Necessary Information about this paper.
Latest revision: June 6, 1973

(The permanent names of this file are
SMALLTALK.DC. ***
SMALLTALK1.DC.
SMALLTALK2.DC.
Its latest incarnation will always be found on the
Learning Research Group Demo Diskpack.

The full structured index is found with each version.
Look under the structure to discover what file to load.

This file should be displayed using font SROMAN.FD.
To print, edit with SMDELEG.FD and Write Translated,
then print on XGP using font SMDELEG.XG)

SMALLTALK, a Model Building Language With Intensional Semantics

by
Alan C. Kay

Learning Research Group
Xerox Palo Alto Research Center

Abstract

SMALLTALK is a language which allows children (and adults) to build semantic models of their ideas in simple uncomplicated ways, and dynamically simulate them with respect to arbitrary environments.

Simplicity is achieved by having

a. only one kind of object in the language (a process) which can act like all other known computer objects,
b. a single uniform scheme for interobject communication, and,
c. an intensional semantics in which the meaning of an object is a part of the class to which an object belongs rather than dispersed through the system as part of more conventional extensional operations.

Benefits are the abilities to create new "functional", "data", "control", etc., entities without the usual problems associated with updating and coercion of generic functions.

****

Acknowledgements

The main influences on the content of this paper were the coprocess and data/function equivalences of FLEX[ka-68,69], Flex's influence SIMULA [Dahl, et.al.], LISP [McC,et.al.], a number of control ideas of Dave Fisher[fi-70], goals as expressions found in Carl Hewitt's PLANNER[he-70], and the simplicity and ease of use of LOGO[pa,et.al-67,...73].

Dan Ingalls of LRG in PARC, the implementer of SMALLTALK, has revealed many design flaws through his several excellent quick "throw away" implementations of the language. SMALLTALK could not have existed without his help and good cheer.

Introduction

SMALLTALK is built from a few simple, yet powerful, ideas.

First, SMALLTALK considers every OBJECT in its world to be an independant entity with local state and control. All distinction between "datalike" and "procedurelike" objects, such as exist in other programming languages, is thus removed. This includes "data", such as numbers, strings, arrays, lists, structures, etc.; "functions", such as 'factorial', 'plus', 'print', etc.; "control structures", such as conditional branches, repeats, recursion, and so on; "IO devices", such as 'files', 'the user', 'display and keyboard', etc.; all are treated alike because they ARE alike.

Next, all objects are composed of PARTS, even if they only contain themselves. The object can be thought of as a dynamic dictionary which contains all the relations and rules in which it can take part.

Third, objects can send and receive MESSAGEs to/from other objects. This may cause new objects to be created, altered, or even destroyed.
(Since there are no "special" objects, there is only one message protocol.)

Finally, each object is considered to be a member (or INSTANCE) of a CLASS, which is another object that contains the rules of behavior shared by all the members. Since each class has a class defining object, they are members of the class of class-defining-objects, as one might expect.

Messages

A message is a stream of zero or more symbols.
If the stream starts with an open parenthesis, its closing parenthesis absolutely terminates the stream.

An embedded "." at the same level will terminate the current message and will cause the message following it to be sent.

If the message is composed of parts whose termination is ambiguous, a "," can be used to clarify matters.

Sending is done from left to right using a very simple rule: control is passed immediately to the first object encountered in the stream, along with information about the context of the send. This is all the EVALuator does. The receiver may gather in the message in any way it chooses.
A common first object is an instance of the class "name" (as with a LISP atom, all of its members start with a letter and are composed of letters, digits , underscores, and other special characters).

The action of a name is to look itself up in the current environment/dictionary to see if it has a meaning (which is another object). If it does, that object is RETURNed by APPLYing it to the remainder of the message;--- And so it goes until the message is consumed.

A venerable example: factorial.

A message
factorial 3.
is sent in the following manner.

Control is passed to the name "factorial" which looks itself up in the current environment and finds another object as its value. The new object is a class defining object which contains the rules for all the instances of the class "factorial":

```
:n. ↑ if n = 0 then 1 else (n *  factorial n - 1).
```

The action of the class defining object is to create a new instance of factorial and APPLY it to the message.

The ":" is a "receive" (or "input") object whose action is to EVALuate the input stream (in this case "3", whose value is "3") and then to make a new entry into the local environment to define the name (in this case "n"). After this a lookup of "n" will have the value "3".

The "↑" is a "send" (or "output") object which will APPLY the EVALuation of its argument to the remainder of the message found in the CALLER's object.
The next message is sent by finding "if" which tries to receive the message consisting of the EVALuation of "n=0".
Control is passed to "n".
It looks itself up and finds "3".
Control is passed to it.
"3" is an instance of the class number which has many relations it can respond to.
"3" receives the next object (unevaluated) to see what it is. (It could be any of +, -, * /, < , > , etc.; in this case it is "=").
"3" wants now to evaluate the next part of the message in order to see whether to RETURN "true" or "false".
Control is passed to "0" which, as with "3", is an instance of class number, and thus shares the same relations.
So, it looks to its right to see if anything like +, -, *, etc., is there which it can respond to.
It finds only "then" for which it has no meaning.
So it RETURNs ITSELF to "3" which now has enough info to decide "not true"
which is RETURNed to "if" which decides not to evaluate the message following "then", but does try to evaluate the message following "else".
"n" looks itself up, finds the value "3"
which picks up the name "*" for which it has a meaning.
So "3" tries to evaluate the next part of its message "factorial n - 1)".
Control is passed to "factorial" which looks itself up and discovers (as before) a class-defining object with the rule:

:n. ↑ if n = 0 then 1 else (n * factorial n - 1).

As before, a NEW instance is created which will try to evaluate the message "n - 1)" to get a new value for ":n".
"n" in the OLD environment looks itself up and discovers "3"
which looks to its right and finds "-" so it tries to evaluate the next object "1"
which which looks to its right and finds ")" (which terminates any message to "1")
so it RETURNs ITSELF to "3"
which knows how to subtract "1"
which causes a new instance of class number to be produced for the result "2"
which is RETURNed to the ":" in the CURRENT instance of "factorial"
which will enter it as a value for "n" in the CURRENT environment.
And so it goes.

The preceding rather long winded explanation of a well known example illustrates a number of important points.
First, although the terminology seems to be more general than is needed, a simple program in SMALLTALK looks simple and can be discussed in simple terms.

Second, only one rule of correspondence is needed to link form and content. The evaluator ONLY needs to know how to pass

control and context to an object. All other meanings are found distributed with the objects in the system. As shown, even such a seemingly primary act as creating a new instance is done by an object and thus can be changed at the user's whim.

Third, there are many cases where this generality of approach pays off handsomely. If we want to trace the activities of a name (such as "n" in instance 1) we need only create an object which can replace "3" as a meaning (so control will be passed to IT when "n" is touched), AND has a local entry of its own for "3" so that the meaning of "n" will not change with respect to its input/output characteristics. This means that an object can simulate any other object.

Fourth, all "relations" and "operators" (such as <, >, +, *, =, etc.) can be defined "intensionally" (or "intrinsically") as parts of an object or object class, rather than "extensionally" (or "extrinsically"), as is usually the case, as global functions.

In fact, "factorial" could have been defined this way as an intensional relation of a number. We might then have said "3!" and the class number would know what to do.

This means that the information pertaining to a class and what its members do need only be stored with the class. No global operations need to be updated. So, a class may be deleted without changing the rest of the world.

Also, this is a very convenient way to handle problems that arise from having multiple classes with operations: such as coercions between classes and the various senses of "fetch" and "store" ("←").

For instance, the message "a ← 3 + 1" means:
pass control to "a" which will look itself up and
pass control to the object it finds
which can gather the rest of the message as it pleases.
It can look to see if the next name is a "←", if so, it can EVALuate "3 + 1" and decide how to store it.

So "b 1 ← 81" , if "b" were an instance of an array, could mean
'store 81 in the 1st position'; or
if "b" were an instance of a hash table routine, could mean
'associate the hash of "1" with "81" in some way',
etc.

The problem of coercions will be discused a bit further on.

Fifth, instances may be EVALuated "concurrently" using the very same EVALuation strategy. Here, the generality of message send/receive becomes much more important.

Class Definitions Already in SMALLTALK

‹See SMALLTALK1.DC for this branch›

Some SMALLTALK Programs
‹See SMALLTALK2.DC for this branch›

Necessary Information about this paper.
Latest revision: June 6, 1973

(The permanent names of this file are
SMALLTALK.DC.
SMALLTALK1.DC. ***
SMALLTALK2.DC.
Its latest incarnation will always be found on the
Learning Research Group Demo Diskpack.

The full structured index is found with each version.
Look under the structure to discover what file to load.

This file should be displayed using font SROMAN.FD.
To print, edit with SMDELEG.FD and Write Translated,
then print on XGP using font SMDELEG.XG)

SMALLTALK, a Model Building Language With Intensional Semantics

by
Alan C. Kay

Learning Research Group
Xerox Palo Alto Research Center

Abstract

<See File SMALLTALK.DC for this branch>

Acknowledgements

<See File SMALLTALK.DC for this branch>

Introduction

<See File SMALLTALK.DC for this branch>

Messages

<See File SMALLTALK.DC for this branch>

Class Definitions Already in SMALLTALK

SMALLTALK is supplied with many useful classes, including quite a few found in one way or another in other programming languages.

These definitions are written in SMALLTALK as though they were not primitive objects. In some cases (such as the definition of "if") a primitive must be used to describe itself---which causes some obscurity.

Input and Output Objects

Informally (i.e.---more readable)

: Input a Value

followed by a name will evaluate the input stream to produce a new object which will be bound to the name.

This is exactly the same as LOGO.

Example; :value
will bind the result of evaluating the input stream to "value"

↑ Input a Token

followed by a ‹name› will not evaluate the input stream but will bind the next object there to the ‹name›.

There is no equivalent for this in LISP or LOGO, it acts as though the next input object were quoted.

Example; ↑value
will bind the next input object to "value"

⌷ Check Input for a Token

followed by a ‹name› will check the input stream to see if an identical ‹name› is there. No evaluation will take place. The Input Stream Pointer (or Program Counter) will NOT be advanced if the match fails. If the match succeeds, the ISP will be advanced to the next position.
This is used frequently to check for "operator" tokens such as +, *, and ←.

Example; ⌷+ will check the input stream for a "+" and will return TRUE if successful

⁝ Input Literal Stream

followed by a ‹name› will bind a reference to the Input Stream at the current point.
This is equivalent to FEXPR in LISP 1.5 or NLAMBDA in BBN-LISP.

Example; ⁝value will bind "value" to the input stream. EVALuation of this fragment may be delayed until later.

‹Other Input Objects›

will be mentioned here in a later version of this memo. An object to EVALuate a sequence of the input stream (like EVLIST in LISP) will probably be included at the very least.

! APPLY-RETURN a value.

This output object is used when when a subroutine control structure and message passing discipline is desired. Its single argument is EVALuated in the CURRENT environment and then APPLYed to the program stream of the CALLER process to which CONTROL also is RETURNed.
When used in "left nested" argument gathering (for example x.first.last or (A + B) + C), APPLY-RETURN will continue the evaluation process.

↑ PASSIVE-RETURN a value.
The single argument is evaluated in the CURRENT environment and RETURNed to the CALLER along with CONTROL.
PASSIVE-RETURN is similar to OUTPUT in LOGO or RETURN in LISP.

⇑ GENERAL-RETURN a value.
⇑ value process
is the form.
⇑ value caller.
is the same as PASSIVE-RETURN.
⇑ (apply value message) caller.
is the same as ACTIVE-RETURN.

‹Other Output Objects›

will be explained soon.

Defining a Class (Function)

There are many ways to define a class depending on how much the user wants to know about the language and how much control he desires to have over the format of the INSTANCE of a definition. For now we will only be concerned with semantic notions (which also require the least amount of explanation to all concerned).

LOGO/SIMULA/FLEX Fashion

"To" will define classes of roughly the power of SIMULA or FLEX which include such things as function, process, and structure definitions in other languages.

To To .name .body .End.
"As shown, "To" takes the first object in the message stream unEVALuated to be the name of the class. All of the rest of the input stream is a structure which is taken to be the code body of the class. A member of the class CLASS is INSTANTIATED and bound to the name. When control is later passed to the name a new instance of the class will be created and run."
End.

no really meaning ?!

Examples;

```
To factorial :n.
   ↑ if n=0 then 1 else (n*factorial n-1).
End.
```

This looks a lot like LOGO (intentionally) except that the input variable ":n" is not part of the heading (as in LOGO), but is part of the "body". This reflects the fact that input objects act like functions and thus can be used anywhere in a program. When a "function" is instantiated, the first thing that is done in most languages is to bind the arguments to a new set of names. The very same effect is achieved in SMALLTALK when the "evaluating input object", ":" , is used in the first set of expressions.

A no need to distinguish between passing formal names & passing variable names or values

Conventional Class Definition

"To" as shown above, was included mainly for people familiar with LOGO and LISP. SMALLTALK really treats "class objects" like any other object. That is, any object is a member of a class---so an object which creates a class is a member of class CLASS.
This means that a more general (and more conventional) way to define factorial would be to say

```
↓factorial ← class↓(If :n = 0 then 1 else (n * factorial n
- 1).
```

or perhaps

```
↓factorial ← class↓ :n.
                    If n = 0 then 1 else ¦n *
                                         ¦factorial n - 1
```

using the ‹tab list› convention. One could even say

```
↓var ← ↓n.
↓factorial ← class
          ↓(:)¦ var ¦ ↓(= 0 then 1 else)
          ¦(var¦ ↓(* factorial n - 1)).
```

where "¦" means "append" pretty much in the LISP sense.

Total Control of the Instance

***for bit pickers, more on this later this summer.

Control (and State changing, etc.)

```
To If :exp.
    !exp.
    End.
```

""If" is really just a dummy which computes a value to be APPLYed to "then" or "_". This means that "TRUE"ness and "FALSE"ness are properties of objects. This allows us to consider all legal numbers as TRUE, if we wish. A class with one instance EMPTY is provided to handle "FALSE" cases.

To ↓

```
        ,name  |„← ,  (:exp. ↑ exp)
                   "lookup the name in current environment (if not
                   there, enter it as most global) and replace BINDING
                   with value of "exp" ".

               ! name.
           "note that the value of the expression on the right  "exp" is
           RETURNed when a rebind is attempted, but when used as QUOTE,
           it is the name which is RETURNed."
           End.

       To Eval  :exp :globalenv :return :msg.
           "There are many ways to EVALuate expressions in Smalltalk.
           This one allows the user to set up an arbitrary environment
           for free variable fetches, an arbitrary RETURN process, and
           an arbitrary MESSAGE environment."Eval" is included here
           since it is very frequently used in definitions of new
           control primitives".
           End.

       To Repeat ,Loopexp.
           Code repeat.
           Eval Loopexp |global |self EMPTY.
           Code again.
           End.
               "Repeat EVALs its loop expression in the context of its
               caller."

       To Again
           "RETURNs control to the caller of its caller--i.e. to a
           looping control primitive of some kind such as "Repeat"
           which can decide what to do next".
           End.

       To Done
           "RETURNs control to the caller of (the caller of its
           caller)--to one level further out than a looping control
           primitive. This automatically terminates the loop.
           Eventually "Done" will have an optional argument for
           passing the RESULT of the loop back".
           End.

       To Create
           "Reschedule caller to be run instead of waiting for a
           subroutine  RETURN".
           :call.
           "This causes an evaluation of the argument. So it will also
           be running".
           End.
                "As seen, "Create" causes a parallel fork in control.
                Actually, this is what happens naturally in
                SMALLTALK---the default message discipline is
                deliberately limited to a subroutine "wait for reply"
                protocol. "Create" simply prevents the caller from being
                passivated".

   To Word

       „Explain„
           ↑"Words are like LISP atoms or ALGOL identifiers. Their basic
           operations have to do with assembly and disassembly of their
           internal structures.
               Words also have a special meaning in the context of
               evaluation. An unquoted instance of a word will be looked
```

up (look itself up) when encountered by the EVALuator. So cat.first means "look up the most local binding of the variable "cat" and APPLY it to .first". But .cat.first means " call routine "." which RETURNs the word "cat", which is APPLYed to .first, which, as seen below, will RETURN "c" ".

Numbers are words also, but have many additional operations having to do with arithmetic and so are defined as a separate class.".

„← ⇒ :value.word ⇒
↑self.
"..."

„first ⇒
↑"the first character of the printname of the word".

„f ⇒
↑"same as "first"".

„last ⇒
↑"...the last character of the printname of the word"

„l ⇒
↑"...the same result as for "last". This is just an abbreviation."

„butfirst ⇒
↑"Somehow return all but the first character of the string representation of the word."

„bf ⇒
↑"...same as butfirst."

„butlast ⇒
↑"Somehow return all but the last character of the string representation of the word."

„bl ⇒
↑"...same as butlast."

„join ⇒ :value1.word? ⇒
↑"This is roughly equivalent to the "cons" of LISP. The word will be connected to the list in "value1", and a new list reference will be returned."

„wjoin ⇒:value1.word? ⇒
↑"This is roughly equivalent to concatenate in SNOBOL. The printname of the two words are joined together to produce a new word which is returned. .cat wjoin .dog produces .catdog."

„word? ⇒
↑value.

„empty? ⇒
↑EMPTY.

„length ⇒
↑"Somehow calculate the length (in characters) of the number (including "-" and ".") ."

„print ⇒
↑"Return a string representation of the object which may be displayed. Each class which has instances which have a meaningful visual representation will have a meaning for .print. This is much simpler than having to inform a global print routine about the format of each new class."

```
„then ⇒ :value1 ⇐ „else ⇒ ◊dum ⇒ ↑ value1
                              or          ⇒ ↑ value1.
```
"Having "then" in "Word" in this way means that we are adopting a convention that legal words in the context of a test act as TRUE and thus cause the "then" expression to be evaluated."

To Number

„Explain⇒
↑"Numbers work in a very intuitive way. The READ program recognizes number literals and creates instances for them in storage.The bits that represent the particular instance of a number are stored in the variable "value" and can be changed by assignment as shown. This might be illegal if it is decided that numbers are unique atoms. The opposite is assumed here."

„← ⇒ :value.number?⇒
↑ self.
If a "↑" is recognized in the input stream, what follows is evaluated and bound to "value" which is applied to number? which returns TRUE if it is. The actual value of the number object itself has been changed so that other objects which have pointers to "self" will feel the change. This might be made illegal.

„first ⇒
↑"Somehow return the first character of the number which is "-" if negative, is "." if between 0 and 1, and a digit from 0 to 9 otherwise. It may be reasonable to calculate this value rather than keep a string representation of the number around."

„f ⇒
↑"...the same result as for "first". This is just an abbreviation."

„last ⇒
↑"Somehow return the last character of the number which is "." if greater than 1 and known inexactly, and a digit from 0 to 9 otherwise. It may be reasonable to calculate this value rather than keep a string representation of the number around."

„l ⇒
↑"...the same result as for "last". This is just an abbreviation."

„butfirst ⇒
↑"Somehow return all but the first character of the string representation of the number."

„bf ⇒
↑"...same as butfirst."

„butlast ⇒
↑"Somehow return all but the last character of the string representation of the number."

„bl ⇒
↑"...same as butlast."

```
▫join ⇒ :value1▪word? ⇒
   ↑"This is roughly equivalent to the "cons" of LISP. The word
   will be connected to the list in "value1", and a new list
   reference will  be returned."

▫wjoin ⇒:value1▪word? ⇒
   ↑"This is roughly equivalent to concatenate in SNOBOL. The
   printname of the two words are joined together to produce a
   new word which is returned. ▪cat wjoin ▪dog      produces
   ▪catdog."

▫number? ⇒
   ↑value.
   "Anything not EMPTY will act as TRUE."

▫word? ⇒
   ↑value.

▫empty? ⇒
   ↑EMPTY.

▫length ⇒
   ↑"Somehow calculate the length (in characters) of the
   number     (including "-" and ".")  ."

▫print ⇒
   ↑"Return a string representation of the object which may be
   displayed. Each class which has instances which have a
   meaningful visual representation will have a meaning for
   ▪print. This is much simpler than having to inform a global
   print routine about the format of each new class."

▫then ⇒ :value1 ⇒ ▫else ⇒ ▫dum ⇒ ↑ value1
                              or          ⇒ ↑ value1.
   "Having "then" in "Number" in this way means that we are
   adopting a convention that legal numbers in the context of a
   test, act as TRUE and thus cause the "then" expression to be
   evaluated."

▫= ⇒ :value1▪number? ⇒
   ↑"value if value and value1 are numerically EQUAL, otherwise
   EMPTY.  Note that this allows "a=b=c" to work correctly."

▫≠ ⇒ :value1▪number? ⇒
   ↑"EMPTY if value and value1 are not numerically EQUAL,
   otherwise  value. Note that this allows "a≠b≠c" to work
   correctly."

▫< ⇒ :value1▪number? ⇒
   ↑"value if value is numerically less than value1, otherwise
   EMPTY.  Note that this allows "a<b<c" to work correctly."

▫> ⇒ :value1▪number? ⇒
   ↑"value if value is numerically greater than value1, otherwise
    EMPTY.. Note that this allows "a>b>c" to work correctly."

▫+ ⇒:value1▪number? ⇒
   ↑"value added to value1."

▫- ⇒:value1▪number? ⇒
   ↑"value1 subtracted from value."

▫* ⇒ :number? ⇒
   ↑"value multiplied by value1."
```

Right to left word
All to mating mog°
all the way

2<4<1
2<4<3

4>2>3

```
    ,/ ⇒ :value1.number? ⇒
       ↑"value divided by value1."

    ,mod ⇒ :value1.number? ⇒
       ↑"value modulo value1."

    ,ip ⇒
       ↑"...the integer part of value."

    ,fp ⇒
       ↑"...the fractional part of value."

    ,exp ⇒
       ↑"...the exponent (to the base 10) of value."

    ,mag ⇒
       ↑ if  value < 0 then (0 - value) else  value.

    <other numeric functions which are stored as attributes>
    sin, cos, other trig functions etc.

To List
    ,Explain
    ,first ⇒ ,← ⇒ :value.list ⇒
                  value.word? ⇒
    ,f ⇒ ,← ⇒ :value.list ⇒
                  value.word? ⇒
    ,last ⇒ ,← ⇒ :value.list ⇒
                  value.word? ⇒
    ,l ⇒ ,← ⇒ :value.list ⇒
                  value.word? ⇒
    ,butfirst ⇒ ,← ⇒ :value.list ⇒
                  value.word? ⇒
    ,bf ⇒ ,← ⇒ :value.list ⇒
                  value.word? ⇒
    ,butlast ⇒ ,← ⇒ :value.list ⇒
                  value.word? ⇒
    ,bl ⇒ ,← ⇒ :value.list ⇒
                  value.word? ⇒
    ,join ⇒
    ,! ⇒
    ,¦ ⇒
    ,sentence? ⇒
    ,list? ⇒
    ,empty? ⇒
    ,length ⇒
    ,print
    ,= ⇒ :value.list ⇒
    ,≠ ⇒ :value.list ⇒
    ,< ⇒ :value.list ⇒
    ,> ⇒:value.list ⇒
    ,makeword ⇒

To String
Position
    Here are a set of useful operations for manipulating
    two-dimensional space. The convention is adopted that "posx" and
    "posy" will refer to position state, and "heading" will refer to
    direction state. The programs are written so that the most local
    occurance of these variables in the dynamic environment will be
    updated. See the program "Spacevehicle" for a simple example.

    To Forward :distance.
       posx ← posx + distance * heading.cos.
       posy ← posy + distance * heading.sin.
```

```
End.

To Right :angle.
   heading ← (heading - angle) mod 360.
End.

To Left :angle.
   heading ← (heading +  angle) mod 360.
End.
```

Output (to displays, music, turtles, etc.)

```
To Show :picture.
   "This comprehensive routine allows the picture to be EVALed
   and then copies the picture information into the display area
   using either the dynamically available variables "posx",
   posy", "heading", if its own bindings for these parameters are
   EMPTY.
```

Some SMALLTALK Programs
‹See SMALLTALK2.DC for Program Examples›