literal text.)

An `include` may occur anywhere the word `include` is recognizable as an identifier; `include` does not need to begin a line. If the file `twentythree` contains the text `23` then

```
x=include "twentythree";
```

sets `x` to 23.

If the named file is not found in the current directory, and does not begin with a slash or period, it is sought in a standard repository.

2. Simple Types

Three basic types exist:

`unit` is a type with exactly one entry, itself named `unit`. It prints as

```
(unit)
```

`int` is the basic signed integer type, typically 32 bits long.

`char` is an unsigned character, 8 bits long. `chars` and `ints` — the *integral types* — are interchangeable as in C, but, when printed, an `int` is a numeric value and a `char` appears as its output form (the letter A, new-line, etc.).

The syntax for integer and character constants is as in C. Character constants are of type `int`.

3. Compound Types

Four type constructors build compound types.

`array` builds an array of objects, indexed by integral values starting at zero. The syntax is

`array[`*size*`] of` *type*

where *size* is an integral expression that is evaluated to determine the number of elements. Thus the type

```
array[10] of int
```

is an array of ten integers, `a[0]` through `a[9]`.

The size of an array is not part of its type, so the size may change on assignment. If `a10` is an array of ten objects, and `a5` is an array of five, after executing

```
a5=a10
```

`a5` has ten elements. Moreover, the size may be omitted from the declaration (e.g. `array of int`) if the size is to be determined later or is implicit, for instance in the declaration of formal parameters.

`struct` builds a compound data structure:

`struct of {` *list of element declarations* `}`

where the list has the normal declaration syntax, described below.

`prog` declares a program, much as in lambda calculus. The syntax is

`prog(`*list of formals*`) of` *type*

where the formals may be empty, a single declaration, or a comma-separated list of several declarations. The final type is that of the expression yielded by the `prog` when it is executed. There is no distinction between functions and procedures; a procedure corresponds to a function returning `unit`. If the return type is `unit`, the `of` clause may be elided.

`chan` defines a bufferless communication channel. The syntax is

`chan of` *type*

Channels are discussed below, in the section on communications.

4. Strings

Arrays of characters are also called strings. String constants are written as in C, so

```
"hello"
```

is a string with 5 characters. Strings are not null-terminated, but rather have a known length. (See Section 8, on expressions.)

5. `mk`

Newsqueak has an object constructor, called `mk`:

`mk(`*type*`)`

returns an uninitialized object of the named type.

`mk(`*type* = *initial value*`)`

gives the object a value. For instance,

```
mk(int=10)
```

creates a storage cell holding the integer 10. For compound objects, `mk` creates storage for only the outermost type. Thus

```
mk(array[3] of array[2] of int)
```

creates a 3-element array of 2-element arrays of integers, but the 2-element arrays are still undefined, so attempting to access them will yield an error.

Some syntax allows more complicated initializations. The initializers for compound objects must be enclosed in braces, as in

```
mk(array[2] of int={1,2})
```

or, recursively,

```
mk(array[2] of array[3] of int={{1,2,3},{4,5,6}})
```

Any object of compatible type may be used in a `mk`. For example, the following two `mk`'s are equivalent:

```
mk(array of char={'h','e','l','l','o'})
mk(array of char="hello").
```

(The size of these arrays will be derived from the initializers.) The order of evaluation of initializers is undefined. (See also Section 7, on deriving type.)

6. Declarations

Declarations consist of a comma-separated list of identifiers, a colon, an optional type, and an optional equals sign and initial expression. For instance,

```
o:int
```

declares `o` to be an integer,

```
p,q:int
```

declares two more integer variables, and

```
nl:char='\n'
```

defines `nl` to be a character with initial value new-line. The declaration

```
p,q:int=1
```

is identical to the two declarations

```
p:int=1
q:int=1
```

except that the initializing expression is evaluated only once. Any part of the declaration may be elided if its form is clear. For instance, the following are all equivalent:

```
i:int=mk(int=100)
i:=mk(int=100)
i:int=100
i:=100
```

When compound objects are being created, the most convenient form is often

```
a:=mk(array[10] of int)
```

Observe that **:=** is *not* an assignment operator. It is two operators, one to declare and one to assign.

Variables may be declared constant by prefixing the declaration with `const`, so

```
const NBUF:int=200
```

declares `NBUF` to be the constant 200. It is an error to change the value of a constant object.

An identifier may be used to identify a type, using the `type` keyword. For example,

```
type point: struct of{ x, y: int; }
```

defines a data type to represent coordinates in Z×Z.

Statements, of which declarations are one form, are terminated by semicolons, so the declarations above must all be followed by a semicolon. This is why a semicolon appears in the declaration of type `point`.

Declarations may appear anywhere a statement is legal, except where the syntax explicitly mentions *executable statement.*

7. Deriving type

The type of an object may often be inferred. Newsqueak infers type in declarations and assignments, including the binding of actual parameters to formals in `progs` and in the values returned by `progs`. In declarations, therefore, the type may be left off if it may be implied from an initializing expression, as in the declaration from above

```
i:=100;
```

which declares `i` to be an integer with initial value 100. Given `i`, another variable may be declared using the value of i:

```
i_plus_one:=i+1.
```

and so on.

For compound objects, initializations may be provided by grouping the initializing expressions, element by element, in braces:

```
p:point={2, 3}.
```

If a type is given, it overrides the type of the initializing expression, which matters only in cases related to these:

```
c:char='\n'                  # '\n' is of type int
a:array[10] of int={1,2,3}    # remaining elements of a are undefined
b:array of int=1
```

Type is also derived in assignments (described below), so given

```
p:point
```

the assignment

```
p={5,6}
```

is equivalent to

```
p=mk(point={5,6}).
```

Similarly, if `f` is a `prog` that acts on `points`, it may be called as

```
f({2,3}).
```

When type may be derived, `mk` needs no explicit type. Thus

```
c:chan of int;
c=mk();
```

defines a new channel and assigns it to `c`.

It is an error to reference an undefined variable, except to assign a value to it. An undefined integer, however, has value zero, and it is not an error to access it.

8. Expressions

Expressions are syntactically and semantically much as they are in C. They include the assignment operator =, the arithmetic operators + − * / %, the bitwise operators & | ^ ~ << >>, the logical operators ! && ||. The . (dot) operator accesses elements of `structs`. Newsqueak does not have the `?:` operator or pointer indirection. It also does not have the augmented assignment operators += and its ilk. It does, however, have postfix ++ and prefix −−, which operate only on `ints` and are guaranteed to be atomic in their update. They are therefore useful for synchronizing shared variables.

The comparison operators == >= <= != < > may be applied to integral types, and (lexicographically) to strings, yielding zero for failure and one for success. As in C, zero is a false Boolean value and non-zero true.

Finally, there are several operators unique to Newsqueak. The first are the unary operators `def` and `len`.

```
def a
```

tells if the object `a` is defined, that is, if it has storage allocated. For example, given

```
a:=mk(array[2] of point);
```

`def a` yields 1 (true), but `def a[0]` yields 0 (false).

```
len a
```

for an array `a` tells how many elements (defined or not) the array contains.

There are two infix array operators.

```
a del n
```

where `a` is an array and `n` an integer, yields `a` with the first `n` elements dropped. Therefore

```
a=a del 1
```

shortens `a` by deleting the first element. If `n` is negative, the last `n` elements are dropped.

```
a1 cat a2
```

yields the concatenation of the arrays `a1` and `a2`.

The communications operator <− is discussed in Section 14, on communications.

9. `progs`

A `prog` expression is a body of executable code that may be assigned to a variable and executed. A `prog` expression is a `prog` type followed by a brace-enclosed body:

`prog(`*formals* `) of` *type* `{`*body* `}`

where *formals* is a comma-separated list of declarations of formal parameters and the *body* is a sequence of statements, defined below. The lone type is that of the resulting value, `unit` by default.

The value returned to the caller of a `prog` is produced by the `become` statement, discussed in the section on statements. Here is a `prog` that adds its two integer arguments:

```
prog(a,b:int) of int{
        become a+b;
}
```

To bind this program to a variable, declare one, say `add`, and assign to it:

```
add:=prog(a,b:int) of int{
        become a+b;
};
```

Now `add` may be called anywhere an expression of integral type is legal, using the traditional syntax, for example as

```
twoplusthree:=add(2, 3);
```

Note, however, that a `prog` is an expression, and may be called directly. Thus the declaration of `twoplusthree` may be equivalently written

```
twoplusthree:=prog(a:int, b:int) of int{
        become a+b;
}(2, 3);
```

10. `rec`

The `rec` keyword, prefixing a declaration or brace-enclosed group of declarations, permits declarations of self- and mutually-recursive variables and types. For example, the factorial function can be written

```
rec fact:=prog(n:int) of int{
        if(n==0) become 1;
        become n*fact(n-1);
};
```

A tree may be declared as a recursive type:

```
rec type tree: struct of{
        value: int;
        left,right: tree;
};
```

11. Data

Variables in Newsqueak, even compound objects, are manipulated entirely by value. For instance, after the declarations

```
a:="hello";
b:=a;
```

`a` and `b` have the same value, but they do not share storage. After the assignment

```
a[0]='j';
```

`a` now has the value `jello` but `b` is still `hello`. This rule applies to all handling of data, including passing variables to `progs`, so if a `prog` wishes to overwrite some elements of an array, it must arrange to return the array to its caller, using a global variable or by using an appropriate return (`become`) value. (For efficiency, the implementation only creates copies of variables when they are needed, so data are shared as long as possible, but the semantics is as described here.)

12. Scope

A variable, once declared, is visible until the end of the brace-enclosed block of statements in which it is declared. Its value may, however, be passed outside. A variable defined outside any block is a global object, as though the entire program were enclosed in a set of braces.

13. Statements

The simplest statement is the empty statement, a bare semicolon. It is equivalent to the empty compound statement `{}`.

A compound statement is one of

```
{}
{list of statements}
```

where a list of statements is a concatenation of one or more statements.

Any expression, followed by a semicolon, is a statement. For instance, assignment to a variable is an expression that yields the value assigned:

```
b=2
```

sets `b` to 2 and yields 2 as its value. This allows chains of assignments:

```
a=b=2;
```

is a statement that assigns 2 to `b` and then to `a`. (C handles this differently; it assigns 2 to `b` and then `b` to `a`.)

A declaration is a statement.

Control statements are much as in C. The `if` statement has form:

```
if(expression )
          executable statement
```

or

```
if(expression )
          executable statement
else
          executable statement
```

The expression (of integral type) is evaluated. If the result is non-zero, the first statement is executed; if zero and an `else` clause is present, its statement is instead executed. An executable statement is any statement except a declaration.

The `for` loop is

```
for(expr1 ; expr2 ; expr3 )
          executable statement
```

The first expression is evaluated. Then, while the second expression is non-zero, the statement is executed and the third expression is evaluated. Any of the expressions in parentheses may be elided; if the second is missing, it is taken to be 1.

The `break` statement,

```
break;
```

which is legal only within a loop, terminates the loop immediately. The `continue` statement,

```
continue;
```

transfers the loop control immediately to the third expression, bypassing the remaining body of the loop.

The `while` loop

```
while(expression )
          executable statement
```

is equivalent to

```
for(;expression ;)
          executable statement
```

The `do` loop

`do` *executable statement*
`while(`*expression* `);`

is equivalent to

executable-statement
`while(`*expression* `)`
    executable statement

The `switch` statement

`switch(`*expression* `){`
`case` *expression* `:`
    list of executable statements
`case` *expression* `:`
    list of executable statements
`default:`
    list of executable statements
`}`

evaluates the first expression, and then compares the value top-to-bottom with the expressions (which need not be constants) in the various cases. When a particular `case`'s expression equals the `switch` expression, the corresponding statement list is executed, and execution then continues after the `switch` statement. At most one case is executed. If no earlier `case` matches, and a `default` is present (it is optional), its statement list is executed. Because strings may be compared using the logical operators, `switch` expressions may be strings as well as integers.

The `become` statement

`become` *expression* `;`

is legal only inside a `prog`. Its effect is to replace the execution of the `prog` by the evaluation of the expression. The resulting value is returned to the caller of the `prog` as its value. For example, consider the `prog`

```
rec sumorial:=prog(n,sum:int) of int{
        if(n==0) become sum;
        become sumorial(n-1, sum+n);
};
```

If `n` is zero, the first `become` yields to the caller the value of `sum`. Otherwise, the second `become` replaces the executing `prog` by a version of itself with different arguments. In general, when a `prog` `P` executes a `become` whose expression is a call of `Q`, the effect is exactly as if the caller of `P` had instead called `Q` with the appropriate arguments. No new stack space is consumed; `become` is a form of process call.

The `begin` statement starts a process — an independently executing computation. The syntax is

`begin` *prog* (*parameters*);

so it may begin only a `prog` as a separate process. (The return value of the `prog` is discarded if the `prog` finishes.) All processes execute concurrently; the current implementation interleaves the execution of processes very finely. Processes are discussed further in the next section.

14. Communications

Newsqueak's communication mechanisms are as in CSP, with channels acting as rendezvous points for processes. Channels have explicit type:

```
c:=mk(chan of int);
```

creates a channel `c` that may be used to send integers.

The communication operator is `<-` and exists as a prefix and postscript form. The prefix form is a receipt:

```
<-c
```

yields the next value transmitted on the channel. The postfix form is legal only on the left side of an assignment:

```
c<- = 3;
```

sends the integer 3 on `c`.* (If the type of the channel is `unit`, the assignment may be elided; `unitchan<-` sends `unit` on such a channel.) For example, given two integer channels `c1` and `c2` the loop

```
for(;;) c1<- = <-c2;
```

receives integers on `c2` and sends them out on `c1`.

For a communication to complete, two distinct processes must simultaneously be able to communicate on the stated channel, one to send and one to receive. Given a channel `c` shared by two processes, when one process executes

```
c<- = 3;
```

it blocks (suspends execution) until another process executes

```
<-c;
```

and vice versa. When both are ready, the sending process evaluates the expression to send, and sends it to the receiving process. Both processes then resume execution. Note that the rendezvous is done before the expression to be sent is evaluated, so in

```
c1<- = <-c2;
```

the rendezvous on `c1` occurs before the rendezvous on `c2`. If the other order is required to avoid deadlock, the statement can be rewritten as

```
{a:= <-c2; c1<- = a;}
```

The `select` statement allows a process to choose among a set of channels on which to communicate. The syntax is

```
select{
case communication :
        list of executable statements
...
}
```

There is no `default` in a `select`. The communications in the `case`s are one of the following:

```
<- channel
variable = <- channel
channel <- = expression
```

where a channel is either a simple channel or an array expression of one of the forms `a[]` or `a[`*variable* `=]`, and `a` is an array of channels and *variable* is an assignable object. In the array forms, all the channels in the array are made available for communication. If the last form is used, and that case proceeds, the index in the array of the channel that communicated is assigned to the indexing variable.

---

* Mnemonic: `c<-` is a send because the arrow points to the channel; `<-c` is a receipt because the arrow points from the channel.

The execution of a `select` proceeds as follows. First, all the channels named, including all the channels in arrays, are evaluated in the order listed. If one or more channels are free to communicate immediately, one of those available is selected at random (by calling a pseudo-random number generator) and the corresponding communication proceeds. Otherwise, the process blocks until one or more communications can proceed. When some can, one of those available is selected at random, and the appropriate `case` is selected. If it is a send, the expression to be sent is next evaluated. If the `case` is an indexed array, the index is then assigned. The value is then passed as appropriate. If the communication is a receipt with a variable named to receive the value (the case `a=<-c`) the variable receives the value sent. Execution then proceeds with the statement list attached to the successful case, after which execution resumes after the `select` statement.

For example, the following `prog` prints the value and array index of the first receiving communication to proceed on any of the channels in its argument:

```
prog(a:array of chan of int){
        i,v: int;
        select{
        case v=<-a[i=]:
                print("chan index ", i, " result ", v, "\n");
        }
}
```

(The `print` expression is discussed below.)

15. `val`

The `val` expression has syntax

`val {` *list of statements* `}`

One or more of the statements in the statement list is a `result` statement:

`result` *expression* `;`

The semantics of `val` is to execute the statement list, and to yield the value of the expression associated with the first `result` statement executed. It is an error to complete a `val` statement without executing a `result` statement. For example, this sets `c` to the maximum of `a` and `b`:

```
c = val{
        if(a>=b) result a;
        result b;
}
```

16. Printing

The `print` expression, with syntax

`print(`*list of expressions*`)`

where the expressions are comma-separated, yields a string containing formatted representations of the expressions, concatenated. If this expression is promoted directly to a statement, without being stored or evaluated, it is written to the standard output. For example:

```
print("23+45=", 23+54, "\n");
```

prints

```
23+45=77
```

but

```
x:=print("23+45=", 23+54, "\n");
```

saves the resulting string in `x` and produces no output.

17. The interpreter

The Newsqueak interpreter, squint, implements the language described above. Its input is a list of statements, executed in the order presented. Expressions at the top level of the interpreter, when promoted to a statement by a following semicolon, have their values printed automatically, so the interpreter may be used as a sort of calculator. A typical program defines several `progs`, then `begins` them and starts some communication.

18. Example

The following program comprises several `progs`. The last, `sieve`, returns a channel of integers that produces the successive prime numbers.

```
counter:=prog(c:chan of int)
{
        i:=2;
        for(;;)
                c<-=i++;
};
filter:=prog(prime:int, listen,send:chan of int)
{
        i:int;
        for(;;)
                if((i=<-listen)%prime)
                        send<-=i;
};
sieve:=prog() of chan of int
{
        c:=mk(chan of int);
        begin counter(c);
        prime:=mk(chan of int);
        begin prog(){
                p:int;
                newc:chan of int;
                for(;;){
                        prime<-=p=<-c;
                        newc=mk();
                        begin filter(p, c, newc);
                        c=newc;
                }
        }();
        become prime;
};
prime:=sieve();
```

This program may be run by typing

```
$ squint
include "sieve"
<-prime;
2
<-prime;
3
i:int;
for(i=0; i<10; i++) print(<-prime, " ");
5 7 11 13 17 19 23 29 31 37
```

(`$` is the prompt from the Unix shell, and output is in italics.) The interpreter may also be invoked

```
$ squint sieve /dev/stdin
```

19. Implementation bugs

The current implementation does not allow local variables to exist past the lifetime of the `prog` in which they were created. This causes problems when local progs are created that access variables neither global nor local to themselves. The situation is flagged as an error. For example,

```
bad1:=prog(){
        a:int;
        p:=prog(){
                a=1;
        };
};
```

draws an error. To mitigate the problem somewhat, squint internally rewrites the program

```
bad2:=prog(){
        a:int;
        begin prog(){
                a=1;
        }();
};
```

into the form

```
bad2:=prog(){
        a:int;
        begin prog(a:int){
                a=1;
        }(a);
};
```

since the argument list for a `begin` is always present. Thus local `progs` not begun as processes must access only globals and variables local to themselves.

20. Future

A couple of things should be tidied up: type derivation should be formalized, and the array forms of communication should be applicable outside select statements.

Developments being considered for Newsqueak include buffered channels and environments. Buffered channels would have syntax

```
chan[10] of int
```

with the number defining the size of a buffer. A synchronous channel, as currently implemented, would be equivalent to

```
chan[0].
```

Environments are a way to specify the variables accessible to a body of code. They make it possible to maintain strong typing while allowing arbitrary strings to be accepted as programs. In other words, they make a sort of `eval` or compile-on-the-fly operator type safe.