Inter-Office Memorandum

To	TinyTalk Disinterest	Date	January 4, 1980
From	Larry Tesler and Kim McCall	Location	Palo Alto
Subject	TinyTalk Stack Changes	Organization	PARC/SSL

XEROX

Filed on: <Tesler>TinyStack.memo

These plans are not guaranteed to be well thought out.

(1) The PC stored in each stack frame is currently that of the caller. The debugger will be simpler if the PC is that of thisContext instead. Dan is planning to do this in full Smalltalk also. So we'll change it.

(2) The callerBP (and callerSP) is currently encoded by taking the real machine address and setting the low order bit. It would be more meaningful to have it be a word offset from the base of the system stack. That way, contexts could do arithmetic on it to access fields of the stack frame. So we'll change it.

(3) Class *TContext* should have the messages: *pc, method, receiver, caller, argn, argn←x, temp$n,* and *temp$n←x.* Eventually, they can do range checking. They should all call *self component: wordOffset* which in turn calls *Sys location: wordOffset + bpOffset.*

(4) To evaluate an expression in the context of a stack frame that is being debugged (the *seen context*), we have two problems to fix.

> (a) When the expression ends normally, it should just return to the eval'er. But if the message is something like *proceed* or *quit,* then the stack must be popped down to the debugged frame (or some ancestor of it). This will be done by calling a primitive *thisDialog returnTo: seenContext* (or whatever context it is returning to).
>
> (b) The compiler must compile any references to temporaries and arguments of the seen context as follows: *t1* is compiled as if it were *t1 value* and *t1←v* as if it were *t1 value←v.* The evaluator must then push, for each argument and temporary of the seen context, a corresponding temporary for the evaluation, whose value is a *StackReference* that has one field, the word offset of that variable from the base of the stack. To do this, the Smalltalk-coded *eval* allocates all the necessary StackReferences into a Vector, and passes that Vector to the primitive *eval,* which copies the Vector onto the stack.

(5) As an experiment in safety, we will add a word to each stack frame that is either *nil* or the oop of the sole instance of class TContext that references that stack frame. During *return,* if the word is non-*nil,* then the TContext it references must be invalidated. For now, we'll simply *nil* out both fields of it (the alternative is metamorphosis).

Doing this makes it inadvisable to cause a deep return by changing *thisContext caller.* But *thisDialog returnTo: context* can just check for TContexts to invalidate along the way.

-- Messages from file: [PARC-MAXC]<ADELE>MESSAGE.TXT;1
-- SUNDAY, JANUARY 6, 1980 13:36:25-PST --

Date: 4 Jan 1980 11:13 am (Friday)
From: Tesler
Subject: Bayles Holt
To: Ingalls
cc: Adele, Tesler

Here is our whole correspondence.

---

2. Date: 5 Dec 1979 10:18 am (Wednesday)
From: McCall
Subject: TinyTalk
To: Tesler
cc: McCall

I received this. It looks as though this guy has the wrong idea of what the interpreter is like; but we might be able to send him copies of the emerging documentation -- just your book chapters or something -- letting him pick out the relevant parts if he's interested enough.

I'll let him know that there is no Tiny-specific documentation and that I am asking you how we might accommodate to his request.

Kim

- -

1. Date: 3 Dec 1979 7:47 am (Monday)
From: Holt.WBST
Subject: TinyTalk
To: McCall.PA
cc: Holt

I have been doing some study in Forth and forth-like languages and have pursued with interest some of the development of Tiny accomplished by you and Larry Tesler. Is there now more information available or some distribution list which can help keep me up to date on current developments? (That is if you do not object to my questioning.)

Thanks, Bayles.

---

3 Date: 5 Dec 1979 10:26 am (Wednesday)
From: McCall.PA
Subject: Re: TinyTalk
In-reply-to: Your message of 3 Dec 1979 7:47 am (Monday)
To: Holt.WBST
cc: McCall, Tesler

I'm glad you're interested. The TinyTalk (as we are now calling it) interpreter is not as Forth-like as some of the original discussion of the system may have made it appear, but you might be interested anyway. We have been concerned mainly with implementation to this point and haven't done terribly much in the way of documentation. LRG is however writing a book on SmallTalk and a discussion of TinyTalk will appear in that book as it is now conceived. Some considerable documentation of the storage management system has been written in that connection. Larry has been writing all the documentation, and I have asked him what we can do for you. When he answers, I'll let you know; or if you prefer, you could msg him directly.

Thanks for your interest,
Kim

---

4 Date: 6 Dec 1979 1:39 pm (Thursday)
From: Tesler.Maxc
Subject: Re: TinyTalk
In-reply-to: Your message of 5 Dec 1979 10:18 am (Wednesday)
To: Holt.wbst
cc: McCall, Tesler

Hi, Bayles. I'll start announcing TinyTalk milestones on SmalltalkInterest (and suggest Dan do the same for new implementations of full Smalltalk). As Kim said, the TinyTalk documentation is embodied in chapters and appendices of the Smalltalk book. I'll talk to Adele about sending you copies to read. Are you interested in function, implementation, or both?

One point of interest is that we think we can support regular Smalltalk syntax for hardly any more code than Forth syntax. On the negative side, the system is now 3 times larger than our target size.

Larry

5

Date: 13 Dec 1979 11:41 am (Thursday)
From: Holt.WBST
Subject: TinyTalk
To: Tesler.Maxc
cc: Holt.WBST, McCall.Maxc

Thanks for message. I have not replied because of several delays mostly due to the file server being down. My interest in TinyTalk stems from a desire to make it useable in a 68000 system. I am therefore interested in both function and implementation but especially implementation. It would be extremely nice to discover ease in transporting, but if not, I'm still interested in its function. The stack architecture is what impresses me.

Thanks

6

Date: 13 Dec 1979 12:18 pm (Thursday)
From: Tesler.Maxc
Subject: Re: TinyTalk
In-reply-to: Your message of 13 Dec 1979 11:41 am (Thursday)
To: Holt.WBST
cc: Tesler, McCall, Adele, Ingalls, Alan

I am glad you are interested in getting TinyTalk up on a 68000 system. How much memory will it have? If 64K bytes, Tinytalk will be good for many applications. If less, it will be good for very few. If 256KB or more, you should consider full Smalltalk-78, which is also based on a stack architecture. Both are easy to transport, although TinyTalk is somewhat easier because it is 2KB instead of 5KB machine code (sizes guesstimated).

If you implement either one on a 68000, I hope it is entirely compatible with our 8086 implementation and that we can arrange to acquire it and possibly publish it.

If you really prefer Forth-like syntax, you can have that, but we offer more conventional syntax in full Smalltalk-78 and hope to soon in Tinytalk.

Larry

7

Date: 2 Jan 1980 7:08 am (Wednesday)
From: Holt.WBST
Subject: Re: TinyTalk
In-reply-to: Your message of 13 Dec 1979 12:18 pm (Thursday)
To: Tesler.Maxc
cc: Holt, Genovese, McCall.Maxc, Adele.Maxc, Ingalls.Maxc, Alan.Maxc

We are interested in making a flexible development system for the 68000 without being tied to Motorola's sometimes inadequate support. We had originally planned about 32K bytes for starters but could probably make 64K without to much trouble. Later on we might even expand that.

Right now the prospect of having access to Smalltalk would be very attractive but let me emphasize that it is the stack architecture that we're after, the syntax is a bonus.

I'm now your student and am entirely open to suggestions as to where to go from here.

8

Date: 2 Jan 1980 1:21 pm (Wednesday)
From: Tesler.Maxc
Subject: Re: TinyTalk
In-reply-to: Your message of 2 Jan 1980 7:08 am (Wednesday)
To: Holt.WBST
cc: Tesler, Genovese.WBST, McCall, Adele, Ingalls, Alan

Bayles,

I am not sure what you mean by "stack architecture"; doesn't all microcomputer software these days make use of the hardware stack?

And what do you mean by "development system"? Do you want to prototype in TinyTalk and then recode in 68000 assembly language? Or make products that run TinyTalk? Or something else?

TinyTalk is meant for relatively small Smalltalk programs, with simple character stream i/o. Is that all you need? There aren't windows, editors, and all that other Smalltalk stuff.

I don't know when we will start to support TinyTalk for users, nor to what extent we will support it. The only sure thing is that it will be well documented in our book. I can't imagine that LRG could give better support than Motorola. I

had in mind that you would read the documentation and support your own copy of the system yourself.

Larry

Today, I sent him a copy of the TinyTalk progress report that SmalltalkInterest got last week. I had told him Dec 6 to get on the list, but he didn't do it.

P.S. You see that my Dec 13 message (cc: you) mentioned his familiarity with Forth.

To: Book Writers, Inc
From: The Whip
Re: A New Outline

Well gang, you cranked it out enough for me to draw up a slightly different blueprint. The changes really only effect the folks writing on Parts 1 and 2. Mostly what I did was shuffle the sections a bit to have more balance between theory and implementation discussions. Don't stop writing--keep the text coming and I will provide the editorial feedback as usual. Remember our goal is to have a full draft by December.

Adele Goldberg
file: < my disk > book.outline2
Version: 8-14-78

theory (prog language)-Prelude,I,II
implementation (kernel)-III-XI
theory (user interface)-XII
implementation (graphics)-XIII-XVI
theory (Language)-XVII
implementation (templates,thinglab, findit, music)-XVIII-XXI
theory (futures)-XXII

Smalltalk: Dreams and Schemes

Prelude: Opening Scenario

A scenario showing how the current system is used for accessing and manipulating information, towards the goal of planning a software implementation (of ?). The plan here is to set the stage for interactive computing the Smalltalk way, emphasizing what Smalltalk is, what it looks like, and why it is an interesting system (done through capsule scenes, not discussion). The selected scenario should show: windows containing text and pictures (that is, user interaction), static organization of class definitions, dynamic operation in terms of sending messages by typing or menu selection, and the existence of active/ interacting objects. The scenario actor browses for and runs some already defined methods, makes some modifications to an existing definition and reruns with a "spy" providing some statistical feedback. The idea here is to leave the reader with enough gaps so he can fill them for himself with his own desires.

PART 1 Towards the Design of a Personal Computing System

This part is philosophical in nature. It includes the notion of a computing system made up of (active) information: that information is simulation; that a computer is a tool that should be malleable like clay; and that, to borrow an analogy from ACK, the goal is to help kids and managers alike fly 747s rather than paper airplanes. By the end of this part, we should be able to trust that the reader knows what a Dynabook is, why LRG is designing it, and what are the programming language problems.

I. Capturing Information for Personal Use

The goal is to provide a new medium with which anyone could describe his ideas about the world in a way that produces a running simulation. A simulation is a dynamic system, a collection of parts which are either acting upon other parts, being acted upon, or mutually interacting. We want to be able to observe, alter the state, or alter the cause of state changes occurring in such systems. There are problems involved in knowing (or specifying) how the parts are constrained to communicate or interact, in being able to write such specifications in a machine readable form, and in being able to make use of the various tools available for realizing these goals.

This medium is what we mean by a personal computer. Its intended users are system builders, application programmers, and users of applications such as painting, animation, music, game design, and text editing systems. Its proposed implementation environment is a personally controllable, ownable computing system. Being able to own the system is a function of its hardware costs; being able to control the system depends on the design of the programming language and the user interface for editing, browsing, debugging, and getting

assistance. Although each personal computer is self-contained in its support of these tools, personal computing should encourage communication with the owners of other such systems. We are interested in computer network problems both to enhance people-to-people interactions as well as to increase an individual's access to information.

II. Basic Concepts of a Programming Language for Personal Computing

Given the goals for personal computing set forth in Chapter I, what concepts must be considered in the design of the programming language? Here is a good place to introduce ACK's biological model as a metaphor for the wholes and parts problem, and to emphasize communication and classification ideas. This is intended as a philosophical discussion which motivates the decision for an object-oriented environment: that is, the desire to create wholes from parts and to communicate with them as well as with their parts.

III. Smalltalk: Object-Oriented Programming

Definition of Object-Oriented Programming

Basic data structure: object
Basic processing: sending messages
Naming: accessing objects, including objects that are collections of objects, through names

References and comparisons to other object-oriented programming languages are provided here (Simula, CLU, Hewitt's Actors, ThingLab and Findit)

Reasons we chose the class/subclass/instance model for implementation

Example Structure

(If possible) Examples in this section will be done using message protocol only; implementation of algorithms will be shown after introducing the syntax in the next part of the book.

Revisit the example in the prelude, focusing on the organization of the objects involved.

Program modularity: adding messages, subclassing

IV. Technical Characteristics of Smalltalk-76

Introduction

Here we divide the system into two parts: the kernel system and the basic system. The idea is to define a system in which the user would do no machine or microcode level programming, but might have to write a lot of class definitions, such as Set, that a lot of users would agree are generally useful. These "useful" definitions will be introduced in Part 3.

By "kernel", we include the data structure of an object and its activation, and message sending. There is "kernel" syntax since we say that the parser is kernel: it is the way to generate executable code. And therefore it includes the notion of a literal, which includes the notion of a code literal--the essential ingredient for sequence control.

In the "basic" system, we extend the kernel system to include additions to kernel definitions as well as new (abstract) class definitions, especially those abstract classes that support user connection to the hardware (because the hardware is not sufficiently powerful or because it is necessary to link to the hardware through machine code for such functions as text display or files). We also add constraints, say, to message syntax, class definition format, and program description syntax.

The main distinction being made here is that, regardless of hardware or the latest implementation experiment, the kernel system exists as described. As such, it is the meaning of the generic term: Smalltalk. Depending on hardware and current implementation design, there is a basic system without which no user

interactions can take place. For example, in the course of our design cycles, we have changed the message syntax considerably in order to be "readable"; we have altered the class definition format in the way we account for temporary (activation) variables; we might further alter the source program description to become a non-linear declaration of constraints (see Chapter XIX). This implies that the process scheduler is part of the basic system.

The Implementation Machine

its function, not its organization; that is, its function is to execute code and to interface to hardware

Program execution

e.g., the bytecode interpreter

The hardware interface

The Kernel System

Objects

(as already described) parts, including Class

Dictionaries

Classes

property dictionary and message dictionary

class hierarchy

here we provide an illustrative example of the subclassing idea. Two examples: (1) HashSet's subclass is Dictionary, and (2) arithmetic, such as

Number

ComplexNumber

RealNumber Complex ShortComplex

WholeNumber Real ShortReal

Integer ShortInteger

(Short* implemented for fixed precision to get efficiency)

Objects revisited (as instance of a class)

creating an instance
managing its storage (in the abstract)
managing its behavior (message dictionary)

Programs (as descriptions of messages to objects)

object code message syntax

(sending a message includes the idea of a class being in the form of a dictionary. Graphically display

context in terms of nesting of dictionaries for (a) nesting message sending and (b) Smalltalk-76 nesting of class dictionaries. Comment here on idea of syntactic extensibility that existed in earlier versions of Smalltalk; discussion of this topic is deferred to a section at the end of this chapter.)

Execution of the object program

activation records (message semantics, interpreter state)

Remote code for sequence control

Execution revisited

the canonical cycle of execution (allocating activation records)

the specially performed bytecodes (arithmetic and indexing)

the primitive methods

Objects re-revisited (owning up to the idea of variable length object which is ultimately an implementation consideration)

the class Array and its subclasses SubArray, String and Vector

The Basic System

Interpreters and Compilers

source code message syntax

Hardware interaction

clock, keyboard, CRT (class Rectangle, TextFrame or Projector), pointing device, disk (class File), ethernet

Error Handler

process interruption, display and resumption

Process Scheduler

process cooperation

V. [needs a title!]

Uniformity of Expression

Notice that Smalltalk is not really that different; a small piece of a Smalltalk program will look a lot like a small piece of a program in other programming languages; moreover, you can write those same programs. In fact, from a dynamic (execution) point of view, the difference in Smalltalk is the mechanism for "calling a procedure"; the decision about which procedure is being called is made at runtime by a dictionary lookup rather than at compile time by a symbol table lookup. But the most significant difference in the design of the system as a whole is the *uniformity of expression* that is seen throughout all layers of the system, from the user interface to the system implementation.

Refer back to the introductory remarks on the definition of a good programming language and examine the effect of the language design on ease of expression, on choice of algorithm, on style of designing a system.

Example Classifications

Illustrate with several examples how we organize worlds into objects. We provide only the message protocol here and will return to look at the implementations after the next part. [Please make suggestions.]

Set and SetReader (some history talks about Stream)
queue, chained list
scheduling the objects on the chained list

animated objects: a class Movie

IC Layout program [to be used later in the debugging chapter]

PART 2 Software Architecture: Realizing a Smalltalk System

VI. Introduction

Assume Smalltalk as the programming metaphor, then the goal is to get Smalltalk running on some hardware configuration. In this section, we describe the Smalltalk-76 implementation, covering the history of prior implementations in a subsequent chapter.

What is the effect of the language design on implementation efficiency? We care about obtaining a lower level on memory cycles (against some criteria for performance and generality) and peripheral support. For a data structure we have chosen an atomic object, a dictionary of properties and messages with associated responses, and a method for putting objects together. Processing means sending messages to objects; algorithms are encapsulated as the response to a message (captured within the context of the object). The design decision for syntax is directed by the issue of "readability": in Smalltalk-76, "compilable" means readable by a human. We compile to parse, not to do type checking. Type checking, in the form of determining that the object can receive the message, is done at runtime.

VII. Basic Data Structure

The idea here is to talk about objects as they are actually represented in the machine.

Objects

Strings

first example of an object
a repository for bits, a variable length set of constants

Programs

Strings whose bits are instructions for the implementation machine

Vectors

variable length collections of objects

Dictionaries

Two vectors plus some semantics

Direct and indirect message and part name

Classes

Property dictionary and message dictionary
Class hierarchy

Objects Revisited (as instances of a class)

Creating an instance

Managing its storage
Managing its behavior (message dictionary)

VIII. Smalltalk Execution

Message Passing

Selectors and arguments

note that selectors is a language concept from Smalltalk-76. There used to be more levels of detail about open colons, keyholes, and so on having to do with syntactic extensibility; we defer discussion on execution having to do with programmable syntax until Chapter X.

Methods

Method/message dictionaries

Classes

Evaluation context

names and values
meaning of Smalltalk method is relative to a context
context includes receiver, method, temporary and global values for names

Fundamental rule of Smalltalk evaluation

the *current context* determines the *next message description*

the *next message description* consists of a *next receiver*, a *selector* and some *arguments*

the *next receiver's* class and the *selector* determine the *next method*

the *next method*, the *next receiver* and the *arguments* determine a new current context (closing the loop)

Basic Control Structure

Activation records (method interpreters)

sender
method progress (method + program counter)
context (receiver + temporary values)

Activation stack formats

special activation stack ala Smalltalk-72
instances of class ActivationRecord
instances of class Stack or Process

Compilation

fundamental change in the process of translation: internal representation changed sufficiently to warrent its being called compilation. The compiler looks up most names and orders the arguments to messages; still do not know which procedure to call until runtime--in fact, procedure might not be there.

The bytecode language

allows the description of messages to objects consisting of a selector and several arguments

The compiler

translates Smalltalk to bytecode language

special treatment for in-line literals: integer, float, large integer, string, unique string, vector

The bytecode interpreter

implements the actions described by the instructions of the bytecode language

Implementation Optimization

small integers and unique strings. Remote code for iteration, if not moved to a lower level, ends up with an extra activation record

Bootstrapping a new system

Smalltalk-76 carries around in it the ability to reproduce itself.

IX. Error Handling and Process Scheduling

Compile Time Errors

parsing errors shown as in-line comments

Run Time Errors

process interruption

viewing and interacting with the semantics of class activation

process resumption

typical error: message not understood

user generated errors: argument or state checking such as incorrect indexing or popping an empty stack; incorrect use of the three loopholes: 's (inst field) and apply (performDangerously) and mem (maybe asObject as well)

Explicit Polling

messages startup, firsttime, eachtime, lasttime

Event Scheduling

ala Simula

Multi-processes

handling input/output without, and then with, Smalltalk processes

X. [History chapter--needs a title]

Interpretation of Smalltalk

Introduction

This section covers two topics within the context of two implementation experiments:

(1) how far you translate before runtime is a function of what object programs look like

(where you find code for the machine that runs a program,
effect of program transformation on syntax choice, programmable syntax)

(2) whether the interpreter is centralized or distributed

Smalltalk-72 Implementation

external representation is a string. You parse it to lists and lists of atoms ("tokenization") and then, at runtime, an interpreter does the rest. The interpreter is a single low level program operating on two

addess spaces, the activation stack and the set of Smalltalk objects (includes programs and classes, and is called the "heap").

Fastalk Implementation

the translation process is the same as in Smalltalk-72. The interpreter is a Smalltalk program in that it is several messages to several objects. That is, the interpreter is distributed: many low level programs implement certain message responses for several system classes (ActivationRecord and Class primarily) which result in the ability to interpret Smalltalk code by the exchange of primitive messages. Because the activation objects are now "first-class citizens", there is only a single address space.

Syntactic Extensibility

By syntactic extensibility, we mean that the message is parsed at runtime by the object that receives it, and that the grammar is defined by the object itself. This section contains a discussion of the advantages (e.g., flexibility for the writer of the program) and disadvantages (e.g., flexibility and therefore non-readability for the reader of the program) of syntactic extensibility. Experimentation evolved conventions in syntax, e.g., left-arrow to mean "store" as part of a message pattern.

Its the programs people write that are not readable, because of the dynamic process of parsing, that cause a user nightmare; execution proceeds as usual. The Fastalk implementation is a "clean" method for interpreting grammar-free message forms. With programmable syntax, a Smalltalk program is composed of lists containing atoms and other lists. It is not possible to know which parts are objects and which messages, other than that the first element of the list evaluates to an object which then gets to decide what its arguments are, with the assistance of the arguments themselves, and so on.

XI. Storage Management

Basic Idea

We have chosen a hardware system and now must provide space for objects; why a virtual memory? what does it do?

Data Flow Maps

how to read them

Schemes

includes optimization considerations

Smalltalk-72

Reference counting with files for backup

objects accessed through direct pointer to their storage so objects not moveable

Page mode Smalltalk

Virtual memory on a swapping disk: OOZE

A virtual memory design using reference counting and implicit use of a single file

Problems: picking up on cyclic structures

Loss of the swapping disk [NoteTaker]

Reference counting with files for backup, objects are movable because they are accessed through indirect pointer to their storage

What we have discovered

standard interfaces
collapsing levels for speed gains
famous tricks

Statistical Tools

Note: the eight hour garbage collection service might be discussed in this chapter, where?

Large Virtual Memories

beginning of a discussion on Findit (retrieval) ideas versus OOZE

PART 3 Coupling to the User

XII. The Programming Environment

Integration of programming tools; integration of modes of interaction

The integration of software tools is one of those important, difficult, and exasperating areas that is too little discussed. Yet it is the comprehensive power, simplification, and uniformity obtained through integration that is most relevant for nonexpert users of computer systems.

Objects and messages are preserved at the top level. This can not happen by accident--i.e., the top level could become a procedure-oriented interaction, translated into the object/message structure. For example, events, like pressing a mouse button or striking a keyboard key, are messages too.

Towards the goal of controlling the user-level interaction style, we have chosen some common metaphors with supporting class definitions. filtering templates as an approach to user interface design.

Viewing and Editing Methods

windows supported by menus (discusses scheduling of window selection);
documents and projectors, filtering information

scrolling (analogy between scrolling, windows and virtual memories)

includes composition ideas; galleys; documents with modeless text editing; command menus; experiments with hand character recognition for editing windows (reference example: GRAIL).

Communicating and Retrieving

Personal computing does not mean computing in isolation
Use of file servers, printers, inter-machine communcation

XIII. Creating a Graphical Interface: System Considerations

When teletypewriter is no longer the user interface...discovering the display medium. This discussion was motivated by the environment considerations of Chapter XII.

The Problem

People want to view informaton in both textual and pictorial forms. There are heavy conventions about text and pictures. The solutions hinge on the ease of communication. We care about how text is handled becsause we want to be compatible with the printing world and we care about making a professional artist happy. Moreoever, we want to satisfy both proessions at the same time. There exists further tension here due to our desires to satisfy novices as well as experts.

The problem is hard because we want to do the above without giving an inch. Text and graphics must be totally editable and have total compatibility for filing and printing. We need to edit these entities fast yet in

a small amount of space, still satisfying the history of opinions of what functions must be available.

Solutions must answer the questions of what does editability mean and where does the control of textual and pictorial structure belong? Add to this a need for common interaction methods that are part of a general programming environment.

Experience with Input/Output Devices

Teletype Simulation

imitative with character generator

Character-generator Approach

first animation and painting attempts

Raster-scan Display

higher resolution, putting it where you want it

Convert

Windows

given the hardware and software environment, several compromises on the idea of updating the screen were made relative to a user interface based on windows

Turtle

line drawing methods, simulation of the LOGO-turtle concept

Text System versus Graphics System

Management of text versus management of "graphical" data

Characters, graphical forms surrounded by highly formal conventions

Bitblt and Its Impact on Text and Graphics

Unification of primitives for display

Printing Becomes Part of the Medium

difficulties in a multi-resolution environment

Smalltalk-76 system primitives

point, rectangle, bitblt, turtle

Animation

Kaos/Shazam, Simulation Kit example, double buffering in ThingLab

XIV. Graphics: Forms, Paths and Images

some graphics applications and the development of a simple system metaphor

Idioms and Tools

addressing the problems of anarchy. When designing the Taj Mahal, it is nice to have the technology of bricks taken care of. Example: an idiomatic approach to illustration--toolbox.

Composition and Layout

Does the printing industry really do all that stuff? Problems associated with typography. Example: the

layout problem--Cypress.

A Simple Metaphor

Paths, forms and images...unifying the manipulation and display of text and graphics.

XV. User Aids: Viewing, Retrieving and Editing

the use of a two-dimensional medium to view a dynamic, multi-dimensional system; break points, tracing, viewing class organization, help system ideas

how may a programmer actually solve and correct a bug (notify window, inspect window)
infinite loops (keyboard interrupt)
rading code dynamically (keyboard interrupt)
reading code statically (browser)
defining new methods
organizing methods
defining new classes
changing class definitions
organizing classes
approaches to program development and effect of working environment on programming style
help aids

XVI. Network Communication

Smalltalk Network Software

handling a real I/O device
real-time constraints
peculiarities of network communications [time-outs,etc]
experience/problems with the scheduling mechanism
earliest Alto/Nova communications
iterations of the design
performance problems
existing design, printing files, WFS

Distributed Computing

[this really belongs in the next part and coordinates with the Findit presentation--let's just write it as shown here and worry about separation ideas after we have the text to work with]

more general problems of distributed computing
coordination among machines
sending code through the network
current thinking about distributing Smalltalk programs over a set of machines
how might we write programs to be run in this manner

PART 4. New Language Ideas: Research Directions

XVII. Introduction

Discusses "language" in the larger sense. Smalltalk-76 is a systems programming language. We have seen already that text editing, as an example, is a different language using mouse clicks and pointing; of course, it has been designed with a great deal of compatibility with the object/message metaphor. If we want to support

a language for information retrieval or for geometric configurations or for planning, further consideration of the metaphorical bases is needed. This part examines several research efforts whose purpose is to explore new views of the programming language and its environment.

XVIII. Findit: Extending the name space of an object

Introduction

In general, knowing the (global) name of an object gives us a reference to that object. If the user does not know the name of something, but knows one or more properties about it, he should still be able to obtain a reference to it (find it). A generalization of the browsing problem in which the retrieval is based on categories, the solution proposed here is to be able to specify the parts of an object, and then to find all objects which satisfy those parts.

Historical Perspective: a Cross Referencing System

teletype-oriented interaction, no object storage outside the Smalltalk system, and no indexing capabilities

Forms as a Retrieval Language

CRT-oriented interaction in which creating and modifying objects take place through record *forms*. The forms are made up of labelled parts called "fields" which can be specified with different editors, e.g., text, pictures, diagrams. Forms for creating records and specifying retrieval requests are the same; this removes the need to use boolean expressions for specifying retrieval. Fields can have incomplete field matches or date ranges. Then retrievals are represented as stream-like set expressions, rather than as explicit lists of objects. References other similar systems from IBM, Berkeley.

Examples: mailing lists, tickets and calculators

Storage Management and Retrieval Methods

Findit-72 introduced file storage for objects and sets, which are lists of references to the objects. Objects were referenced through a master-object table (this system, therefore, was a precursor to the design of the OOZE virtual memory system). External storage of indices as B-Trees [ref. McCreight] and calendars. Problems: insufficient memory, response time too slow.

With OOZE, the virtual memory system, it was possible to discard a separate file management system because records and sets were automatically stored as objects in the virtual memory. This became a disadvantage due to the limited address space and awkwardness of file backup method. Decision to keep B-Trees as external files was due mainly to the complexity of the B-Tree implementation.

In anticipation of different hardware storage capacities (limits of local storage), we are currently returning to the methods of Findit-72, but incorporating extensions to data bases accessible over a communications network.

Viewing and Editing Methods

Use of a "modeless" text editor. Because the parts are named, the forms can be collapsed into menus of the names that can also be edited. When a form is considered to be a "document", the menus of names provide an alternative approach to the idea of scrolling. With the introduction of "projectors" (Chapter V), forms became "labelled galleys".

Viewing: sets have set readers; documents have windows; objects have notions of whether or not they are viewed so it is possible, when they are not viewed, to compact them.

A Research Center Library Application

XIX. ThingLab: Constraints and Merges in the Language Kernel

Purpose of ThingLab (as a simulation laboratory)

ThingLab provides an environment for building simulations. Within this environment, new objects are constructed by combining and editing existing objects. Constraints are used to describe the relations among the parts of an object, while the operation of merging is used to specify connectivity and to apply pre-defined constraints.

Focus of this Chapter

describe ways in which the Smalltalk notion of an object has been extended in order to be able to have a manipulatable handle on constraints, their specification and satisfaction.

1. why the extension is needed: because Smalltalk method of dynamic object creation does not lend itself to handling constraints.
2. want to handle constraints because it is a new languag eidea in which the programmer focuses on the solution as a set of state changes and stae relations -- a way to package up the necesssary procedural and decclarative knowledge in order to maintain a relationship among objects.
3. interactive graphic definitional approach to programming is a form of programming by demonstration with constraint satisfaction.

Solution Idea

1. objects are in terms of parts and wholes
2. constraints are objects that stand for relations. A case of this is when the relation is =, which we can implement as a sharing. Pragmatically, sharing is handled as a subclass of a constraint called a merge. i.e., merges are objects that stand for sharing of parts.
3. messages are objects that stand for the act of sending a request
4. prototypes are objects that casn be interactively constructed and used as master versions for generating like objects.

Example

problems stated in terms of constraints and merges: bar chart example, inadvertant sharing of parts, no constraints implies explicit programs are needed

Historical Perspective

the evolution of the ThingLab constraint representation and the procedural-declarative controversy; references Sketchpad, Abset, Simula, KRL

Simulating TinkerToy models

Partial re-implementation of Sketchpad

Pointers everywhere scheme

Fastalk implementation

Smalltalk-76 implementation

Prototypes, Parts and Whole

Subparts: paths

Prototypes and instances of prototypes

comparison with class-instance structure

Relation to inheritance and subclassing

Interactive construction of prototypes

Describing Relations Among Parts

Merges

done with paths; who owns the merge and why

Constraints

constraints as bundles of continuously sent messages; applying constraints

Why is this way useful?

it allows the user to add new information in a linear manner, even though the interactions of the new information with existing information may be non-linear.

Constraint Satisfaction

why this is hard--some perverse examples
compile time versus run time
methods used: working forwards, working backwards, relaxation, inclusion of runtime checks
description of constraint satisfaction for bar chart example

ThingLab Editor

interactive construction of objects, again emphasizing the need for prototypes

Putting ThingLab Ideas into the Smalltalk Language Kernel

Philosophy

note on the general philosophy of attacking hard problems in language design by first building a subsystem

Interactions between current Smalltalk code and constraints

Mutiple superclasses

whenFirst statement

Time-dependent ideas; event queues

Parallelism

Storage management using merges

Generative procedural information from declaractive information for constraints

Meta-constraints

Multiple views

XX. Programming in Two Dimensions

Templates as a Surface Syntax

Graphic templates for object definition and sequence control: a new surface language for Smalltalk.

Dictionary template

Literal template

Message sending template

Control template

Object description template

Menus and Forms Editing

Automatic menuing for message sending; layout editing; program illustration.

XXI. Music

Why Include a Chapter on Computer Music?

Music for its own sake

For musical interaction, the computer is not just another process control device, like a robot. Conventional musical systems, with some qualifications, lack several characteristics that could enhance ones ability to produce structured acoustic phenomena:

(1) it is not easy for a single individual to play more than one instrument at a time; need to find others who play the right instruments;

(2) in order to explore other instruments, have to take the time to gain a reasonable level of mastery of them; and

(3) it is totally impossible, with a given ensemble, to explore variations in the actual execution of some acoustic performance.

Music as an exemplar of all the various media for which the computer can serve as "meta-medium"

The "minimal" personal computer offers certain means of interaction which happen to be most cost-efficient for the human user. Thus the ubiquitous typewriter keyboard and display. The ultimate personal computer strives to make accessible all I/O modalities. It will be more richly connected with the user's physical and sensory environment--not just his intellectual environment. The destiny of the personal computer can lie not so much in its service through one particular medium, but as a *meta-medium* through which the user can interact with the various media in his environment. This potential is particularly strikingly illustrated by the Smalltalk music system.

Fundamental Design Criteria

Software-based system.

No hardware other than keyboard and D/A

Real-Time

12 simultaneous voices, frequency response greater than 5 khz. Note: this criterion was so firmly established so early in the development of our computer hardware systems that it became one of the design criteria for the hardware itself.

Organization and Interaction Principles: Graphical Editing Methods

Introduction: Music is structured sound

Sound is a rich and potentially very complex medium. In order to give user full control over it, must invest considerably in developing new idioms for manipulating sound. Typically, these idioms will *not* be

acoustic or musical, since they would be too cumbersome and inefficient to use. Use of graphical devices is more efficient.

The higher level structure (notes) is a set of discrete pitches and durations--the musical score. The lower level structure, invoked for each pitch-duration pair in the score, is a specification for playing a single note of music--the timbre or instrument. This represents a partitioning of information according to the idiosyncracies of conventional occidental music. But this is just a special case of the more general problem: Arbitrarily nested levels of specification of the basic parameters--pitch, volume and harmonic spectrum. A facility for producing music must provide means for interacting with the various levels of musical structure.

Drawing wave forms

Timbre editing

novel, compact, four-dimensional graph

Note editing

straightforward "piano roll" model

Implementation Considerations

Bandwidth

Sampling synthesis

Saunders' poor man's FM method

Dreams

semi-special purpose hardware
conventional score notation
unification of representation for music and graphics

XXII. The Book End

Here we complete the scenario started as the prelude to the book, but now the idea of a system that aids a designer in his quest for accessing and making good use of information is not understood in the sense of already having had one round of implementation. The intention of this final scenario is to indicate our next steps so that the next book can be written and read on a Dynabook.

Part 5 Appendices

Appendix 1 The Basic Smalltalk System

A summary of definitions from Parts 1 and 2

Appendix 2 The "Useful" Smalltalk System

A summary of definitions from Part 3

Appendix 3 Acknowledgements: An LRG History

-- Messages from file: [PARC-MAXC]<TESLER>MESSAGE.TXT;1
-- FRIDAY, AUGUST 25, 1978 09:35:17-PDT --

Date: 25 AUG 1978 0757-PDT
From: ADELE
Subject: book chapters
To: tesler, ingalls, robson

I am off for the weekend and Dan and Larry will be gone
for longer so I put on IVY <ADELE>BOOK>CHAPTER.* the
working versions that I have of chaters 3, 4, 6, 8, and 15.
Dan has a large edit planned for 3, I have a lot of notes
in small print where chater 4 needs exaaples and me prose.
Much of that prose can still come from Larry's original
manual albeit, carefully still the style is different. It
dives too fast into the informaton, a. well it ought to
since it is a reference manual. Chapter 6 is simply my
piecing together the two writings of Dave--I have not
examined them carefully to see if they make since--they certainly
do not in the order currently there. Chapter 8 is the
compiler text Larry did--I ran out of disk space so the
figures are not there (they are on maxc<tesler>compiler-
chapter.press. Also on IVY<ADELE>BOOK>BOOKUSER.cm is
the user file I have been using--it is slightly
different than all of yours because I have been formatting!!
Chapter 3 has figures--the full chapter is
on that same directory as chapter-3.press.

[Looks like I am not typing well--in the above,
chaters is chapters and since is sense]

Current assignments are: have a good vacation but if you
decide to write, there is a large need for examples for chaper 5 (I am trying to figure out a good
format); Dave will get the content of Chatr 6 (Implementation)
straightened out; Larry has a big rewrite of the
user-aids (15) using Kim's work, as well as writing up the
ic example; when Ted comes back, Dan had some suggestions
to have his stuff make better sense. And it is stil not
clear to me what does in Chater 7--currently thought of as
Program Execution or introducing activation records in
more detail.

Adele

To: Book Writers, Inc
From: The Whip
Date: July 25, 1978

Here is a list of class names in the summer release of Smalltalk. (Thanks to John Maxwell whose class Tree generated the file.) Rather than my editing the list (to exclude John's personal definitions, for example), I am distributing it in this way. The idea is for you to highlight, any old way you choose, the classes to which you refer in your writing. That will give me a first pass on appendices. I do want to sort the references from the "Kernel" system from the "Basic" system, but I can do that later. Add any classes you invent for illustrative purposes (in alphabetical order and/or in the class hierarchy).

Array	CoreLocs	
	Interval	
	Paragraph	TextEntity
	String	UniqueString
	Substring	
	Vector	
BitBlt		
BitRectTool		
Class	VariableLengthClass	
ClassOrganizer	SystemOrganizer	
Compressor		
Context		
Cursor		
DefineVariables		
Directory		
Document	Galley	
	Page	
Etherworld		
FieldReference		
FontWindow		
Form		
FormSet		
Generator	JustParser	
	Searcher	
HashSet	Dictionary	SymbolTable
	MessageDict	
Menu		
MessageTally		
Number	Date	
	Float	
	Int32	
	Integer	
	LargeInteger	
ObjectReference		
Pacbuf		
PageBuffer		
ParagraphPrinter	BravoPrinter	
	PressPrinter	
ParagraphScanner		

ParsedAssignment		
ParsedConditional		
ParsedConjunct		
ParsedDisjunct		
ParsedFieldReference		
ParsedLoop		
ParsedMessage	ParsedNegation	
ParsedObjectReference		
ParsedRemote		
Parser		
ParseStack		
Point	UserEvent	
PressFile		
PriorityInterrupt		
PriorityScheduler		
Projector		
RadioButtons		
Reader		
Rectangle	BitRect	
ScrollBar		
Selection	BitRectSelection	
	TextSelection	ParagraphEditor
Socket	EFTPReceiver	
	EFTPSender	
	RoutingUpdater	
Stream	Dispframe	
	File	AltoFile
	ParsedBlock	
	PQueue	SafeQ
	Queue	EventQueue
	Set	Image
		Path
	SetReader	
Style		
Textframe	ListPane	ClassPane
		OrganizationPane
		SelectorPane
		StackPane
		SystemPane
		VariablePane
TextStyle		
Time		
Timer		
TokenCollector	FieldNameCollector	
Turtle	PressTurtle	
UserView		
VirtualMemory		
Vmapper		
WidthTable		
Window	BitRectEditor	

CodePane
DocumentWindow

PanedWindow

GalleyWindow
PageWindow
BrowseWindow
CodeWindow
InspectWindow
NotifyWindow
SyntaxWindow

8-7-78
talk 8-8-78

Optimizations

(1) Boolean

{and | or | not}

Don't evaluate unnecessarily

{if | while} {and | or | not}

Jump after each conjunct
instead of forming
Boolean result
Suppress Jump after Return

(2) Pop/Push

x←y + z. t←x-r.

... store&pop x; push x; ...

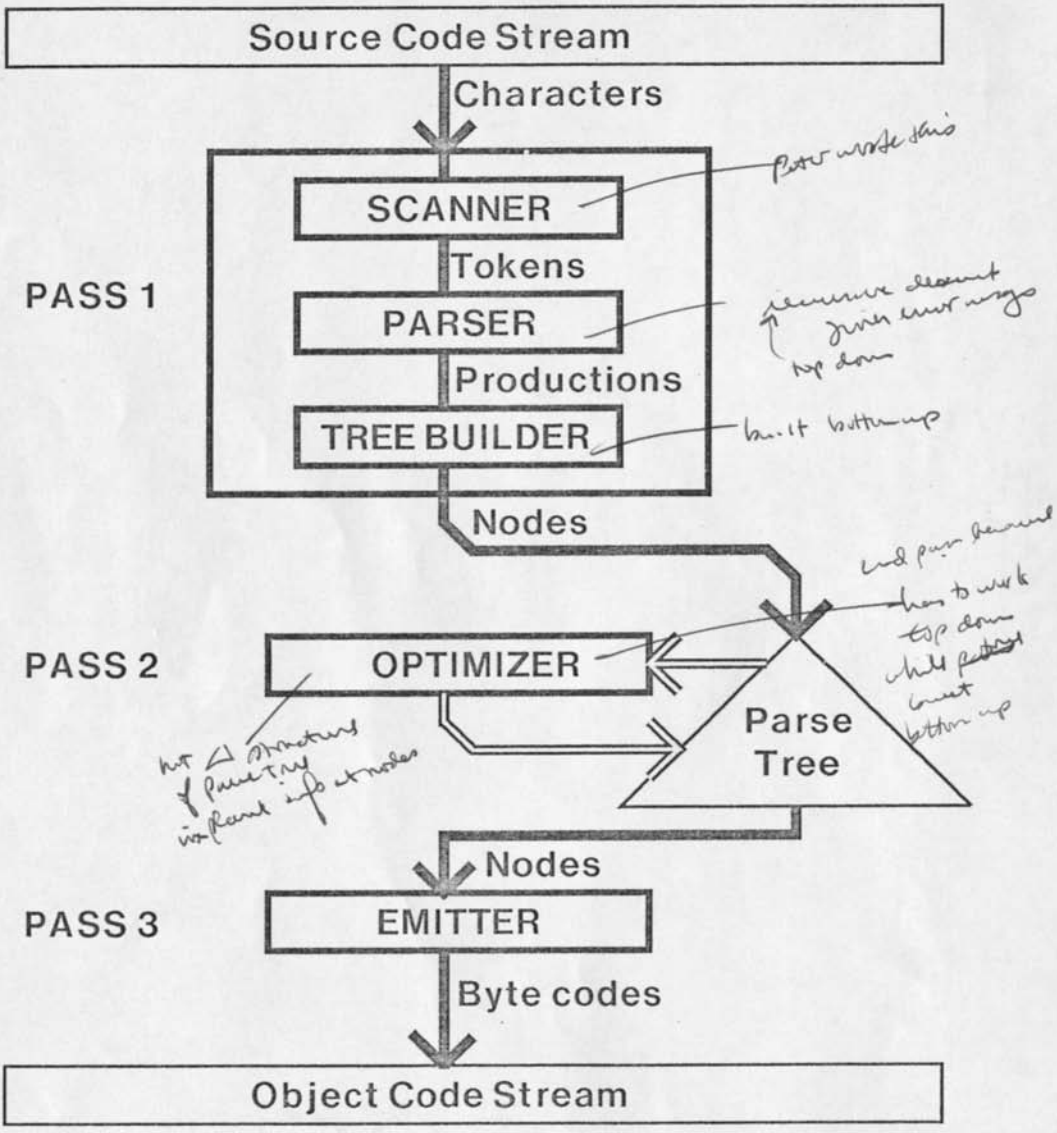
... store x; ...

(3) Short jump

Jmp1...Jmp8, Bfp1...Bfp8

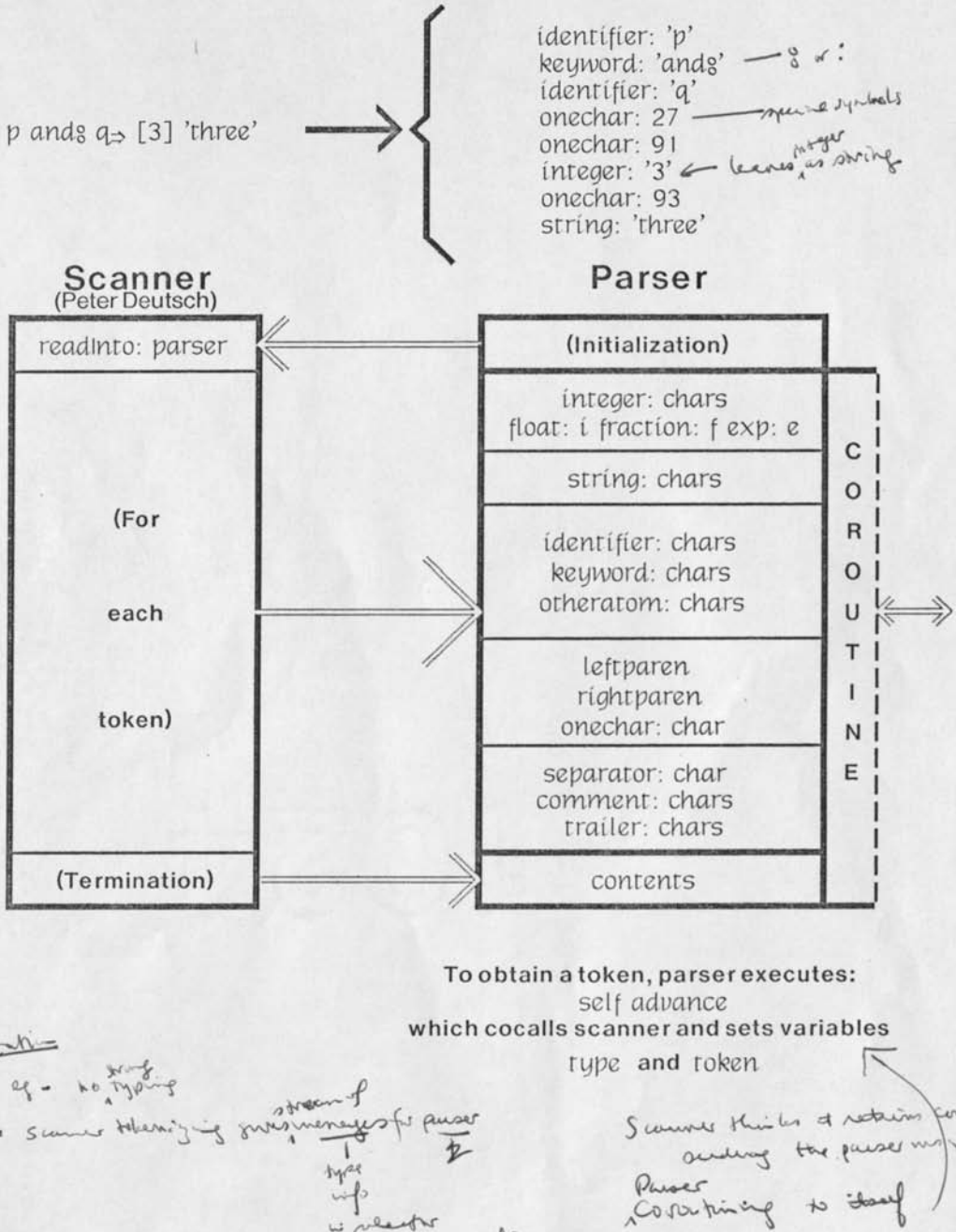
are one byte instead of two

Compiler Organization



Written as / code compressor by Peter Deutsch

Scanner/Parser Interface



To obtain a token, parser executes:

self advance

which cocalls scanner and sets variables

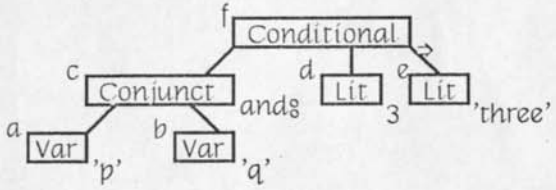
type **and** token

Parser/Builder Interface

p ands q⇒ [3] 'three'

Productions

a ← builder variable: 'p'
b ← builder variable: 'q'
c ← rcvr: a selector: 'ands' args: b
d ← builder literal: 3
e ← builder literal: 'three'
f ← builder ifExpr: c thenExpr: d elseExpr: e



Parser

to: generator
pattern: block temporaries: block
(for each decl)
body: block
(for each production)
mustBeDone

Builder

(initialization)
(compile heading)
declaration: token asArg: bool
(compile body)
literal: obj variable: name rcvr: rcvr selector: sel args: args ifExpr: e1 thenExpr: e2 elseExpr: e3 assignment: var expr: expr . . .
(termination)

? Spike with all kinds of nodes // one w/ recursive evals special problem?

Slide Low No
much info out 5-

Optimizer Pass

of 2g

silver attached, derived work

7 ifExpr(0,F), thenExpr, jmp L, F: elseExpr, L:

Conditional p and: q ⇒ [3] 'three'

sizeForValue

1 elseSize ← elseExpr sizeForValue.
2 thenSize ← thenExpr sizeForValue + elseSize jmpSize.
4 ifSize ← ifExpr sizeForTruth: 0 falsity: thenSize.
7 ↑ifSize + thenSize + elseSize

ifExpr	thenExpr	thenSize	elseExpr	elseSize

2 1

1 push **Literal** 3
sizeForValue 1
location

1 push **Literal** 'three'
sizeForValue 1
location

4 left(0,F), right(T,F)

Conjunct p and: q
0 2
sizeForTruth: trueSkip falsity: falseSkip

2 rightSize ← right sizeForTruth: trueSkip falsity: falseSkip
4 ↑(left sizeForTruth: 0 falsity: rightSize+falseSkip) + rightSize

left	right	rightSize

2

2 push p, bfp F
Variable p
0 4
sizeForTruth: t falsity: f

0 size ← t jmpSize.
2 ↑1+ (size+f) bfpSize + size

location

2 push q, bfp F
Variable q
0 2
sizeForTruth: t falsity: f

0 size ← t jmpSize.
2 ↑1+ (size+f) bfpSize + size

location

Parts of node

Variables retained to emit the pair

Sweet! Can you always address a literal w/ 1 byte?
A) NO — in real compiler this is conditional

2 passes through tree
Optimizer - measuring size of object code how much code will get generated? - encoded turns exactly size string to allocate (recursively true at every node)

how much if don't care about value & effect, just jump

either — (5. Better Effect — if throw away value)

~~jump~~ fixup

Mitchell

crossing vs strict nesting

loop exits

jump back ~~[illegible]~~

short wins

2 jumps

both long

v

both short

affecting length of code

is there clean order to

check this out on

? measure every body on

assumption of short jump

want to avoid redoing

measuring

problem not usable in language

with jumping out of very

nested situation

tradeoff -

multiple passes over

parse tree vs

over object code

mitchell idea - floating bounds up tree

Sweet did it in measure

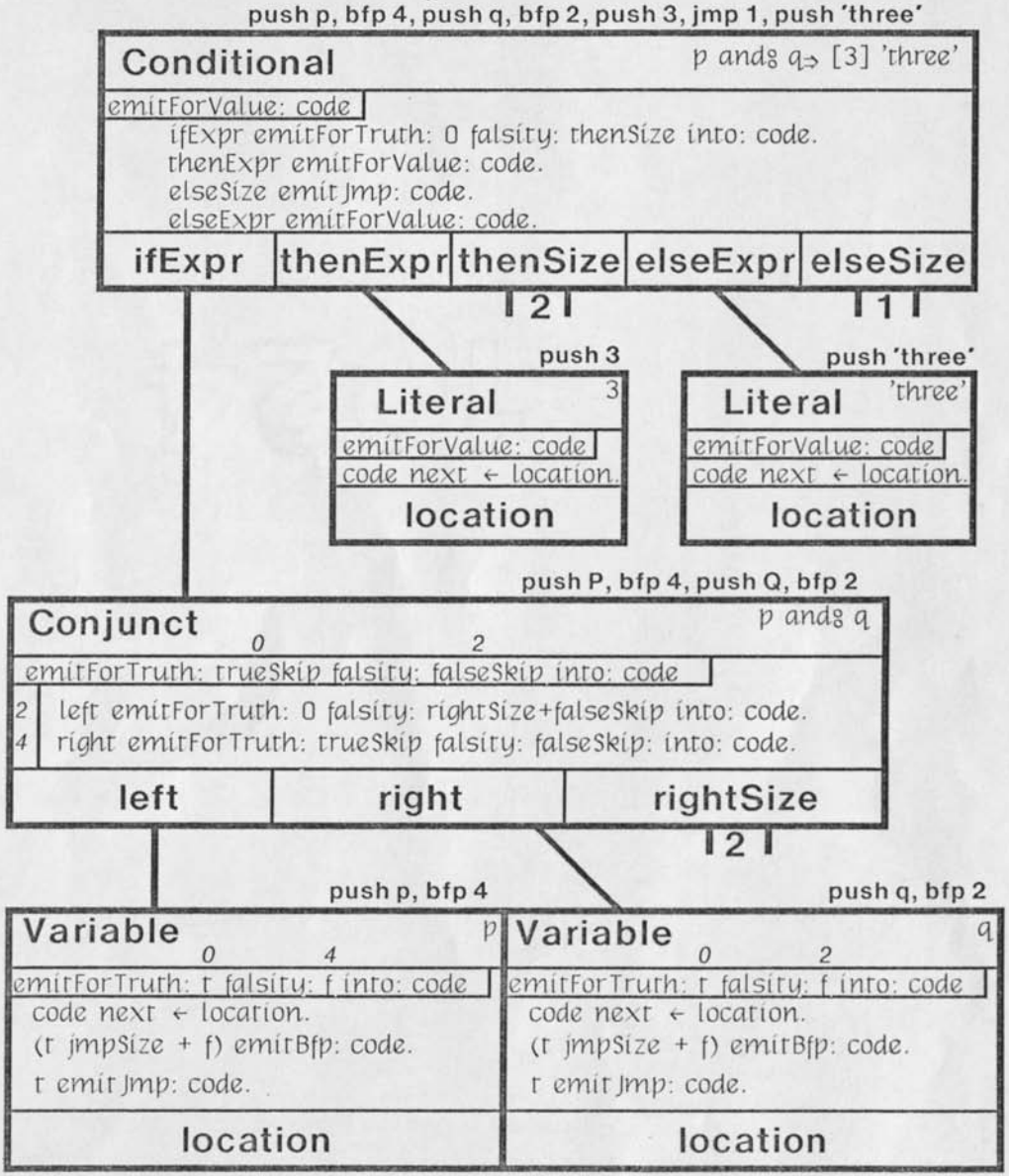
Sunny - ~~[illegible]~~ ~~better if~~

better if conditions

are rare

Trading space for time

Emitter Pass



Strategy

(1) Optimize as much as possible in the tree.

Cheaper than recopying object code.

(2) Each node measures itself,
optimizes itself,
remembers its optimization,
emits its own code.

Easier to program.

(3) Distinguish uses of result:

For value;

For effect;

Discard or repush?

For truth. (generate jumps)

8-7-78

Compiler

Draft 2 -- August 7, 1978 5:28 PM

[note to editor: I assume the following concepts are already understood by the reader: .]

The Purpose of Compilation

The compiler takes a single method expressed in the Smalltalk source language and translates it to the byte-coded object language. If it detects errors during translation, it notifies the user by inserting an error message into the source text at the point of the error. If no errors are detected, the method is installed in the specified message dictionary and immediately becomes part of the environment.

The Structure of the Compiler

The compiler has three major phases of operation: initialization, translation, and installation. Initialization builds a symbol table with the names of fields of the class and other standard names, such as `self`. It also initializes and links up the various routines needed during translation. Translation scans the source text and generates object code. Installation converts certain special methods (such as those with no code body) to a special form, installs the object code in the message dictionary, and updates the class organizer.

The translation phase is performed by five cooperating modules: the scanner, the parser, the tree builder, the optimizer, and the emitter. The scanner scans the source text and tokenizes it. The parser matches the token stream against the productions of the grammar by recursive descent. The tree builder constructs a parse tree with one node per production matched. The optimizer walks the tree finding opportunities for optimization. The emitter walks the tree emitting optimized object code.

The translator operates in three passes. In pass one, the scanner, the parser, and the tree builder march together through the input stream; the scanner and parser are coroutines and the builder is a set of subroutines of the parser. In pass two, the optimizer walks the parse tree. In pass three, the emitter walks the parse tree and appends byte codes to the output stream. Of the three passes, only the first is through the source text, and only the third is through the object code. All three passes span the parse tree, the first one building it, the second one seeking optimizations, and the third one emitting optimized code.

The time taken by a typical compilation is divided as follows:

Initialization	17%
Translation	61%
Installation	22%

Translation time is typically divided as follows:

Scanner	42%
Parser	42%
Tree Builder	6%
Optimizer	5%
Emitter	5%

These times were determined using the Smalltalk "spy" facility [see chapter Y]. Note that a large portion of the time is taken in scanning and parsing. If the method were stored and edited in parsed form, as in the program template notation (chapter T), these modules could be eliminated and the compiler would become about twice as fast. Of course, the speed advantage and the clarity of program template notation must be weighed against the conciseness and the free-form editability of linear text notation.

The Scanner

The scanner is a typical table-driven one. Each time it is asked for a token, it scans over one identifier, number, or special character and tells its coroutine (the parser) what it found by sending a message such as `integer: chars` where `chars` is a string of the characters forming a token that is an Integer literal (see Figure S/P). The scanner used by the compiler is general enough that it is used as a front end to other programs such as a code compressor.

The Parser

The parser is a typical recursive descent type. When it finds a complete syntactic construct it tells its helper (the tree-builder) what it found and the helper returns an object that shall represent that construct in enclosing constructs. (see Figure P/B) For example, during the parse of `x ← y + z`, the parser executes the following statements:

```
...
var ← helper variable: 'x'
...
rcvr ← helper variable: 'y'.
...
arg ← helper variable: 'z'.
...
expr ← helper rcvr: rcvr op: '+' args: arg.
...
stmt ← helper assign: var expr: expr
...
```

The parser is general enough to be used as a front end in other applications than compilation. All one must do is provide a helper other than the tree builder. This technique is used in a program that parses a method and displays it as a two-dimensional program template.

The Tree Builder

Each call on the tree builder returns a node of the parse tree. The node belongs to a class associated with the syntactic construct reported by the parser, e.g., `ParsedAssignment` in the last example above, `ParsedMessage` in the next-to-last. The fields of the node include the arguments of the call (e.g., `var` and `expr` in a `ParsedAssignment`).

The Optimizer

The optimizer makes a top-down scan through the tree. The scan is performed by passing a message to the root node, which passes messages to each of its descendants, and so forth. In response to the message it receives, each node returns the number of bytes of object code that it will emit. Of course, the node is able to find out from all its descendants how much code they will emit before it must compute its own amount. Thus, in a single pass through the tree, the size of the object code can be determined (see Figure OP).

During the tree walk, any node may attempt an optimization. For example, a node of class `ParsedConditional` representing `ifExpr⇒[thenExpr]elseExpr` can tell `ifExpr` to generate jumps for `and's/or's` expressions instead of generating `true/false` results. It can determine the relative destinations of the jumps before activating `ifExpr` by first measuring `thenExpr` and `elseExpr`.

Thus, each node must be ready to measure itself and emit code differently when it is in normal use or when it is the condition of a conditional. Another distinction is made as well: whether the value of the expression the node represents is to be used in further computation (wanted for value) or discarded (wanted for effect). Each node should respond to three different measuring messages:

`sizeForValue`
: returns the amount of code the node would emit to evaluate its subtree and to return the result.

`sizeForEffect: nextPush`
: returns the amount of code the node would emit to evaluate its subtree and to discard the result. The argument `nextPush` (if not `false`) is a variable that the parent node knows will be pushed onto the stack immediately after this node completes its evaluation (the statements in a block are asked their size from right to left). If the last code emitted by this node would be *store&pop var* and the next code would be *var*, then this node emits only *store* without a pop and the next node emits *var*, which has the same overall effect but is less expensive.

`sizeForTruth: trueSkip falsity: falseSkip`
: returns the amount of code the node would emit to evaluate its subtree and to jump `falseSkip` when the value is `false`, `trueSkip` otherwise.

For some classes of node, some of these messages are defined in terms of each other or are defined in superclasses.

Each node remembers certain information it computed during the optimizer pass so that it can generate the correct code during the emitter pass.

Because Smalltalk is a structured language without a `goto` statement, each node is able to measure its descendants in an order that assures that the jumps are measured after the code they jump over.

The Emitter

At the end of the optimizer pass, the compiler knows how much code will be generated by the method, and it allocates exactly the needed amount of space.

The emitter then makes a top-down scan through the tree by passing messages to the nodes. One argument of each message is a stream onto which the node must emit the amount of code it promised in the optimizer pass. Another argument to each message is an object that keeps track of the current and maximum depth of the stack that will be needed during evaluation; the node must inform that object of all pushes and pops (see Figure EM).

Each node should respond to three different measuring messages:

`emitForValue: code on: stack`
: appends bytes to `code` (informing `stack` of pushes and pops) to evaluate its subtree and to return the result.

`emitForEffect: code on: stack`
: appends bytes to `code` (informing `stack` of pushes and pops) to evaluate its subtree and to discard the result.

`emitForTruth: trueSkip falsity: falseSkip into: code on: stack`
: appends bytes to `code` (informing `stack` of pushes and pops) to evaluate its subtree and to jump `falseSkip` when the value is `false`, `trueSkip` otherwise.

As above, some of these messages are defined in terms of each other or are defined in superclasses.

Quality of Code

The compiler has no peephole optimizer and no postscans over the object code, yet it optimizes boolean expressions perfectly and optimizes out pop-push pairs. It is cheaper to walk a parse tree an extra time

than to recopy object code squeezing out unneeded bytes.

The implementation of parse tree nodes as objects means that they can retain state between passes, and that they can receive and pass different messages to evaluate their subtrees for different purposes.

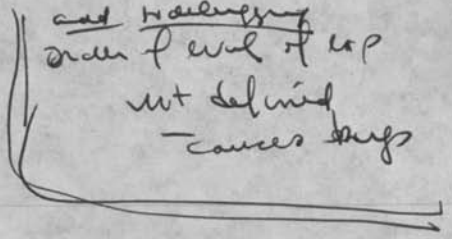
incremental compilation

⇒ tradeoff decision
in space + time

multiple edits

source code – byte codes

Differences



---

are these ops really cheaper?
wrt that language constraints?
if goto's need post pass on (3)

break
loop

Peter suggests: compiler could clean instance var
Lang gain 5 or 10%.

internal exit of while, until, for

jump optimization – most interesting
bode
short
long

ping taylor
Goldstein
on paper

1st time
(1) weak on return ops
(2) with arrow and if, then nothing more to do
(3)

[this part for Chapter ?]

An IC Design Example

During the design of an integrated circuit, a geometric layout must be produced that specifies the masks for each layer of the chip. There are a number of ways to have a computer system assist the designer in creating such a layout. This topic is discussed in [chapter 4, Mead and Conway, Introduction to VLSI Systems]. We will adopt one of these methods and show how the core of the machine-assisted portion was implemented in Smalltalk in one week by a Philosophy major with a few months of programming experience.

The method requires the designer to plot (usually by hand) each component, or "cell" of the circuit, on graph paper, using different colored pencils to outline the portions on the different layers. If the same cell is repeated in many places, it can be detailed just once and then represented by an outline of the right shape and size everywhere else.

When the plot is done, it is hand-coded into a symbolic layout language that can be interpreted by a computer program and converted to a form (CalTech Intermediate Form, "CIF") that can drive output devices such as a plotter and a pattern generator. An example of such a language is ICLIC, developed at CalTech by Maureen Stone and Ron Ayres and described in [Stone, chapter ?, ???, How-To Manual]. There is a compiler for ICLIC that Ayres developed by extending an existing compiler for the strongly typed language ICL. The ICLIC compiler compiles a program that when excuted converts the description to CIF.

We can define a set of classes in Smalltalk that serve the same role as the features of ICLIC. The most important class is class Symbol, which is an abstract class. By subclassing class Symbol, the designer can define new kinds of cells. Each subclass must have a method that specifies the layout of the cell in terms of smaller cells. It must also have or inherit a method for displaying the layout on the screen and one for converting it to CIF.

Since cells are built up out of smaller cells, there must be atomic cells from which all others are built. The atomic cells in this system are instances of class Box, which represents a rectangle on a specified layer of the chip, and class Wire, which represents a path of specified width connecting a series of points on a specified layer.

There must also be a way to compose smaller cells into larger ones. The primary constructor in this system is class CompSymbol, which is implemented as an array of symbols that represent cells of any kind. There are also some special constructors, such as class Contact, which is a stack of overlapping boxes in different layers, and class CompWire, which represents a wire that is routed between layers in mid-path via a suitable contact.

All the classes mentioned so far are subclasses of Symbol. Finally, there is class Layer, which is not. A layer helps to identify a symbol, and knows characteristics such as the default wire width. The layers are predefined in the system:

poly diff metal **etc**

Standard contacts are predefined between pairs of layers that may be connected:

pToD pToM mToD

Here are some examples of expressions that compute cells in the system:

to be copied from Kim's inspect window

Here is an example of a user-defined cell class:

Kim's pullup example

Note that when a pullup is initialized, it builds its structure and stores it in a field. Thus, when it is asked its mbb (minimum bounding box) or asked to display or to convert to CIF, it need not recompute the structure. This saves time, but costs space. A contrasting example is:

Kim's nand example

Note that when a nand gate is initialized, it only remembers its parameter, which is the width of its pulldown part. When it is asked its mbb, or asked to display or to CIF, it must compute its structure every time. This costs time, but saves space. The system gives the designer the choice of approach in each class of cell. In the original ICLIC, objects were not available, so this flexibility was lacking.

Another advantage of an object-oriented implementation over the original ICLIC is that the classes that are defined can be given other messages, such as *simulate*, and the structure that is built can then be simulated. The original ICLIC was built on a functional rather than an object-oriented language and so this opportunity for extensions was not readily available.

[this part for Debugging Chapter]

Not written yet, here are Kim and my notes on his best bugs:

(1) (Box printon) called (Stream append: rect) led to error because (Stream append:) requires a string. He should have used (Stream print: rect) or (Stream append: rect asString).

(2) There was a variable 'black' at the top level and also one in class Symbol. He was at the top level when he executed some code so he got the wrong one.

(3) uninteresting

At this point, he filed out IcDesign.st!1

(4) uninteresting

(5) he used (is: Integer) instead of (Is: Number) so a Float wasnt one

(6) He made a Path with (Path new) and forgot to init the fields, so they were all nil and something blew up

(7) He omitted a period between statements. The last thing in the first statement was a string; the first thing in the second statement was self. It tried to pass the message 'self' to the string!

(8) uninteresting

(9) uninteresting

(10) Infinite recursion. The 'array' field of WireSet was supoosed to have a Vector value and then the 'printon: strm' message was [array printon: strm]. But the 'array' field was self instead, thus the endless recursion. This happened because 'WireSet formContacts' in the simplest case did ?[] instead of ?[↑vec], so self got returned instead of vec.

(11) uninteresting

At this point, he filed out IcDesign.st!2

(12) Layer nextInCluster omitted the ↑stream so it returned itself.

(13) uninteresting

(14) uninteresting

(15) uninteresting

(16) color: black looks good but is wrong because the value of black is (and is supposed to be) a Layer. (This was a bad choice for Smalltalk in general but is copied from ICLIC where it makes perfect sense. Maybe we should call layers 'diff', 'poly', 'metal', etc instead of color names.)

At this point, he filed out IcDesign.st!3

(17) In a for-loop on i, i was a symbol, and he used it as an mbb. He should have done a for-loop on j and then (i←j mbb) before using i.

At this point, he filed out IcDesign.st!4

(18) 0-Point (in CompSymbol show) should have been 0@0-Point. Subtraction of Points and Integers is non-commutative, unfortunately.

(19) In implementing multi-layer wires, he had essentially {...~aPoint~aContact~...} and aPoint was used three times: where shown, as the first point on the next layer, and as the center of aContact. He forgot to copy aPoint to use it as the center of aContact, so when the whole shebang was rotated, the loop down the list rotated aPoint twice! This is why Peter Deutsch thinks points should be "scalars", i.e., unalterable.

(20) He divided an integer by 2 and forgot to float it first, so it got truncated, which was not desired.

(21) He forgot to initialize the field 'path' of an object.

(22) don't remember what this one is -- could replay tape

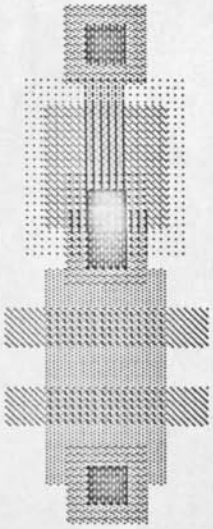
(23) (Integer * Float) got Integer result, he expected Float.

(24) something about Integer round ... ? It wasn't defined maybe?

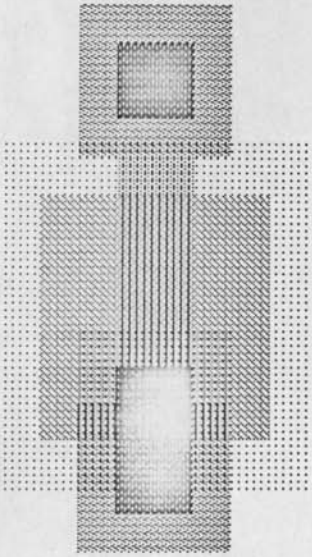
At this point, he filed out IcDesign.st!9

(25) 'someclass mbb' essentially did a 'self mbb' and got an infinite recursion

At this point, he filed out IcDesign.st!10



Icide output 8/78



Chap 4

Kernel — won't run — but must have these to be called "Smalltalk"

- Object
- Class [vLClass]
- [illegible]
- Dictionary
- Method
- Activation
- Message
- SmallInteger (for [illegible] program counter)

} whole cloth

"TOOL" — -a possible system to build on the kernel
[illegible] (see next page) (but one we don't actually use, luckily)

Blind — - will run, but no i/o
— - useful to build data structures

Obj
- HSet
  - Dict
    - SymTab
- Arr
  - vec
  - Str
    - UST
- N..
  - Int
  - Flt
- Class etc. from Kernel

Basic — - has a user & secondary storage
(1) compilers or interpreters, and thus Syntax
(2) user i/o
- (a) TTY - streams, files
- (b) display — pt, rect, [illegible], [illegible], textframe, dispframe, turtle, usrview, [illegible] windows & scheduler

1/8 — ways to get
2/8 — [illegible] marks on screen (chars, graphics)

uniform user interface
(3) Errors; processes

[illegible].

Actual — - What we use — misc. user aids [illegible]

TOOL

To Kernel, add: (Object, Class), add Integer (from Kernel)

Then add...

Class List

flds: first (car) rest (cdr)

methods: first (car) rest (cdr) atom

[to Object, add: atom ≡)

& the period instances nil & false (& true)

This gives us a lot of LISP—

lists of (lists, integers)

plus lists of classes

Define class Association (fields: key, value (name) list of properties (property list) — List aka Lisp

Define class Atom as a subclass of Association to which key is name (a List) of value is the property list (Widgets)

Define class Property List as a different List whose elements are Associations (Atom, value)

Atom methods: getProp: p

putProp: p value: v

nameIs: name

& add toothpick name to list which

Association method...

propertyList method...

Now discuss inefficiency of LL's

— leads to classes Vector, String

eventually, discard Assoc, Atom, List!

voila Smalltalk

BLUE

Object

List Array Class

VLC

Vector

vec

Smalltalk Class Outline

September 18-22, 1978
Adele Goldberg and Larry Tesler

Monday September 18

Theme What is object-oriented programming?

Reference Draft-0 of Chapter III *Smalltalk: Dreams and Schemes*

basic data structure: object
state
behavior

basic processing: message sending

Smalltalk's version
conceptual object: internal and external view
conceptual class, instances, subclasses

message determination

Examples environments to organize into objects and to specify the message protocol of the objects
what are the objects?
what are their protocols?
[car wash, amusement park]

Off-line assignment

choose one from each column and specify the objects and their protocols

bank data base	text editor
integrated circuit	calculator
animated movie	musical performance
telephone network	computer
	PARC

On-line assignment [see handout #1]

using the Smalltalk user interface, learn to text edit and print your off-line assignment

Please write down all your questions and give them to us before class so we can order them for a better organized sequence of responses.

Tuesday September 18

Theme hierarchical class organization

Reference Draft-0 Chapter IV *Smalltalk: Dreams and Schemes*

subclassing

methods

pseudo-variables self and super

examples

HashSet, Dictionary, DataDictionary
Record, BankAccount
a job shop simulation

Syntax of Smalltalk-76

special consideration for initializing class, pool, and instance variables

Classes in the Basic System

Off-line assignment (but actually done on-line) [see handout #2]

goal: do some browsing on-line to find the indicated definitions; watch out for use of self and super. Try to read the definitions of Stream, Point, and Rectangle. Read the class definitions for HashSet and Dictionary to determine what they can do. How do you create a new instance of the class Dictionary? An instance of the class Dictionary understands a message of the form insert: name with: value. In executing the method associated with this message, a number of messages to self are sent. Which ones are they and who holds the message dictionary in which each is found?

On-line assignment [see handout #3]

implement a class that represents a data structure

examples to choose from:

chained list, ordered or unordered
set
binary tree
ring

Wednesday September 20

SubTheme The nitty gritty folklore of files and printing [see handout #4]

Theme Message protocol as a command language; browsing and reading class definitions already available in Smalltalk-76

Examples

Putting text up on the screen (Textframe)
Making a sketch (BitRect); making a movie
Making a PanedWindow (PanedWindow not to be confused with PainedWindow)

On-line assignment

Using classes Vector or Set, BitRectEditor, CodeWindow, and/or ListPane in making a kind of four-paned PanedWindow of the following form

first pane	list of components, that is, classes, that can be used in retrieving information or displaying information
second pane	list of existing subclasses or instances (note one of the panes should make it possible to add new ones)
third pane	the interface to the component (typically part of the message protocol) that the user can view and/or modify
fourth pane	editable view of the implementation of the interface (possibly code or a picture)

Here's an example

First pane contains the list

clerks
salesmen
instructors
everyone

Second pane contains the names of people who are in the selected position. For example, suppose salesmen is selected in the first pane, then the second pane might show

joe brown
henry clay

Third pane contains information about the people, for example,

face
salary

Fourth pane would show either a picture of the person's face or the person's salary.

Try to write something like the above....

Thursday September 21

SubTheme reading class definitions written by your classmates!

Theme scheduling the active objects

Example Windows and Menus

On-line assignment [see handout #5]

Define the following rectangular areas:
a menu
a pictorial view of something like a circuit, office, telephone, etc
a place for defining new objects

clean up your display screen, create one or more instances of the above objects, and schedule each one such that moving the mouse inside the area "wakes the window up"

Friday September 22

SubTheme how long will your version be up-to-date?

Theme Distributed control and modularity of class design

Example

the message print
the message show
read the definitions written by the students

On-line assignment

your work!! ask for help whenever...

'From Smalltalk 5.3c on 19 September 1978 at 10:58:43 pm.'

"Movie"

```
Class new title: 'Movie'
    subclassof: Object
    fields: 'frames path myrtle'
    declare: '';
    asFollows~
```

A simple movie maker -- you create a filmstrip of n frames, sketch in those frames, define a path for the movie to traverse, and run it

create using

Movie new init: <number of frames>

you will be asked to provide the first frame

or

Movie new init: <number of frames> with: <rectangle for first frame>

Initialization

```
clear | each
    [for: each from: frames do: [each clear: white]]
init: length | rect
    ["the filmstrip is length long, the user must draw the rectangle"
     rect ← Rectangle new fromuser.
     self init: length with: rect.]
init: length with: rect| i
    ["the filmstrip is length long"
     frames ← Vector new: length.

    "first frame is rect;  rest of frames are next to it and same size"
     for: i to: length do:
        [frames o i ← rect copy.
         rect moveby: ((rect extent x + 5) @ 0).
         (frames o i) outline.].

    "create a turtle for sketching in the filmstrip"
     myrtle ← Turtle init
                    frame: ((frames o 1) origin rect: (frames o length) corner).]
```

Painting

```
copy: a into: b |
    [(frames o a) blt: ((frames o b) origin) mode: storing.]
path | turt pt
    [turt ← Turtle init.
     path ← Stream new of: (Vector new: 10).
     until: user bluebug do:
        [user redbug ⊃ [ pt ← user mp.
                         "connect lines or place turtle"
                         [path empty ⊃
                            [turt penup; goto: pt; pendn]
                            turt goto: pt].
                         path next ← pt.]]]
```

```
sketch  | "sketch with redbug, erase with yellowbug, finish with bluebug"
    [until: user bluebug do:
        [user redbug => [myrtle black; goto: (user mp - (frames•1) origin)]
         user yellowbug => [myrtle white; goto: (user mp - (frames•1) origin)]
         myrtle penup;  goto: (user mp - (frames•1) origin); pendn]]
```

Running

```
run|  "movie shows once at the fixed test area"
    [self runAt: 20@20]
runAt: loc| each "movie shows once at location loc"
    [for: each from: frames do:
        [each blt: loc mode: storing]]
travel | each "movie shows once at each point in the path"
    [for: each from: path contents do:
        [self runAt: each]]
↵
SystemOrganization classify: ↑Movie under: 'Graphical Objects'.↵
```

"PanedWindow"

```
Class new title: 'PanedWindow'
   subclassof: Window
   fields: 'panes templates title'
   declare: '';
   asFollows
```

A paned window is a Window that has subwindows (panes) that are awakened and resized in unison.

Initialization

```
title: title with: panes at: templates | pane
   [self reset. CitationReasons ← CitationReasons+1.
   fors pane from: panes dos [pane init]]
```

Window protocol

```
close | pane
   [fors pane from: panes dos [pane close].
   CitationReasons ← CitationReasons-1]
eachtime | pane
   [frame has: user mp⇒
      [user bluebug⇒[ⁿself bluebug]
      fors pane from: panes dos [pane startup]]
   self outside⇒[]
   user anybug⇒[frame has: user mp⇒[] ⁿfalse]
   user kbck⇒[user kbd. frame flash] "flush typing outside"]
enter | pane
   [super show.
   fors pane from: panes dos [pane windowenter]]
erase
   [self titlerect clear. super erase]
fixframe: f
   [ⁿRectangle new origin: f origin extent: (f extent max: 160@80)]
frame: frame "(Re)initialize my frame, and tell my panes their locations."
      | templateStream template pane
   [templateStream ← templates asStream.
   fors pane from: panes dos
      ["It would be nice to have parallel fors as in MLISP."
      template ← templateStream next.
      pane frame ← (template*frame extent /36 +frame origin inset: 1)]]
hardcopy | p
   [p ← dpO pressfile: (self title+'.press') asFileName.
   self hardcopy: p.
   p page; close]
hardcopy: p | pane
   [self showtitle.
   titleframe hardcopy: p.
   fors pane from: panes dos [pane hardcopy: p]]
kbd | pane
   [(pane ← self pickedpane)⇒ [ⁿpane kbd]]
keyset | pane
   [(pane ← self pickedpane)⇒ [ⁿpane keyset]]
leave | pane
   [fors pane from: panes dos [pane windowleave]]
outline
   [frame outline: 1]
```

```
pickedpane | pane
    [fors pane from: panes do: [pane picked→ [↑pane]]
    frame flash. ↑false]
redbug | pane
    [(pane ← self pickedpane)→ [↑pane redbug]]
show | pane
    [super show.
    fors pane from: panes do: [pane outline]]
takeCursor
    [(panes◦1) takeCursor]
title
    [↑title]
yellowbug | pane
    [(pane ← self pickedpane)→ [↑pane yellowbug]]

Pane services
vanish
    [self close; erase. user unschedule: self.]

Private
titlerect
    [↑frame origin - (2 ⊙ (DefaultTextStyle lineheight + 4)) rect: (frame corner
x⊙frame origin y) + (2⊙0)]
┘
SystemOrganization classify: ↶PanedWindow under: 'Windows'.┘
```

```
                                                    "Window"
Class new title: 'Window'
   subclassof: Object
   fields: 'frame collapsed titlepara growing exitflag '
   declare: 'titlerun border titleloc titleframe windowmenu ';
   asFollows_!
```

This is a superclass for presenting windows on the screen. Besides outlining and scheduling the frame, it includes the distribution of user events which will someday be driven by interrupts.

Initialization

```
classInit       "Window classInit"
   [border ← 2⊙2.
   titleframe ← Textframe new para: nil frame: nil.
   titleloc ← 4 ⊙(¯3-titleframe lineheight).
   titlerun ← String new: 2.
   titlerun word: 1 ← 0177401.
   windowmenu ← Menu new string:
'under
frame
close
print
printbits
']
reset
   [exitflag←true. growing←false]
```

Scheduling

```
eachtime
   [frame has: user mp⊃
      [user kbck⊃[↑self kbd]
      user anybug⊃
         [user redbug⊃[↑self redbug]
         user yellowbug⊃[↑self yellowbug]
         user bluebug⊃[↑self bluebug]]
      user anykeys⊃[↑self keyset]]
   self outside⊃[]
   user anybug⊃[frame has: user mp⊃[] ↑false]
   user kbck⊃[user kbd. frame flash] "flush typing outside"]
firsttime
   [frame has: user mp⊃ [self reset.  ↑self enter] ↑false]
lasttime
   [self leave. ↑exitflag]
```

Framing

```
clearTitle: color
   [(titleframe window inset: ¯4 ⊙¯4) clear: color]
erase
   [(frame inset: ¯2⊙¯2) clear.
   self clearTitle: background]
fixedwidthfromuser: width | a b oldframe [
   user waitnobug.
   [frame≡nil⊃[] self aboutToFrame; erase].
   a ← OriginCursor showwhile: user waitbug.
```

```
    growing ← true.
    self frame: (frame ← self fixframe: (a rect: a+(width⊙32))); show.
    CornerCursor showwhile: [
      while: (a ← user mpnext) do: [ a x ← frame corner x.
        [oldframe=nil⊃ [user cursorloc ← a max: frame corner]].
        oldframe ← frame copy.
        self frame: (frame ← self fixframe: (frame growto: a));
          moveFrom: oldframe]].
    growing ← false.
    self takeCursor]
fixframe: f [^f]
frame: f
    [frame ← self fixframe: f]
moveFrom: oldframe
    [(oldframe inset: ¯1) clear. self show]
newframe | a oldframe [
  \user waitnobug.
    [frame=nil⊃[] self aboutToFrame; erase].
  (a ← OriginCursor showwhile: user waitbug.
    growing ← true.
    self frame: (frame ← self fixframe: (a rect: a+32)); show.
    CornerCursor showwhile: [
      while: (a ← user mpnext) do: [
        [oldframe=nil⊃ [user cursorloc ← a max: frame corner]].
        oldframe ← frame copy.
        self frame: (frame ← self fixframe: (frame growto: a));
          moveFrom: oldframe]].
    growing ← false.
    self takeCursor]
outline
    ["Clear and outline me."
    frame outline]
show [
    self outline.
    growing⊃[]
    self showtitle]
showtitle
    [titleframe put:
      (Paragraph new text: self title runs: titlerun alignment: 0)
      at: frame origin+titleloc.
    titleframe outline]
takeCursor
    ["Move the cursor to my center."
    user cursorloc ← frame center]
title [^'unoccupied']
```

Default Event responses

```
aboutToFrame
    ["My frame is about to change.  I dont care."]
bluebug
    [windowmenu bug
      =1⊃[^exitflag ← false];
      =2⊃[self newframe. self enter];
      =3⊃[self close. self erase.
        user unschedule: self. ^false];
      =4⊃[self hardcopy];
      =5⊃[self print]]
```

```
close []
enter [self show]
hardcopy [frame flash]
kbd [user kbd. frame flash]
keyset [frame flash]
leave []
outside [↑false]
print
   [(dpo pressfile: (self title + '.press.') asFileName)
      screenout: frame scale: PressScale]
redbug
   [frame flash]
yellowbug
   [frame flash]
]
SystemOrganization classify: #Window under: 'Windows'.]
Window classInit]
```

```
'From Smalltalk 5.3ie on 21 September 1978 at 10:21:59 am.'_!

                                                    "MovieInkPane"

Class new title: 'MovieInkPane'
   subclassof: Object
   fields: 'window frame ink'
   declare: '';
   asFollows_!

I am a pane to select an ink color for sketching

Initialization
initIn: window
   [ink ← ¯1]

Pane protocol
close
eachtime
   [frame has: user mp⇒
      [user bluebug⇒ [↑false]
      user redbug⇒
         [ink ← ¯1 - ink.
         self show.
         window ink: ink.
         user waitnobug]]
   ↑false]
firsttime
   [self picked⇒ [self enter] ↑false]
frame ← frame
init
lasttime
   [self leave]
outline
   [frame outline]
picked
   [↑frame has: user mp]
takeCursor
   [user cursorloc ← frame center]
windowenter
   [self show]
windowleave

Private
enter
   [frame flash]
leave
show
   [frame color: ink mode: storing]
_!
SystemOrganization classify: #MovieInkPane under: 'Movie'._!
```

"MovieSketchPane"

```
Class new title: 'MovieSketchPane'
   subclassof: Object
   fields: 'window frame sketch myrtle'
   declare: 'sketchMenu ';
   asFollows_[
```

I am a pane for sketching a movie frame

Initialization

```
classInit
   [sketchMenu ← Menu new string:
'clear
copy
paste
cut
run']
initIn: window
```

Pane protocol

```
close
eachtime
   [frame has: user mp⊃
      [user bluebug⊃ [↑false]
      user yellowbug⊃ [self operate]
      [user redbug⊃ [myrtle pendn] myrtle penup].
      myrtle goto: user mp - frame origin]
   ↑false]
firsttime
   [self picked⊃ [self enter] ↑false]
frame ← f
   [frame ← f inset: 2⊙2.
   sketch ← Form new extent: frame extent.
   myrtle ← Turtle init frame: frame]
init
lasttime
   [self leave]
outline
   [frame outline]
picked
   [↑frame has: user mp]
takeCursor
   [user cursorloc ← frame center]
windowenter
   [self show]
windowleave
```

Private

```
enter
   [frame flash.
   self show.
   window ink=black⊃ [myrtle black] myrtle white]
leave
   [self save]
operate
   [sketchMenu bug
```

```
        =1⇒ [self clear];
        =2⇒ [self copy];
        =3⇒ [self paste];
        =4⇒ [self cut];
        =5⇒ [self save; run]]
save "save the dots"
    [sketch fromrectangle: frame]
show
    [self outline; showAt: frame origin]
showAt: pt
    [sketch displayat: pt effect: storing clippedBy: user screenrect]
```

Menu commands

```
clear
    [frame outline]
copy
    [window buffer ← sketch]
cut
    [self copy; clear]
paste
    [sketch ← window buffer.
    self show]
run
    [window runThrough: self]
⌟
SystemOrganization classify: ↶MovieSketchPane under: 'Movie'.⌟
MovieSketchPane classInit⌟
```

"MovieViewPane"

```
Class new title: 'MovieViewPane'
   subclassof: Object
   fields: 'window frame path'
   declare: '';
   asFollows_!
```

I am a pane for showing a movie along a path

Initialization

```
initIn: window
   [path ← frame center inVector]
```

Pane protocol

```
close
eachtime
   [frame has: user mp⇒
      [user bluebug⇒ [↑false]
      user redbug⇒ [self path]]
   ↑false]
firsttime
   [self picked⇒ [self enter] ↑false]
frame ← frame
   [path ← frame center inVector]
init
lasttime
   [self leave]
outline
   [frame outline]
takeCursor
   [user cursorloc ← frame center]
windowenter
   [frame outline. self showPath]
windowleave
```

Viewing

```
run: sketchPanes | pt
   [for: pt from: path do:
      [frame outline.
      self run: sketchPanes at: pt]]
run: sketchPanes at: pt | sp
   [for: sp from: sketchPanes do:
      [sp showAt: pt]]
```

Private

```
enter
   [frame flash]
leave
path | pt s myrtle
   [frame outline.
   s ← (Vector new: 10) asStream.
   myrtle ← Turtle init frame: frame.
   while: user redbug do:
      [[s empty⇒ [myrtle penup] myrtle pendn].
      pt ← user mp.
      myrtle goto: pt-frame origin.
```

```
        s next ← pt].
    s empty⊃ [path ← frame center inVector]
    path ← s contents.]
picked
    [ΠframeMP has: user mp]
showPath | myrtle pt
    [myrtle ← Turtle init.
    myrtle penup; goto: path∘1.
    for: pt from: path do: [myrtle pendn; goto: pt]]
┘
SystemOrganization classify: ↶MovieViewPane under: 'Movie'.┘
```

"MovieWindow"

```
Class new title: 'MovieWindow'
   subclassof: PanedWindow
   fields: 'ink'
   declare: 'buffer ';
   asFollows
```

I am a movie window with a pane for viewing it, a number of panes for sketching frames, and a pane for selecting the ink color for sketching

Initialization

```
default | i temps "init with 9 frames in the strip"
   [temps ← (Vector new: 11) asStream.
   temps next ← 0@0 rect: 36@18.
   temps next ← 0@18 rect: 36@22.
   for: i to: 9 do:
      [temps next ← (4*(i-1))@22 rect: (4*i)@36].
   self init: 9 in: temps contents]


init: length in: templates | each "init with the specified length+2 templates"
   [ink ← black.

   "Create and acquire the panes"
   self title: 'Movie'
      with: (MovieViewPane new, MovieInkPane new concat:
         ((Vector new: length) transform: each to: MovieSketchPane new))
      at: templates.

   "Let user specify locations"
   self newframe; show.

   "Initialize them"
   for: each from: panes do: [each initIn: self].

   "Start it up"
   user restartup: self]
```

Sketching

```
buffer "the form in the buffer"
   [↑buffer]
buffer ← form "copy form into the buffer"
   [buffer ← form recopy]
ink
   [↑ink]
ink: ink
```

Viewing

```
runThrough: lastPane | viewPane firstSketchPane "show the movie"
   [viewPane ← panes@1. firstSketchPane ← 3.
   viewPane run: (panes@(firstSketchPane to: (panes find: lastPane)))]
```

```
SystemOrganization classify: #MovieWindow under: 'Movie'.
```

```
'From Smalltalk 5.3ie on 21 September 1978 at 8:35:31 pm.'_!

                                                    "MovieWindow"

Class new title: 'MovieWindow'
   subclassof: PanedWindow
   fields: 'ink brush'
   declare: 'buffer ';
   asFollows_!

I am a movie window with a pane for viewing it running, a number of panes for
sketching frames, and a pane for selecting the ink color and pen width  for
sketching

Initialization
default "init with 9 frames in the strip"
   [self init: 9]
init: length | each    "length frames (up to 36) in the strip"
   [ink ← black.  "initial ink color for sketching"
   brush ← 1. "initial pen width for sketching"

   "Create and acquire the panes"
      self title: 'Movie'
            with: (MovieViewPane new, MoviePalettePane new concat:
               ((Vector new: length) transform: each to: MovieSketchPane new))
            at: (self templates: length).

   "Let user specify frame"
      self newframe; show.

   "Initialize them"
      for: each from: panes do: [each initIn: self].

   "start me up"
      user topWindow leave.
      user restartup: self]

Parameters for sketching
brush
   [↑brush]
brush: brush
buffer     "the form in the buffer"
      [↑buffer]
buffer ← form     "copy form into the buffer"
      [buffer ← form recopy]
ink
   [↑ink]
ink: ink

Viewing the movie
runThrough: lastPane eraseWith: whiteSketch "show the movie"
      | pane sketches viewPane firstSketchPane
   [viewPane ← panes@1.
   firstSketchPane ← 3.
   sketches ← (panes@ (firstSketchPane to: (panes find: lastPane))) transform:
pane to: pane sketch.
   viewPane run: sketches, whiteSketch]
```

```
Private
templates: length | temps width i "the pane proportions on a 36@36 scale"
    [temps ← (Vector new: length+2) asStream.
    temps next ← 0@0 rect: 36@18.  "top half is view pane"
    temps next ← 0@18 rect: 36@22.  "thin stripe for ink pane"
    width ← (36/length) asInteger. "width of each sketch pane"
    for: i to: length do:
        [temps next ← (width*(i-1))@22 rect: (width*i)@36].
    ↑temps contents]


SystemOrganization classify: #MovieWindow under: 'Movie'.
```

"Pane"

```
Class new title: 'Pane'
   subclassof: Object
   fields: 'window frame'
   declare: '';
   asFollows
```

This class handles the protocol expected by the class PanedWindow.
A PanedWindow contains one or more Panes, scheduling them in a
manner similar to the top-level window scheduling.
Subclass this class to implement a typical pane.
Note that the pane knows its own location on the screen (frame).
It also knows what window it is in. The window also references the pane, so
there is a a cycle which must be broken when the window is closed.

Initialization

```
init "gets called right after the pane is created"
initIn: window "gets called a while later after frame← and show"
```

Pane protocol

```
close "break cycles"
   [window ← nil]
eachtime "this superclass implements no command language"
   [self picked ⇒ "is the cursor 'in' this pane?"
      [user kbck ⇒ [↑self kbd] "keyboard activity"
       user bluebug ⇒ [↑self bluebug] "commands for the paned window as a
whole"
       user yellowbug ⇒ [↑self yellowbug] "commands for the pane itself"
       user redbug ⇒ [↑self redbug] "selection in the pane"
       user anykeys ⇒ [↑self keyset] "keyset activity"
       ↑self inside] "in the pane, but no activity"
   ↑self outside] "outside the pane"
firsttime
   ["should the pane wake up?
    response to self picked determines the answer"
    self picked ⇒ [self enter] ↑false]
frame ← frame "determine the border of the pane; subclass
                might want to initialize any instance variables
                that depend on the frame"
lasttime "the pane has gone to sleep, needs some last minute housecleaning"
   [self leave]
outline
   [frame outline]
picked
   ["a typical reason for waking up the pane is that the cursor is in it"
    ↑frame has: user mp]
takeCursor "put the cursor inside the pane"
   [user cursorloc ← frame center]
windowenter "The PanedWindow distributes this message to its panes"
   [self show]
windowleave "The PanedWindow distributes this message to its panes"
```

Event defaults

```
bluebug "yield control to the paned window"
   [↑false]
enter "Indicate to the user that this particular pane has been entered"
```

```
    [frame flash]
inside
kbd
    [user kbd. "discard keystroke"
    frame flash]
keyset
    [frame flash]
leave "This particular pane has been left"
outside "yield control"
    [↑false]
redbug
    [frame flash]
show "each class of pane should define what it means to display"
yellowbug
    [frame flash]
]
SystemOrganization classify: #Pane under: 'Movie'.]
```

"MoviePalettePane"

```
Class new title: 'MoviePalettePane'
   subclassof: Pane
   fields: 'ink brush'
   declare: '';
   asFollows↵
```

I am a pane to select an ink color and a pen width for sketching

Initialization
initIn: window

Event protocol
redbug

```
   [[window ink=black⇒ [window ink: white; brush: 4] window ink: black; brush:
1].
    self show.
    user waitnobug]
```

show

```
   [frame color: window ink mode: storing]
↵
SystemOrganization classify: #MoviePalettePane under: 'Movie'.↵
```

"MovieSketchPane"

```
Class new title: 'MovieSketchPane'
   subclassof: Pane
   fields: 'sketch myrtle'
   declare: 'sketchMenu ';
   asFollows_!
```

I am a pane for sketching a single movie frame

Initialization

```
classInit      "my command language is executed from a menu"
   [sketchMenu ← Menu new string:
'clear
copy
paste
cut
run']
```

Aspects

sketch

```
   [↑sketch]
```

Pane protocol

frame ← f

```
   [frame ← f inset: 2⊙2.
    sketch ← Form new extent: frame extent. sketch white.
    myrtle ← Turtle init frame: frame.
    myrtle penup.]
```

Event protocol

enter

```
   [frame flash.
   self show.
   myrtle width: window brush.
   [window ink = black ⊃ [myrtle black] myrtle white].
   myrtle penup.
   self trackCursor]
```

inside

```
   [myrtle penup. self trackCursor]
```

leave

```
   [self save]
```

redbug

```
   [myrtle pendn. self trackCursor]
```

show

```
   [frame outline.
   sketch displayat: frame origin effect: storing clippedBy: frame]
```

yellowbug

```
   [sketchMenu bug
      =1 ⊃ [self clear];
      =2 ⊃ [self save; copy];
      =3 ⊃ [self paste];
      =4 ⊃ [self cut];
      =5 ⊃ [self save; run]]
```

Private

```
save "save the dots"
    [sketch fromrectangle: frame]

trackCursor
    [myrtle goto: user mp - frame origin]
```

Menu commands

```
clear
    [frame outline]
copy
    [window buffer←sketch]
cut
    [self copy; clear]
paste
    [sketch ← window buffer.
     self show]
run | eraser
    [eraser ← Form new extent: frame extent. eraser white.
    window runThrough: self eraseWith: eraser]
┘
SystemOrganization classify: #MovieSketchPane under: 'Movie'.┘
MovieSketchPane classInit┘
```

"MovieViewPane"

```
Class new title: 'MovieViewPane'
   subclassOf: Pane
   fields: 'path myrtle'
   declare: '';
   asFollows↵

I am a pane for showing a movie along a path

Initialization
initIn: window
   [path ← frame center inVector]

Viewing the movie
run: sketches | pt "sketches is a Vector of Forms"
   [frame outline.
   forş pt from: path doş
      [self run: sketches at: pt]]

Pane protocol
frame ← frame "done before initIn:"
   [path ← frame center inVector.
   myrtle ← Turtle init frame: frame]
windowenter
   [self show]

Event protocol
bluebug
   [ñfalse]
enter
   [self show]
redbug | pt s
   [frame outline.
    s← (Vector new: 10) asStream.
    whileş user redbug doş
    [[s empty ⊃ [myrtle penup] myrtle pendn].
      pt ← user mp.
      self track: pt.
      s next ← pt].
     s empty ⊃ [path← frame center inVector]
     path ← s contents.]
show | pt
   [frame outline.
    myrtle penup.
    forş pt from: path doş [self track: pt. myrtle pendn]]

Private
run: sketches at: pt | sketch
   [forş sketch from: sketches doş
      [sketch displayat: pt-(sketch extent/2) effect: storing clippedBy: frame]]
track: pt
   [myrtle goto: pt-frame origin]
↵
SystemOrganization classify: ↷MovieViewPane under: 'Movie'.↵
```

1. Why don't the successive messages sent to 'Class new' require semicolons between them?

2. 'Point' has a message 'x←' whose method is 'x←x', according to Browser. Don't understand syntax.

3. In Browser, is there an easy way to find out what superclass accepts a particular message that is no
**t accepted by the current class? Is there a way to find out what classes accept a particular message
**? (e.g., I couldn't figure out who accepts 'hash').

4. I wasn't able to decide whether Vectors were 0-origin or 1-origin. I guessed zero, and apparently I
** was wrong. If you browse far enough down into the definition of Vector you end up diving into micro
**coded primitives that don't seem to be documented.

"SortedStream"

```
Class new title: 'SortedStream'
   subclassof: Object
   fields: 'list current length'
   declare: '';
   asFollows
```

A linear list of objects that are sorted according to a quantity called "rank". The objects might be anything, as long as they respond to the message "rank" with an integer value. The list will be kept sorted according to these values.

This class can not be replaced by Stream, because Stream recognises 61 different message types, while SortedStream only recognises 4. Sorted stream should therefore be easier to use for the novice.

Trygve Reenskaug - Sept -78.

Initialization

```
init: maxLength
   [list ← Vector new: maxLength.
   current ← 0.
   length ← maxLength]
```

Read - Write

```
current
   [↑[(current≤length and: current>0)⊃[list⊙current]nil]]
first
   [current ← 1.
   ↑self current]
next
   [current ← current + 1.
   ↑self current]
next←object | i
   [i ← length.
   while: (i>0 and: list⊙i=nil) do: [i ← i-1]. "finds last significant element"
   i=length ⊃ [user notify: 'list full']
      while: (i>0 and: ((list⊙i) rank > object rank)) do:
         [list⊙(i+1) ← list⊙i.
         i ← i-1]. "move large objects up one position"
      list⊙(i+1) ← object]
```

Illegal

```
,x          [user notify: 'illegal message to this class']
argsOff: stack      [user notify: 'illegal message to this class']
asVector      [user notify: 'illegal message to this class']
emitForValue: code on: stack   [user notify: 'illegal message to this class']
firstPush       [user notify: 'illegal message to this class']
length          [user notify: 'illegal message to this class']
max         [user notify: 'illegal message to this class']
nail      [user notify: 'illegal message to this class']
printon: strm      [user notify: 'illegal message to this class']
remote: generator       [user notify: 'illegal message to this class']
sizeForValue      [user notify: 'illegal message to this class']
unNail      [user notify: 'illegal message to this class']
```

"TestSortedStream"

```
Class new title: 'TestSortedStream'
   subclassof: Object
   fields: 'rank'
   declare: '';
   asFollows↵
```

Tests objects of class SortedStream

```
Operation
init: rank
rank [↑rank]

Testing
printon: strm
   [strm append: 'rank='+ rank asText]
testing
   ["
   list ← SortedStream init: 5.
   list next←TestSortedStream init: 3.
   list next←TestSortedStream init: 5.
   list next←TestSortedStream init: 9.
   list next←TestSortedStream init: 7.
   list next←TestSortedStream init: 8.
   list next←TestSortedStream init: 3.

   list first    list next      list nail


   list inspect

"]
↵
SystemOrganization classify: ↶TestSortedStream under: 'Current'.↵
```

Sorted would be better single for lookups, unions & intersections but worse for what Btrees? Sparse sorted?

'From Smalltalk 5.3ie on 20 September 1978 at 6:13:20 pm.'

"Bucket"

```
Class new title: 'Bucket'
  subclassof: Dictionary
  fields: 'cardinal ordinal'
  declare: '';
  asFollows
```

Buckets are structured sets in which multiple instances of an element may reside.
Union and intersection are defined as the max and min # of elements respectively.

Initialization

```
init: size | t "initialize counts of number of elements and number of distinct elements to 0"
  [super init: size. cardinal←ordinal←0.]
```

Insertions and deletions

```
delete: element "Tests if element in set, decrements cardinality and ordinality, then invokes delete within Dictionary"
  [self lookup: element⇒
       [cardinal←cardinal-1.
        ordinal←ordinal-(self∘element).
        super delete: element
       ]
  ]
extract: element | t "Check if element in bucket, take one instance of element out of bucket by decrementing count and invoking delete if necessary"
  [self lookup: element⇒
       [ (t←self∘element)>1⇒[self∘element←t-1.
                             ordinal←ordinal-1]
            self delete: element
       ]
  ]
insert: element | t "Put one instance of element in to bucket by incrementing count and calling upon Dictionary insert if necessary"
  [ordinal←ordinal+1.
   t ← self lookup: element ⇒
          [self∘element←(t+1)]
           cardinal←cardinal+1.
           super insert: element.
           self∘element←1
  ]
```

Set operations

```
hasMember: element "test if 'element' is in my bucket"
  [self lookup: element⇒ [⇑true] ⇑false]
intersection: bucket2 | bucket1 element t card ord "for each element in both buckets,put it into the resultant bucket1 with count value = minimum of count1 and count2"
  [bucket1←bucket2. ord←card←0.
   for: element from: bucket2 do:
     [t←self lookup: element⇒
```

```
        [card←card+1. bucket2⊙element>t⇒
                        [bucket1⊙element←t.ord←ord+t]
                         ord←bucket2⊙element+ord
        ]
       bucket1 delete: element
      ]
   bucket1 cardinality←card; ordinality←ord.
   ⇑bucket1
   ]
isEmpty "test for empty bucket"
   [cardinal=0⇒[⇑true]⇑false]
union: bucket2 | bucket1 element t card ord "for each element in either
bucket,put it into the resultant bucket1 with count value = maximum of
count1 and count2"
   [bucket1←self.
    card←self cardinality.
    ord←self ordinality.
    for: element from: bucket2 do:
      [t←self lookup: element⇒
         [bucket2⊙element>t⇒
         [bucket1⊙element←bucket2⊙element.
          ord←ord+(bucket2⊙element)-t.]]
       bucket1 insert: element.
       bucket1⊙element←bucket2⊙element.
       card←card+1.
       ord←ord+(bucket2⊙element)
      ]
   bucket1 cardinality←card; ordinality←ord.
   ⇑bucket1
   ]
+ bucket "union (maximum) of two buckets"
   [self union: bucket]
* bucket "intersection (minimum) of two buckets"
   [self intersection: bucket]
```

Counting

```
cardinality "returns number of distinct elements in bucket"
   [⇑cardinal]
cardinality ← integer "sets number of distinct elements"
   [cardinal ← integer]
ordinality "returns total number of (indistinguishable) elements in bucket"
   [⇑ordinal ]
ordinality ← integer "sets total number of (indistinguishable) elements"
   [ordinal ← integer]
┘
SystemOrganization classify: ↗Bucket under: 'Sets and Dictionaries'.┘
```

Example Organizations -- First Ideas (distributed September 26, 1978)

The first assignment in the Smalltalk class was to specify the objects and their protocols for various suggested situations. Several members of the class turned in their assignment. With only minor editing, they are presented here for your consideration and possible implementation.

Bank Data Base

The following is a basic set of specifications for one bank database assuming that it consists of a file of customers (or customer account numbers) and, for each, a current account balance. JNo history of the account is preserved.

```
class Database
subclassof: Dictionary

  withdraw: amount                    returns "done" or "insufficient funds"
    from: account
  deposit: amount
    into: account
  total: account                      returns the current balance in the specified account
```

Second suggestion distinguishes amoung the kinds of accounts. Perhaps a superclass BankAccount could be usefully described. The accounts are not inter-related, nor are they aware of the actual bank in which they exist.

```
class SavingsAccount

  newAccount                          assign new account number to customer, balance
                                      is 0
  makeDeposit: amount                 add amount to balance of account, put in date of
                                      deposit
  makeWithdrawal: amount              compute interest, add interest to balance, see if
                                      enough funds. Renegotiate on insufficient funds;
                                      otherwise subtract amount from balance, date the
                                      withdrawal, and give to customer
  balanceBooks                        periodically compute interest by account number,
                                      credit to account number, show deposits as + to
                                      bank assets, show interest and withdrawals as
                                      liabilities
  updateStatement                     periodically or on customer demand print
                                      transactions and interest and balance for
                                      customer
  closeAccount                        compute interest, add interest to balance, give
                                      total to customer and invalidate account number

class CheckingAccount
```

subclassof: SavingsAccount — *except interest is 0 in Calif*

printChecks — *initially and on custem demand print blank checks, make withdrawal of cost from account*

class LoanAccount

newAccount — *credit check, assign new account number, balance to amount of loan, date and interest and terms of loan record*

makePayment — *subtract amount from balance, if less than due foreclose, if more than due reduce principal, if excess per terms levy fine, if some late levy fine, if very late foreclose, if balance is now 0 close, enter date and payment amounts*

balanceBooks — *periodically add up balances on loans and add to bank liabilities*

updateStatement — *periodically or on customer demand print transactions and interest and balance for customer*

billAccount — *periodically per terms bill customer*

foreclose Account — *check payments against terms, foreclose if necessary*

Third suggestion focuses on the user of the database. However, this is actually an "account" being described, rather than a "customer", since a customer might have more than one account.

class Customer
fields: accountNumber, customerName, accountType, balance, balanceDayProduct

deposit: amount — *add an amount to the balance of the account*

withdraw: amount — *subtract an amount less than or equal to the balance*

dailyUpdate — *compute new balanceDayProduct, etc*

monthlyUpdate — *produce monthly statement*

Animated Movie

The Movie and MovieWindow examples presented in class are further ideas about how to do an animated movie.

```
class EditableComposite

  init                                  initialize to empty composite
  add: part                             add part to composite
    at: position
  delete: part                          remove previously added part

class Film
subclassof: EditableComposite

uses superclass to add or delete scene with respect to a scene number

  newFilm: title                        initialize to empty film
  view: startFrame                      look at film
    at: speed

class Scene
subclassof: EditableComposite

uses superclass to delete previously added layer

  add: cel                              add layer to scene
    on: levelNumber
    at: coordinates
  setDuration: frameCount               specify length in frames

class Cel
subclassof: EditableComposite

uses superclass to add figure at specific coordinates

class Figure
subclassof: EditableComposite

  addLineFrom: relativeTrajectory1      add line with own motion
    to: relativeTrajectory2
  addRectangleUpperLeft: relativeTrajectory1   fill a solid area
    lowerRight: relativeTrajectory2

class RelativeTrajectory

  setOrigin: xyPair                     starting point
  setMotion: motionFunction             take function mapping time into location relative
```

position: time

to origin
return location as function of time

Calculator

class Display

show: string

class Keyboard

depress: keyId
currentKey

class Microprocessor

turnOn

clockTick — *Read keyboard, compute if necessary, update display if not current*

Text Editor

Handling input and editing

class Character

show
characterClass: classTable — *returns class of character from table*

class String

show
copy: string
length — *returns length*
concatenate: string
compare: string — *returns true or false*
search: string — *returns position of first character, or false*
insert: character
atCharacterNumber: position
remove: firstCharacterNumber
length: length
get: charNumber
put: character

class Word
subclassof: String

makeIntoIndexWord
hyphenation — *returns list of positions for hyphenation*
hyphenate: position

class Paragraph

paragraph — *value is a special character*

output control

class Document

file Keyls: string
chapter: numbercode
heading: heading — *indicates start of new chapter*
pageOfCharacter: character — *returns page number for given character*
illustration: page
position: rectangle

class Font

class Illustration

class CrossReference

crossReference: word

class Footnote

footnote: string
onPage: pageNumber

class MailingList

class Date

currentDate

class Index

produceIndex *returns a string*

```
class Transistor

  changeGate: point
  changeSource: point
  changeDrain: point
  scrunch: scrunchList

class Contact

  newSides: point1 and: point2

class Instance

  changeConnection: point

class Line

  split
  doesItCross: otherLine

class EndPoint

  x
  y
```

Computer

This is a stack-oriented computer. (Note, there are seveeral ways to compress the number of classs definitions shown here.)

```
class Controller

  executeAt: loc

class InstructionRegister

  giveValue
  setValue: val

class ProgramCounter

  giveValue
  setValue: val
```

processor section

```
class ALU

  add
  subtract
  and
  or
  xor
  zero
  conditionCode                    gives the current code

class Stack

  push: val
  pop                              is first operands for ALU

class OperandRegister              second operand for ALU

class ZeroRegister                 read-only registers

  giveValue

class OneRegister                  read-only register

  giveValue

class MinusOneRegister             read-only register

  giveValue
```

non-processor section

```
class Memory

  giveValue: loc
```

Outline of Questions and Decisions to Think About and Decide About

Issues

Book
Smalltalk Interpeter
Smalltalk storage management
Epist'logical design ideas
(messages, primitive classes, naming conventions,...)
Metaphors
(e.g., form,path,image)
Information Storage and Retrieval
(basis for the interpreter, help, communications)
User Interface
New Hardware
(Ethernet connection for the NoteTaker)

Time considerations
pre-moratorium
moratorium
post-moratorium

The idea of a moratorium is that during some time period (suggestion is 6 months):

*no new machine code that is to be supported will be written

*no new releases of any Smalltalk (Alto or NoteTaker)

*paper design and short-term expendable software tests might be done
a first draft on an epist'logical design will be created

*a first draft of the book ~~will be completed~~

During the pre-moratorium period, no new interpreter will be designed and no new metaphors tested (these are for post-moratorium and moratorium periods). Doug would like strategy for Ethernet hardware to be completed in the pre-moratorium period. OK

~~Major Questions that help in making decisons~~

what do you expect to get from a NoteTaker Smalltalk?
i.e., what do you want, expect to learn?
what sort of demonstration of hardware do you anticipate needing?
what kinds of leverage would you like for investing in future--marketing, software development, Dynabook prototyping?

~~To be considered~~

1. Need to have a "demo" working of the hardware in order to

--give a computing forum (Doug)
--show the NoteTaker to Shugart or Diablo (Chris, Doug)

Claim this is done because we can use BitBlt so can blt characters from keyboard strikes and can read/write image from the disk. Doug agrees that he has what he needs for a computing forum.

2. Need to have a Smalltalk working on the NoteTaker or do we? before the moratorium

--we need to get on with a house cleaning soon
--but 10 NoteTakers will be here by end of January and, without a Smalltalk, we take a chance on the machines being idle or someone implementing some other language (Mesa? could be worse-Fortran) on the NoteTaker before a Smalltalk is done

~~Ways of getting a first Smalltalk on the NoteTaker in pre-moratorium period~~

Model A
transfer Alto Smalltalk to NoteTaker

Model B
create a new Smalltalk for the NoteTaker

Model C
???

Claim: Models B and C belong in post-moratorium period. The following are ways to do Model A.

~~Aspects of the transfer to consider~~

I. same interpreter (2 week job, mostly done already)
II. simple file system (1 week job)
III. storage management (designed already, 1 week job)
IV. proper subset of existing classes (being done now, new choice is function of metaphors and is ruled out of pre-moratorium period)
V. text (code escapes currently handled by short machine code -- 10 liners -- will be rewritten but 5,6,7 are "biggies")
VI. floating point
VII. line drawing

~~Transfer packages to consider~~

We agree that I-IV is correct and pick from one of the text, floating point, and line drawing options.

text options

1. write the current routines in Smalltalk
write new machine code in which the primitive for text is
2. paragraphs (second easiest to do as is translantion of Diana's current code, least flexible, fastest)
3. lines (hardness equal to runs)
4. runs (second most flexible, second slowest)
5. characters (easiest since is practically just BitBlt, most flexibility but least speed)

floating point options

1. leave it out
2. if there exists Intel software, grab it, otherwise
3. use Sproull's code, translated
4. transcribe 8080 code
5. design from scratch

line drawing options

1. leave it out
2. if have floating point, do it in Smalltalk
3. do it in Smalltalk with single precision
4. do it in machine code

~~Some Plans to Add Ideas To~~

~~Plan A. have no NoteTaker Smalltalk and call moratorium December 1~~

Plan B. do text, floating point, and line drawing options 1. Call moratorium with this slower than molasses Smalltalk. Have parallel effort (by Bruce?) to do different V,VI,VII so that a faster Smalltalk is done during or at end of moratorium.

Plan C. do Smalltalk as in B but call a split moratorium so that certain key people start the moratorium late in order to do text option 2, floating point option 4, and line drawing 2. This might take an extra month for, say, Dan, Diana and Bruce working together.

~~Issues not raised in the above~~

what really has to be rewritten in Alto Smalltalk to check out form, path, image metaphor? wasn't there talk of redoing text machine code to eliminate textframe in favor of textimage?

how can we use the moratorium to assist dissertations (John, Al, Steve, Ted????)

can we control demons, visitors better?

7) demo-driven
MUNdane

~~demo~~

both u.i. & impl.

5) doing for other people / teachable / bookable - what Kim said / [illegible]

2) ~~getting~~ get it running before Mead - CSL < promises / knowledge

releases
3) ~~[illegible]~~ / design - - length of moratorium - - impl time - summer?

& book & theses
4) interpreter will last than moratorium - { agree w. John / Thinglab } & fragility - S.T. as exec.

6) 4 yrs from now / 2 yrs from now - { implementor-driven / free reign

7) 2 systems - - the Winchester Mystery House (or Bldg 35!) vs. Laura + Sean's house (or Bldg. 33) { user-driven / q.a.

~~theses~~
($) { ic-design / FPI

TS

Tiny Storage Management

Storage Management Functions

The storage management section of any programming environment must allocate and deallocate storage for objects.

In Tiny Smalltalk, a new object is allocated explicitly (e.g., when the message `new` is sent to a class) but there is no explicit language construct that causes an object to be deallocated. Instead, the storage manager must identify objects that have become *inaccessible* from the program and deallocate them automatically, a task that is traditionally known as *garbage collection.*

In many storage management systems, an object once allocated at a certain memory location stays at that location until it is deallocated. To allocate a new object in such systems, it is necessary to find a "hole" between existing objects large enough to hold the new object. After a while, such systems tend to "fragment" or "checkerboard" memory; that is, there is a lot of unallocated space in the memory, but it is fragmented into pieces too small to satsify allocation requests. Theoretical explorations of this problem have proven that no allocation strategy can reduce it to acceptable proportions [ref. McCreight's source].

The simplest way to repair a fragmented memory is to *compact* object storage. All objects that are still in use are moved towards one end of memory, squeezing out all the free space between them, and leaving one large unallocated block at the other end. Adopting this strategy, Tiny Smalltalk compacts storage after every garbage collection. The *compacting garbage collector* is a space-thrifty deallocation technique on a small computer.

Rack Allocation

Tiny Smalltalk employs a simple allocation technique we call *rack allocation.* A rack is similar to a stack in that all new allocation occurs at the top. To allocate *n* words of storage, the top-pointer is simply incremented by *n.* However, unlike a stack, deallocation can -- and usually does -- occur in the middle of the rack.

The trouble with a rack allocator is that the rack will become full as soon as the top-pointer reaches the end of available memory. When that happens, the rack must be *compacted.* In Tiny Smalltalk, this is accomplished by invoking the compacting garbage collector.

During compaction, when an object is moved, it is necessary to update all pointers to its rack storage. To make this update inexpensive, only one pointer to an object's rack storage is kept. That pointer is kept in an *object table* (OT). References to the object from the rest of storage must be indirected through the OT. Thus, the *ordinary object pointers* (OOPs) found in Smalltalk objects are really pointers to the OT, in which pointers into the rack are in turn found.

Deallocation Speed

Although the rack allocator can allocate an object very quickly, the compacting garbage collector has a severe performance handicap. When there is very little unallocated space and substantial allocation/deallocation activity, very few allocations have a chance to occur between successive compactions, and the overhead of the compacter becomes unacceptably high.

One solution would be to defer compaction as long as possible by allocating objects in the middle of the storage area when sufficient free space can be found there. Techniques for doing so can be found in [Knuth], but they lose the simplicity inherent in the rack approach.

Tiny Smalltalk keeps compactions from occurring too frequently by declaring memory to be "exhausted" whenever the proportion of time spent during compaction exceeds the time spent between compactions. The typical effect is to keep the rack from getting more than about 90% full, thus reducing storage efficiency by about 10%.

A Simple Object Format

A simple format for an object stored in the rack is:

Header

OOP of this object
SIZE of this object (including header)
oop of this object's CLASS

Data

first field or element
...
last field or element

In this format, the actual data of the object is preceded by a three word *header* that contains the object's OOP, its storage size (including header), and its class. In our Intel 8086 implementation, the SIZE is always an even number of bytes.

The OOP of the object is present in the header so that when the compacter moves the object it can quickly find the OT entry to update.

The SIZE of the object is present so that the compacter can know how many words to move when the object is relocated.

The CLASS of the object is present so that when a message is sent to the object the correct message dictionary can be consulted.

Objects are of varying sizes, but in typical Smalltalk programs they average about 10 words. Thus, a three word header and a one word OT entry constitute a large space overhead to pay for every object. Later in this chapter, we will show how to reduce that overhead. For now, the simplicity of the three word header will make it easy to program the allocator and the compacting garbage collector.

Addressing Conventions

In our 8086 implementation of Tiny Smalltalk, every object header and every OT entry starts at an even byte location in the rack. This convention has no cost, because every ROT entry and every rack object has an even number of bytes. The advantages are that 8086 memory bus access is faster when words are at even byte locations, and that all OOPs and OT entries have a

low order bit of zero that can be used as a flag bit by the storage management system.

Each data word of the object contains a field of the object (or an element, if the object is an array). The low order bit of the field is used as a flag bit. If the flag bit is zero, then the field contains the OOP of an object. If the flag bit is one, then the field contains an instance of class Integer, stored as immediate data in the remaining 15 bits. This representation of integers limits their range to $\pm$16K, and forces the interpreter to make special checks to distinguish OOPs from integers, but it also saves a lot of memory space (an OT entry, a three word header, and a word of data) for each of these very common objects.

The OT lies in the low 32K bytes of the 64K address space. This convention frees the high order bit of an OOP to be used as a flag bit. The high order bit of the CLASS field of a header is sometimes used in that way.

The entire address space is partitioned as follows, in order of *ascending* address:

Machine Language programs (interpreter, etc.)
OT (fixed size)
Rack (grows upward)
Stack (grows downward on the 8086)

A Simple Allocator

To allocate an object, it is necessary to obtain sufficient space for the header and the data from the next available location `h` at the top of the rack and one word from the next available location `oop` in the OT. Location `oop` is made to point at `h`, and the three words starting at `h` are filled in with the header information.

Checks must be performed for insufficient room in the rack and for an exhausted OT. If either condition occurs, compaction is invoked and the allocation is retried. If the allocation fails again, the system stops and reports an error to the user.

To make the finding of a free OT entry rapid, all free entries are kept on a linked list headed at `nextOop`. The top of the rack is at `nextSpace` and the end of available storage is at `endSpace`.

The fields of a header are accessed by offsets from the first word, as follows:

```
oopX     =0    Offset of the OOP field
sizeX    =2    Offset of the SIZE field
classX   =4    Offset of the CLASS field
```

The allocation algorithm is:

```
allocate: bytes class: class  | newNext oop h
    [newNext ← nextSpace + bytes. "proposed new top of rack"
    [newNext > endSpace or: nextOop = 0⇒  "rack or OT full?"
        [self gc.  "run the compacting garbage collector"
        newNext ← nextSpace + bytes.  "new top of rack"
        newNext > endSpace or: nextOop = 0⇒  "rack or OT still full?"
            [self outOfMemoryError  "hopeless -- stop and report"]]].
    oop ← nextOop.  "obtain a rot entry"
    nextOop ← rot word: nextOop.  "remove it from the free list"
    h ← nextSpace.  "address of header"
    nextSpace ← newNext.  "new top of rack"
    rot word: oop ← h.  "make rot entry point at header"
    rack word: (h+oopX) ← oop.  "fill oop into header"
    rack word: (h+sizeX) ← bytes.  "fill size into header"
    rack word: (h+classX) ← class.  "fill class into header"
    ↑oop  "return the oop"]
```

A Simple Compacter

Assume that the garbage collector has marked every accessible object by setting a *mark bit.* In our Intel 8086 implementation, the mark bit is the normally zero low order bit of the normally even OT entry. The job of the compacter is to sweep through the rack from the bottom to the top, squeezing all accessible objects together and updating their OT entries. For the benefit of subsequent allocation, it should also link the OT entries of inaccessible objects onto the free-list, and clear the area of the compacted rack above the top to all nil.

The compacter maintains two pointers into the rack: si and di. The pointer si points at the header of an object that is currently being considered for keeping or discarding. The pointer di points at the place further down the rack to which the object will be moved if it is kept.

If the bottom of the rack is at beginSpace, the compaction algorithm is:

```
compact | si di words oop i
    [si ← di ← beginSpace.  "both pointers start at the beginning"
    whileS si ≤ endSpace doS  "for each object"
        [words ← (rack word: si+sizeX) / 2.  "its size in words"
        oop ← rack word: si+oopX.  "its oop"
        (rot word: oop bit: markBit) = 0⊃  "is it unmarked?"
            [si ← si + (2*words).  "inaccessible, so skip over it"
            rot word: oop ← nextOop.  nextOop ← oop  "add to free
            list"]
        "The object is accessible"
        rot word: oop ← di.  "update OT entry"
        si=di⊃  "will distance moved be zero?"
            [di ← si ← si + (2*words)  "just advance the pointers"]
        forS i from: 1 to: words doS  "for every word in the object"
            [rack word: di ← rack word: si.  "move the word"
            di ← di + 2. si ← si + 2  "on to the next word"]].
    forS i from: di to: endSpace by: 2 doS  "nil out free rack space"
        [rack word: i ← 0]]
```

A Simple Garbage Collector

The job of the garbage collector is to set the mark bit of just those objects that are accessible. This is most easily done by a recursive search from the "roots of the world", namely, the interpreter's stack and the global variable table. To make things easier, the collector first pushes the OOP of the global variable table onto the stack, and pops it off when it is done; thus, all the roots of the world are in the stack. It also marks the high order bit of that stack entry as a *stop bit* so it can tell when it gets there. Then it points a roving index ri just past the base of the stack (highest address in the 8086) and executes the algorithm described in the next paragraph.

For each oop stored before location ri and after the first preceding location whose stop bit is set, do the rest of this paragraph. Follow the oop to the OT and check whether the object is marked. If it is already marked, there is nothing to do. Otherwise, set the mark bit of the

object, set the stop bit in the last header word of the object, save ri, repoint ri just past the last data word of the object, execute the algorithm described in this paragraph, and restore ri. When the stop bit is detected, reset the stop bit and terminate the algorithm.

The algorithm of the preceding paragraph can be implemented by the following method, which accomplishes the saving and restoring of ri by its recursive behavior:

```
gc: ri  | oop h
    [until: ((oop ← rack word: (ri←ri-2)) bit: stopBit) = 1 do: "for all data"
        [(oop bit: intBit) = 0⊃  "is it an oop (not an integer)?"
            [(rot word: oop bit: markBit) = 0⊃  "is it unmarked?"
                [h ← rot word: oop.  "address of the header"
                rot word: oop bit: markBit ← 1.  "set mark bit"
                rack word: (h+classX) bit: stopBit ← 1.  "set stop
                bit"
                self gc: h + (rack word: h+sizeX).  "recur"]]].
    rack word: ri bit: stopBit ← 0  "reset mark bit"]]]]
```

This simple collection algorithm is recursive and therefore must use a stack to execute. The depth of the stack must be greater than the longest chain of pointers in memory. This requirement is hard to satisfy when memory space is tight, which is the situation whenever the collector runs.

A Non-Recursive Collector

To avoid using a stack during garbage collection, the previous value of ri can be remembered in the current object, by storing it in the header field usually reserved for the OOP. Of course, the OOP of the current object must be restored to the header after the current object is all marked. This is easily done, because when the collector returns to ri it will find the current object's oop there, that being how it got to the current object in the first place. This sort of trick is common in garbage collectors [see Deutsch, et al, etc.].

The non-recursive garbage collection algorithm is shown below. Parts that are different from the recursive version are underlined. The value of hdrSize is 6, the number of bytes in the header.

```
gc: ri  | oop h
```

```
[whileS true doS  "repeat until return executed below"
    [untilS ((oop ← rack word: (ri←ri-2)) bit: stopBit) = 1 doS  "for all data"
        [(oop bit: intBit) = 0⇒  "is it an oop (not an integer)?"
            [(rot word: oop bit: markBit) = 0⇒  "is it unmarked?"
                [h ← rot word: oop.  "address of the header"
                rot word: oop bit: markBit ← 1.  "set mark bit"
                rack word: (h+oopX) ← ri.  "save ri"
                rack word: (h+classX) bit: stopBit ← 1.  "set stop
                bit"
                ri ← h + (rack word: h+sizeX).  "recur"]]].
    rack word: ri bit: stopBit ← 0  "reset mark bit".
    ri > endSpace⇒ [↑true  "stack all marked: done!"]
    h ← ri -hdrSize +2.  "the address of the header"
    ri ← rack word: (h+oopX).  "restore ri to the pointer that got us here"
    rack word: (h+oopX) ← rack word: ri  "restore oop to header"]]
```

A Two-Word Header Format

As was pointed out earlier, the overhead of a three word header and an OT entry is significant when objects are on the order of ten words in size. Here we will show how to reduce every header to two words. In Chapter [Ooze], we will show how the average header size can be reduced even further.

The word we will remove from the header is the OOP field. That field is not needed by the interpreter, only by the compacter, which uses it to find the OT entry when it changes the rack location of the object. Just before the compacter runs, we will slip the OOP into the header of every object, overwriting the CLASS field. When the object is moved by the compacter, we will put the class back into the CLASS field. We need a place to keep the class in the meantime; let's keep it in the OT entry, overwriting the rack pointer. When the object is moved, we will have to change the OT entry to point at the object's new location -- but we had to do that anyway!

The new object format is:

Header

SIZE of this object (including header)

CLASS (except during gc: oop of this object)

Data

first field or element

...

last field or element

The allocator is the same as before except that the following statement is deleted:

```
rack word: (h+oopX) ← oop.  "fill oop into header"
```

The compacter changes a little more. Assume as before that the garbage collector has marked every accessible object by setting the mark bit. Also assume that the OOP of every object has overwritten the CLASS field of its two-word header, and that the OT entry has been overwritten by the object's class. The mark bit shares the OT entry with the class, and the stop bit will share the header entry with the OOP.

The compaction algorithm is the same as the two-word header version, except that the offsets classX and oopX have the same value (namely, 2), and, right after the comment:

```
"The object is accessible"
```

the following statement must be added:

```
rack word: si+classX ← (rot word: oop) lxor: 1.
```

which restores the class field to the header from the OT entry, stripping off the mark on the way.

If we use the recursive garbage collector shown earlier, then between marking and compaction a step is required that installs the OOP into headers and the class into OT entries. This step is called *OT reversal*, since it makes headers point at OT entries instead of the usual situation in which OT entries point at headers. The algorithm is:

```
preCompact  | oop h mark
    [for: oop from: beginOT to: endOT do:  "for all OT entries"
        [h ← rot word: oop.  "either rack pointer or free list link"
        h > endOT⇒  "is it a rack pointer?"
            ["reverse the entry"
```

```
mark ← rot word: oop bit: markBit.
rot word: oop ← (rack word: (h+classX)) lor: mark.
rack word: (h+classX) ← oop]]]
```

A Non-Recursive Collector for Two-Word Headers

If we wish to use a nonrecursive collector, the one shown earlier won't work with two word headers, because it uses the OOP field of the header to remember ri while the object is being marked. Instead, we'll use the following approach.

Before starting to process the object, overwrite the CLASS field with ri. This takes care of remembering ri, but we'd better not forget the object's class, so let's keep it in the OT entry.

After the stop bit has been encountered and processing of the object is complete, recover ri from the CLASS field and follow it back to the pointer that got us to this object. That pointer is the OOP of this object. We could set things back in their original places, but let's instead just overwrite the CLASS field with this OOP, so that the precompaction step will not have to do that. The precompacter will not have to put the class in the OT entry either, because it is already there. Thus, all accessible OT entries will be reversed as a side effect of garbage collection, without having to perform a separate reversal step! The inaccessible entries still will have to be dealt with in a separate step.

The garbage collection algorithm is the same as the non-recursive collector except: (1) oopX and classX both have the value 2; (2) hdrSize has the value 4; (3) the final statement must set the mark bit in the CLASS/OOP field of the header so that the compacter can distinguish objects that are free; (4) the statement that zeroed the stop bit in the CLASS/OOP field can be deleted as superfluous because the final statement overwrites that whole field.

The free-list maker is best run between garbage collection and compaction. It is the same as the OT reverser except that the statements that began with the comment:

```
"reverse the entry"
```

are replaced by:

```
"add to free list"
rot word: oop ← nextOop.
nextOop ← oop
```

The compacter must change slightly to omit the free list update and to find the mark in the header instead of the OT, because there is no OOP in the header for a free object. The statements:

```
(rot word: oop bit: markBit) = 0⇒  "is it unmarked?"
    [si ← si + (2*words).  "inaccessible, so skip over it"
    rot word: oop ← nextOop.  nextOop ← oop  "add to free
    list"]
```

are changed to:

```
(oop bit: markBit) = 0⇒  "is it unmarked?"
    [si ← si + (2*words).  "inaccessible, so skip over it"]
```

Non-Pointer Objects

The simple object format of Tiny Smalltalk is not particularly space efficient, but since its uniformity makes the system software small and simple, this inefficiency can generally be forgiven. There is one class of object for which the inefficiency is intolerable, namely, character strings. There are usually many strings in storage, and when stored one character per word, they are very wasteful of space. We also would like to treat byte-coded methods as regular Smalltalk objects, and thus would like to store them two byte-codes per word in the rack.

To store such objects more efficiently, we will allow an alternative storage format in which the data part of an object packs two bytes into a word. When there are an odd number of bytes of data, the last byte of the last word will be zero (a slight waste of space), and the SIZE field of the header will be odd.

The element-accessing primitives for byte-per-element classes must be aware of the special storage method and convert between full-word and half-word representations.

The elements of byte-per-element objects are always small integers, and never OOPs. Therefore, the garbage collector need not process their fields in search of further accessible objects. To let the collector know that an object contains no OOPs, the allocator sets the high order (stop) bit of the CLASS field of the header, and the collector notices its presence and leaves it set.

The allocator must be changed as follows. The `bytes` argument is allowed to be be odd. The `class` argument must have the high order bit set if the fields are to be non-OOPs. The

statement:

```
newNext ← nextSpace + bytes.
```

must be followed by:

```
[newNext isOdd⇒ [newNext ← newNext+1]].
```

The garbage collector is changed as follows. After the statement:

```
rot word: oop bit: markBit ← 1.  "set mark bit"
```

is added:

```
(rack word: (h+classX) bit: stopBit) = 1⇒ []
```

which bypasses processing of the object's fields if they are known to contain no OOPs.

The precompacter and compacter are unchanged.

The 8086 assembly language versions of all the storage management algorithms are in Appendix {whatever].

Note that a class could be defined that is like a string but whose elements are 16-bit integers instead of 8-bit integers, as long as the element-accessing primitives know about the special format and as long as the high order bit of the CLASS field is set to warn the garbage collector.

```
                        1   ;        BCA TinyInterp.bca; RESUME SMALL.BOOT
                        2   ;        BCA/E TinyInterp.bca; GYPSY
                        3   ;        BCA/L TinyInterp.bca; EMP TinyInterp.ls;DEL TinyInterp.ls
                        4   ;
                        5   ;       This file contains the TINY SMALLTALK INTERPRETER
                        6   ;       Author: Dan Ingalls (made tiny by Kim McCall)
                        7   ;       Last changed: June 6, 1979  7:48 PM
                        8   ;
                        9           .PREDEFINE      "8086PREDEFS.SR"
                       10           .PREDEFINE      "EXTERNALS1.BCA"
                       11   ;
3000                   12           .LOC    INTERPRETER
3000 43 30             13           .ADR    NEXT    ;ENTRY POINTS
3002 79 31             14           .ADR    IRETN
3004 D9 30             15           .ADR    SENDAX
                       16   ;
3006 FE 03             17   STOREMODE:      .ADR    ILLOOP  ;*ILLOOP OR OOP TO STORE
                       18   ;
                       19   ;
                       20   ;       /*TABLES WHICH HAVE TO COME AT BEGINNING
                       21   BYTETABLE:      ;DISPATCH TABLE FOR 16 BYTE CATEGORIES
3008 94 30             22           .ADR    LDINST
300A 5E 30             23           .ADR    LDTEMP
300C 71 30             24           .ADR    LDLIT
300E 5D 30             25           .ADR    BADBYTE
3010 7F 30             26           .ADR    LDLITI
3012 5D 30             27           .ADR    BADBYTE
3014 5D 30             28           .ADR    BADBYTE
3016 5D 30             29           .ADR    BADBYTE
3018 B4 30             30           .ADR    SUNDRY
301A 5D 30             31           .ADR    BADBYTE
301C 92 31             32           .ADR    LJMP
301E 5D 30             33           .ADR    BADBYTE
3020 5D 30             34           .ADR    BADBYTE
3022 CD 30             35           .ADR    SEND
3024 5D 30             36           .ADR    BADBYTE
3026 5D 30             37           .ADR    BADBYTE
                       38   ;
                       39   CONTRL:         ;DISPATCH TABLE FOR CONTROL OPERATIONS
3028 B8 30             40           .ADR    STOPOP
302A BF 30             41           .ADR    STONP
302C C3 30             42           .ADR    POPS
302E 76 31             43           .ADR    RETURN
3030 5D 30             44           .ADR    BADBYTE
3032 5D 30             45           .ADR    BADBYTE
3034 5D 30             46           .ADR    BADBYTE
3036 C7 30             47           .ADR    LSELF
3038 5D 30             48           .ADR    BADBYTE
303A 5D 30             49           .ADR    BADBYTE
303C 5D 30             50           .ADR    BADBYTE
303E 5D 30             51           .ADR    BADBYTE
3040 5D 30             52           .ADR    BADBYTE
                       53   ;
```

```
                                    54    ;       /* PUSH VALUE IN BX FROM PREVIOUS OPERATION */
3042 53                             55    PNEXT:  PUSH    BX
                                    56    ;
                                    57    ;       /* INSTRUCTION FETCH */
3043 8A 1D                          58    NEXT:   MOV     BL,0!DI ;DI IS PC CORE ADDRESS
3045 FF C7                          59            INC     DI
3047 8B F3                          60            MOV     SI,BX
3049 81 E6 0F 00                    61            AND     SI,#0F
304D D1 E6                          62            SHL     SI      ;SI ← LOW 4 BITS SHIFTED FOR WORD OFFSET
304F 81 E3 F0 00                    63            AND     BX,#0F0
3053 D1 EB                          64            SHR     BX
3055 D1 EB                          65            SHR     BX
3057 D1 EB                          66            SHR     BX      ;BX ← HI 4 BITS SHIFTED FOR WORD OFFSET
3059 FF A7 08 30                    67            JMPI    BYTETABLE!BX    ;DISPATCH INTO BYTE TABLE
                                    68    ;
                                    69    ;       /* BAD BYTE */
305D CC                             70    BADBYTE:  INT3
                                    71    ;
                                    72    ;       /* LOAD TEMP */
305E 8B DD                          73    LDTEMP: MOV     BX,BP
3060 8B 46 04                       74            MOV     AX,NARGS!BP
3063 2D 03 00                       75            SUB     AX,#3   ;2*NARGS - 2
3066 F7 DE                          76            NEG     SI      ; - (+ AFTER NEXT INSTR) IF THIS IS AN ARG
3068 2B F0                          77            SUB     SI,AX
306A 78 32                          78             JS     LDORSTORE
306C 83 C6 0A                       79            ADD     SI,#SELF-STEMP1
306F EB 2D                          80             J      LDORSTORE
                                    81    ;
                                    82    ;       /* LOAD LITERAL */
3071 8B 5E 06                       83    LDLIT:  MOV     BX,METHOD!BP    ;ADDR OF CODE BASE
3074 8B 9F 00 10                    84            MOV     BX,ROT!BX
3078 03 F3                          85            ADD     SI,BX           ;INDEX OF LITERAL
307A 8B 5C 08                       86            MOV     BX,CMLIT0!SI    ;LOAD LITERAL
307D EB C3                          87             JMP    PNEXT
                                    88    ;
                                    89    ;       /* LOAD LITERAL INDIRECT */
307F 8B 5E 06                       90    LDLITI: MOV     BX,METHOD!BP    ;ADDR OF CODE BASE
3082 8B 9F 00 10                    91            MOV     BX,ROT!BX
3086 03 F3                          92            ADD     SI,BX           ;INDEX OF LITERAL
3088 8B 5C 08                       93            MOV     BX,CMLIT0!SI    ;GET LITERAL
308B 8B 9F 00 10                    94            MOV     BX,ROT!BX
308F 8E 06 00                       95            MOV     SI,#ORREF       ;OFFSET OF OBJ REF
3092 EB 0A                          96             J      LDORSTORE
                                    97    ;
                                    98    ;       /* LOAD INST FIELD */
3094 8B 5E 08                       99    LDINST: MOV     BX,SELF!BP
3097 8B 9F 00 10                   100            MOV     BX,ROT!BX
309B 83 C6 06                      101            ADD     SI,#OBFLD1
                                   102    ;
                                   103    ;
                                   104    ;       /* ALL LOADS,  BX = BASE,  SI = OFFSET
309E A1 06 30                      105    LDORSTORE:      MOV     AX,STOREMODE
30A1 3D FE 03                      106            CMP     AX,##ILLOOP
30A4 75 04                         107             JNE    DOSTORE
30A6 8B 18                         108            MOV     BX,0!BX!SI
30A8 EB 98                         109             JMP    PNEXT
                                   110    ;
30AA 89 00                         111    DOSTORE: MOV    0!BX!SI,AX      ;EXCHANGE NEW VALUE W/PREV
30AC C7 06 06 30 FE 03             112            MOV     STOREMODE,##ILLOOP      ;RESET STORE-MODE
30B2 EB 8F                         113             JMP    NEXT
                                   114    ;
                                   115    ;       /* SUNDRY */
30B4 FF A4 28 30                   116    SUNDRY: JMPI    CONTRL!SI
                                   117    ;
30B8 5B                            118    STOPOP: POP     BX      ; /* STORE AND POP
30B9 89 1E 06 30                   119    STP1:   MOV     STOREMODE,BX
30BD EB 84                         120             JMP    NEXT    ;STORE FLAG IS SET
                                   121    ;
30BF 5B                            122    STONP:  POP     BX      ; /* STORE, NO POP
30C0 53                            123            PUSH    BX
30C1 EB F6                         124             J      STP1
                                   125    ;
30C3 5B                            126    POPS:   POP     BX      ; /* POP
30C4 E9 7C FF                      127             JMP    NEXT
                                   128    ;
```

```
30C7 8B 5E 08          129   LSELF:  MOV     BX,SELF!BP
30CA E9 75 FF          130           JMP     PNEXT
                       131   ;
                       132   ;
                       133   ;       /* SEND MESSAGE */
30CD 8B 5E 06          134   SEND:   MOV     BX,METHOD!BP    ;ADDR OF CODE BASE
30D0 8B 9F 00 10       135           MOV     BX,ROT!BX
30D4 03 F3             136           ADD     SI,BX
30D6 8B 44 08          137           MOV     AX,CMLIT0!SI    ; AX ← MESSAGE SELECTOR
30D9 5B                138   SENDAX: POP     BX              ;RECEIVER
30DA 53                139           PUSH    BX
30DB F7 C3 01 00       140           TEST    BX,#INTBIT      ; BX ← OOP OF DICT OF OBJ IN BX
30DF 74 05             141            JZ     NOTINT
30E1 BB 12 00          142           MOV     BX,#CLINTEG
30E4 EB 07             143           J       HAVCLS
30E6 8B 9F 00 10       144   NOTINT: MOV     BX,ROT!BX
30EA 8B 5F 04          145           MOV     BX,CLWORD!BX
30ED 8B 9F 00 10       146   HAVCLS: MOV     BX,ROT!BX
30F1 8B 5F 0C          147           MOV     BX,CMDICT!BX
30F4 8B 9F 00 10       148           MOV     BX,ROT!BX
30F8 8B F7             149           MOV     SI,DI
30FA 8B 7F 06          150           MOV     DI,MDSELS!BX    ;SELECTOR VECTOR
30FD 8B BD 00 10       151           MOV     DI,ROT!DI
3101 8B D7             152           MOV     DX,DI           ;SAVE FOR LATER SUBTRACTION
                       153   ;       ... GET LENGTH AND SCAN FOR SELECTOR
3103 8B 0D             154           MOV     CX,LENWORD!DI
3105 FC                155           CLD
3106 D1 E9             156           SHR     CX      ;LENGTH IN WORDS INCL HEADER
3108 F2                157           REP
3109 AF                158           SCAW
310A 87 FE             159           EXCHG   DI,SI
310C 74 01             160           JZ      FOUND
                       161   ;
310E CC                162   MESSFAIL:       INT3    ;NOT YET IMPLEMENTED
                       163   ;
310F 83 EE 02          164   FOUND:  SUB     SI,##2
3112 2B F2             165           SUB     SI,DX
3114 8B 5F 08          166           MOV     BX,MDVALS!BX    ;MD METHODS
3117 8B 9F 00 10       167           MOV     BX,ROT!BX
311B 8B 18             168           MOV     BX,0!BX!SI      ;INDEX BY SI TO GET METHOD
311D 8B B7 00 10       169           MOV     SI,ROT!BX
3121 8B 4C 06          170           MOV     CX,CMHDR!SI     ;METHOD HEADER
3124 53                171           PUSH    BX              ;METHOD OOP
3125 8B C1             172           MOV     AX,CX
3127 25 0F 00          173           AND     AX,#NAMSK
312A D1 E0             174           SAL     AX
312C FF C0             175           INC     AX      ;MAKING IT A SMALLTALK INTEGER
312E 50                176           PUSH    AX      ;NARGS
312F 8A C5             177           MOV     AL,CH   ;(CONTAINS IPC)
3131 25 7E 00          178           AND     AX,#IPCMSK
3134 8B 5E 06          179           MOV     BX,METHOD!BP
3137 8B 9F 00 10       180           MOV     BX,ROT!BX
313B 2B FB             181           SUB     DI,BX
313D D1 E7             182           SAL     DI
313F FF C7             183           INC     DI      ;MAKING IT A SMALLTALK INTEGER
3141 57                184           PUSH    DI
3142 D1 E5             185           SAL     BP
3144 FF C5             186           INC     BP      ;MAKING IT A SMALLTALK INTEGER
3146 55                187           PUSH    BP
3147 8B EC             188           MOV     BP,SP   ;NEW FRAME POINTER
3149 8B FE             189           MOV     DI,SI
314B 03 F8             190           ADD     DI,AX   ;SO NOW DI IS PC
314D 8B D1             191           MOV     DX,CX   ;STILL HAVE THE HEADER
314F D1 F9             192           SAR     CX
3151 D1 F9             193           SAR     CX
3153 D1 F9             194           SAR     CX
3155 D1 F9             195           SAR     CX
3157 81 E1 1F 00       196           AND     CX,#NTMSK
315B 74 06             197            JZ     CKPRIM
315D BB 00 00          198           MOV     BX,##NIL
3160 53                199   NILTMP: PUSH    BX      ;STORE NIL IN ALL TEMPS
3161 E2 FD             200           LOOP    NILTMP
3163 85 D2             201   CKPRIM: TEST    DX,DX
3165 78 03             202            JS     DOPRIM  ;DO PRIMITIVE FIRST, IF INDICATED
3167 E9 D9 FE          203           JMP     NEXT
                       204   ;
```

```
                        205     ;       /* DISPATCH TO PRIMITIVE CODE
316A 8B 75 FE           206     DOPRIM: MOV     SI,-2!DI        ;FIND PRIM INDEX AT IPC-2
316D D1 E6              207             SHL     SI      ;WORD INDEX
316F 8B 46 08           208             MOV     AX,SELF!BP      ;AX← RECEIVER
3172 FF A4 42 37        209             JMPI    PRIMTABLE!SI
                        210     ;
                        211     ;       /*RETURN */
3176 58                 212     RETURN: POP     AX      ;VALUE TO BE RETURNED
3177 8B E5              213             MOV     SP,BP
3179 5D                 214     IRETN:  POP     BP      ;RESTORING OLD STACK REFERENCE POINT
317A D1 ED              215             SHR     BP              ;WAS ST INTEGER
317C 5F                 216             POP     DI      ;OLD METHOD PLACE
317D D1 EF              217             SHR     DI              ;WAS ST INTEGER
317F 8B 5E 06           218             MOV     BX,METHOD!BP
3182 8B 9F 00 10        219             MOV     BX,ROT!BX
3186 03 FB              220             ADD     DI,BX   ;DI NOW POINTS AT NEXT BYTE
3188 59                 221             POP     CX      ;ST INTEGER FOR NARGS
3189 83 C1 03           222             ADD     CX,#3   ;ABOVE,SHR,+2(Meth & Self),SHL(Wds)
318C 03 E1              223             ADD     SP,CX   ;ELIMINATING ARGS, SELF, & METHOD
318E 50                 224             PUSH    AX      ;PUSH NEW VALUE
318F E9 B1 FE           225              JMP    NEXT    ;AND RESUME EXECUTION
                        226     ;
                        227     ;       /* LONG JUMPS */
3192 8A 05              228     LJMP:   MOV     AL,0!DI ;PICK UP NEXT BYTE
3194 FF C7              229             INC     DI      ;AX ← DELTA LOW BITS
3196 83 FE 10           230             CMP     SI,#10  ;AX=DELTA, SI<16 => UNCONDITIONAL
3199 7C 06              231              JL     DOJMP
319B 5B                 232             POP     BX
319C 83 FB 02           233             CMP     BX,#FALSE
319F 75 0E              234              JNE    NOTFALSE
31A1 8B CE              235     DOJMP:  MOV     CX,SI
31A3 D0 F9              236             SAR     CL
31A5 80 E1 07           237             AND     CL,#7
31A8 80 E9 04           238             SUB     CL,#4
31AB 8A E1              239             MOV     AH,CL   ; + BIAS*256
31AD 03 F8              240             ADD     DI,AX
31AF E9 91 FE           241     NOTFALSE: JMP   NEXT
                        242     ;
                        243     ;
                        244             .END
```

```
compile: code ...
    [...
    whileS self actOnNextToken doS [].
    ↑...]

actOnNextToken  | first word
    [whileS [first ← sourceStream peek⇒ [first isDelim] ↑false] doS
        [sourceStream next].
    first>071⇒
        [first isLetter⇒
            [word ← self nextWord.
             word last isAlphanumeric⇒ [self variable: word] self selector: word]
        sourceStream next.
        first=0136⇒
            ["↑" methodStream next ← 0203]
        first=0175⇒
            ["]" methodStream last≠0203⇒ [methodStream next←0161; next←0203]]
        self illToken: first inString]
    first≥060 orS first=025⇒
        ["digit or high minus" self emit: lLit literal: self nextWord asInteger]
    sourceStream next.
    first=056⇒
            ["." methodStream next ← 0202]
        =047⇒
            [""" self emit: lLit literal: (sourceStream upto: 047). sourceStream
next]
    self illToken: first inString]

nextWord  | s first
    [s ← Stream default.
    untilS (first ← sourceStream next) isDelim doS [s next ← first].
    ↑s contents]

findOrInsertLit: lit
    [same as before, but use lit instead of (obj←self wordAsObj)]

emit: kind literal: lit
    [methodStream next ← kind + (self findOrInsertLit: lit)]

variable: word  | b w ref
    [(b ← locals lookup: word)⇒ [self next ← b]
    (b ← ctrls lookup: word)⇒ [self perform: b]
    w ← word unique.
    (ref ← classVars lookupRef: w) orS (ref ← Smalltalk lookupRef: w)⇒
        [self emit: lLitI literal: ref]
    self illToken: word]

selector: word
    [self emit: send literal: lit]

illToken: token
    [user notify: 'input not understood ' + token]
```

get rid of actOnWord and wordAsObj
make their clients use appropriate other messages

be sure nextWord is only used when a variable or selector is expected

classInit should initialize the dictionaries
    locals (including self, temps, fields) as byte codes
    classVars -- right from the class

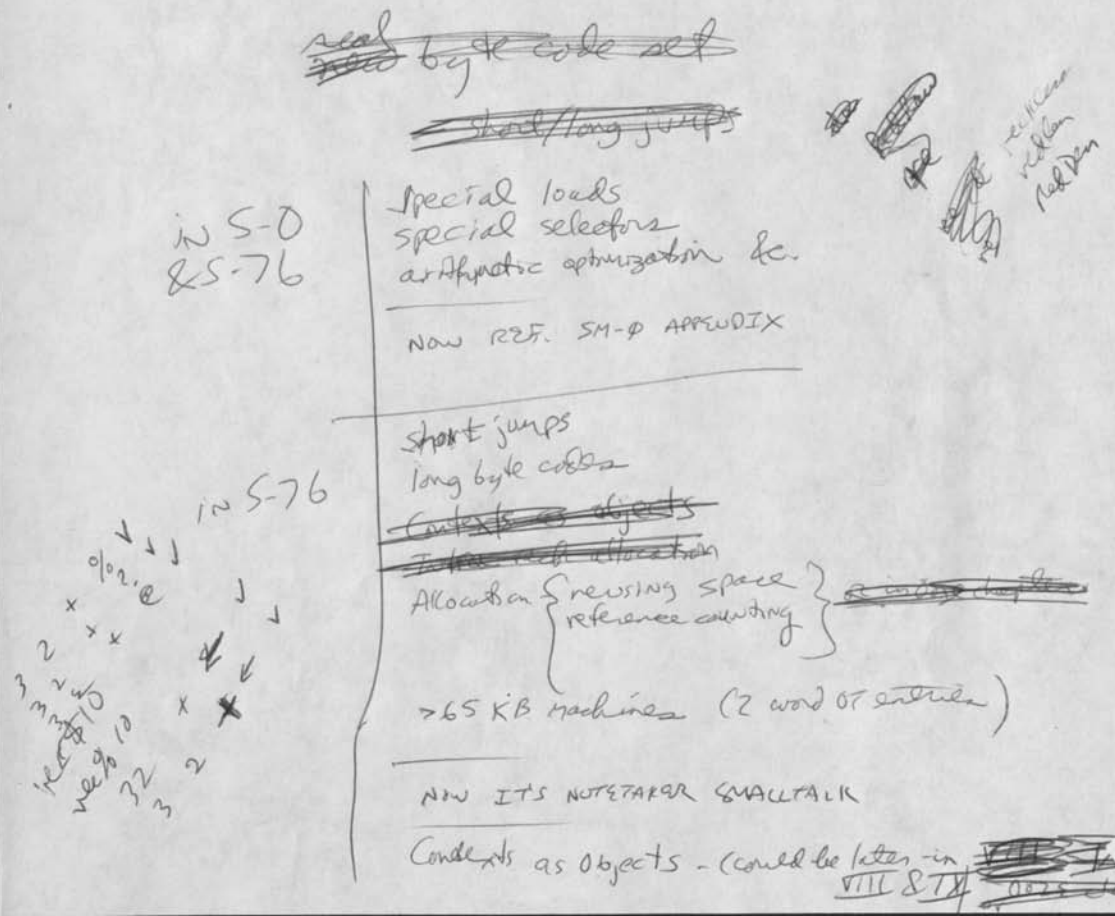
Smalltalk should have true false nil

do
    [blockStack last do: self]

ForLoop's do: compiler
    should use compiler selector: 'asStream!' etc.

Optimizations of Smalltalk-76 Implementations

We could use an outline of this chapter to see if there really is a chapter worth of material



```
global thisDialog


Class Dialog
    fields level topContext seenContext ancestor

default
    [topContext ← Context default.
     self converse: 1]

converse: level
    [self see: topContext.
     while: true do:
        [thisDialog ← self.
        self prompt.
        (self obeyFrom: terminal nextCommand asStream) printon: terminal]]

obeyFrom: q | w method errorText className fieldNames
    [w ← q nextToken.
     w='ε'⊃
        [method ← Method new patternFrom: q.
         errorText ← method behaviorFrom: q⊃ [↑errorText]
         method class insertMethod: m. ↑m]
     w='ε ε'⊃
        [className ← q nextToken.
         fieldNames ← q restOfTokens.
         ↑Class new title: className fields: fieldNames]
     method ← seenContext method copy behaviorFrom: q reset.
     ↑seenContext evaluate: method]


prompt | i
    [terminal newLine.
     for: i to: level do: [terminal append: '!'].
     terminal space]


see: context
    [context⊃ [↑seenContext ← context]
     ↑'no more contexts']
```

for debugger only...

```
in: topContext
    [ancestor ← thisDialog.
     self converse: ancestor level+1]

callerOf: context | earlier
    [earlier ← context caller.
     earlier≡ancestor topContext⊃ [↑false]
     ↑earlier]

calleeOf: context | earlier later
    [earlier ← topContext. later ← false.
    until↓ earlier≡context do↓
        [later ← earlier.
         earlier ← later caller].
    ↑later]

topContext
    [↑topContext]

level
    [↑level]

M
    [↑seenContext method asText: seenContext pc]

I
    [↑seenContext receiver class asText: seenContext receiver]

E
    [↑self see: (self callerOf: seenContext)]

L
    [↑self see: (self calleeOf: seenContext)]

Q
    [ancestor⊃ [↑thisContext caller ← topContext]
     ↑'no outer dialogs']
```

```
Class Terminal

fields canErase

nextCommand | s c
    [s ← Stream default.
     until: (c←self next)=doitChar do:
       [c=bsChar⇒
          [s empty⇒ [self erasedAll]
           self eraseChar: s pop.
           self erasedChar]
        c=bwChar⇒
          [s empty⇒ [self erasedAll]
           until: [s empty or: s last isDelim≡false] do: [self eraseChar: s pop].
           until: [s empty or: s last isDelim] do: [self eraseChar: s pop].
           self erasedWord]
        c=delChar⇒
          [until: s empty do: [self eraseChar: s pop].
           self erasedAll]
        c=retypeChar⇒
          [self echoAll: s contents]
        s next ← c. self next ← c].
     s next ← c. terminal newLine.
     ↑s contents]

eraseChar: c
    [canErase⇒ [self reallyEraseChar: c]]

erasedChar
    [canErase⇒ [] self append: '\']

erasedWord
    [canErase⇒ [] self append: '←']

erasedAll
    [canErase⇒ [] self append: ' XXX'; newLine]

echoAll: s
    [canErase⇒ [] self newLine; append: s]

next
    [] primitive: n

next ← c
    [] primitive: n

reallyEraseChar: c
    [] primitive: n
```

Class Method

```
patternFrom: q | s t
    [s ← Vector new asStream.
     t ← q nextToken.
     until: (t='[' or: t='|') do: [s push: t. t ← q nextToken].
     selector ← s pop.
     class ← Smalltalk lookup: s pop unique.
     nArgs ← s position. arguments ← s contents.
     s reset.
     until: t='[' do: [s push: t. t ← q nextToken].
     nTemps ← s position. temporaries ← s contents]

behaviorFrom: stream
    [Compiler new behaviorFrom: stream into: self]
```

for debugging only...

```
asText: pc
    ["decompile this method, marking pc location"
     "return a string"]
```

Class Class

```
title: title fields: fields
    ["to be written"]

insertMethod: method
    ["to be written"]
```

for debugging only...

```
asText: instance
    [title printon: terminal.
    terminal space.
    fields printon: terminal]
```

Class VariableLengthClass

for debugging only...

```
asText: instance
    [title printon: terminal.
    terminal space.
    instance length printon: terminal]
```

Class Context

fields sp bp

```
default
    [sp ← bp ← someplaceWithADefaultMethod]

evaluate: method
    ["copy to top of stack,
        jam in method,
        run it,
        copy temps & args back down"] primitive: n
```

for debugging only...

```
bp
    [↑bp]

sp
    [↑sp]

receiver
    [↑systemStack word: (bp+selfOffset)]

caller ← context
    [systemStack word: (bp+oldBpOffset) ← context bp.
     systemStack word: (bp+oldSpOffset) ← context sp.
     ↑context]

caller
    [↑Context new bp: (systemStack word: (bp+oldBpOffset)) sp: (systemStack
word: (bp+oldSpOffset))]

notUnderstood
    ['Message not understood' printon: terminal.
     Dialog new in: self]
```

Class Stream

```
nextToken | s
    [s ← Stream default.
     until: (self empty or: self peek isDelim=false) do: [self next].
     self empty⇒ [↑false]
     until: (self empty or: self peek isDelim) do: [s next ← self next].
     ↑s contents]

restOfTokens  | s t
    [s ← Vector new asStream.
     while: (t ← self nextToken) do: [s next ← t].
     ↑s contents]
```

Class Compiler

```
behaviorFrom: stream into: method
    ["compile source method into byteString and literals, possibly increasing
nTemps"
     "return an error string, or false if OK"]
```

Class SystemStack

```
word: w
    [] primitive: n

word: w ← x
    [] primitive: n
```

Tiny Smalltalk Byte Codes

0	0	LOAD/STORE	TEMP
20	1	"	ARG
40	2	"	FIELD
60	3	"	LITERAL INDIRECT
100	4		
120	5	LOAD	LITERAL
140	6		
160	7	SEND	LITERAL
200	8		
220	9	SEND	SPECIAL
240	10		
260	11	LOAD SPECIAL	
300	12	JUMP	FORWARD
320	13	JUMP	BACKWARD
340	14	BRANCH . FALSE + POP.	
360	15		SUNDRY

no SPECIAL in 1st version & no superclasses

Later versions

(1) Bare.

(1½) compact storage

(2) superclasses

(3) perform

(4) specials

TINY S.T.

(5) new byte code set
extended
encoded jumps
short jumps
thisContext
:

later

grow last array in place
(become)
decompiler
debugger
Floating point

Inter-Office Memorandum

To	Adele, Dave, Kim	Date	April 30, 1979
From	Larry Tesler	Location	Palo Alto
Subject	Polish Tool	Organization	PARC/SSL

XEROX

Filed on: <Tesler>PolishTool.press, .memo

Here is yet another plan for Tool. We do have an interpreter for a symbolic language, but instead of LISP-like S-notation (prefix Polish), we use Forth-like postfix Polish. The interpreted code is a list of symbols with no sublists. Compare:

Smalltalk

```
x ← 25⊙10 rect: x1⊙y1. user show: x asString.
```

S-notation

```
[[← x (rect: (⊙ 25 10) (⊙ x1 y1))] (show: user (asString x))]
```

Postfix Polish

```
y1 x1 ⊙ 10 25 ⊙ rect: ← x . x asString! user show: .
```

Both S-notation and Postfix Polish yield a usable system without a compiler.

S-notation is somewhat more readable and is more amenable to symbolic manipulation if one wants to have programs that write programs ala LISP.

Polish postfix has a couple of advantages for the book. The read routine is easier than that for S-notation. The interpreter is easier, and is much closer in its structure to the byte code interpreter, because byte codes are also postfix Polish.

We would introduce the postfix Polish notation and write its interpreter in Smalltalk. [I think there is no need to write the same interpreter in machine language ala Forth; it requires more curriculum material and ends up with an inherently slow and uncompact system.] Then we would introduce byte coding; present a Polish-symbolic to byte-code translator in Smalltalk; present the interpreter in Smalltalk; present the same interpreter in (Kim's) machine code.

To obtain a complete runnable system for the 8086, we could then either (1) present the Polish-symbolic to byte-code translator in machine code or, better, (2) print a precompiled hexadecimal byte code version of the Smalltalk translator, which would then be interpreted by the interpreter to read other Smalltalk programs. Choice (2) would be slower but would require less machine code, code that we won't want later in the book when the real compiler is introduced.

Inter-Office Memorandum

To	Smalltalk Interest	Date	January 22, 1979
From	Larry Tesler	Location	Palo Alto
Subject	Stalk-notation	Organization	PARC/SSL

XEROX

Filed on: <Tesler>stalk.press,.memo

One of the shortcomings of Smalltalk is the absence of a LISP-like S-expression notation. It is true that in every version of Smalltalk there has been a *read* message that turns parenthesized program text into a nested structure. However, that structuring does not qualify as a true S-notation, because the levels of the nested representation do not correspond to the semantic or syntactic units of the language. The lack of correspondence is due to the fact that style, syntax, and precedence rules discourage the use of parentheses to delimit units.

The current compiler for Smalltalk-76 creates a parse tree according to syntactic units, with selector names and byte codes at the leaves. The tree can print itself in a fully parenthesized form, but that form was designed for debugging purposes and is not convenient, consistent, or comprehensible. Furthermore, there is no facility for converting that form to a parse tree, and such a facility would be of little use given the presence of byte codes at the leaves.

Peter Deutsch has long maintained that Masterscope-like facilities would most easily be implemented in Smalltalk if there were an S-expression notation. Additionally, all the well-known advantages that LISP gains from S-notation would be available. Generating, transforming, and interpreting programs would be easier. The output of symbolic mathematics manipulators could be executed. Alternate user languages could be provided that all compile into the same intermediate notation. More precise and concise descriptions of Smalltalk semantics could be written. Ideas for language extensions could be proposed more formally and analyzed more rigorously.

There are several possible starting points for specifying an S-notation for Smalltalk. I will use a variant of the aforementioned compiler parse tree, bent more towards traditional LISP. I will call it S-talk, or simply *stalk*-notation.

The LISP S-notation has a simple structure. An S-expression is either an atom or a parenthesized list of S-expressions. When an S-expression represents a program, the atoms are string constants, numeric constants, and variable names. The lists are of the form:

(f a1 ... aN)

where f evaluates to a function name and a1...aN evaluate to the arguments of the function.

For Smalltalk, it would seem practical to place the recipient of the message at the front of the list, thus:

(r m a1 ... aN)

where r evaluates to the recipient object, m evaluates to the message selector, and a1...aN evaluate to the arguments. Examples:

(2 + 3)

(user sched)

(a max: b)

(2 to:by: 10 1)

((2 + 3) to:by: ((user sched) length) (a max: b))

So far, stalk-notation does not account for non-numeric literal data, assignment, or sequencing constructs like conditionals, blocks, loops, and cascading.

LISP's solution to these cases is to reserve certain names to have special meaning in the first position, e.g., QUOTE for literals, COND for conditionals, PROG for blocks, SETQ for assignment. The special names imply nonstandard evaluation of the arguments: none for QUOTE, selective for COND, and sequential for PROG.

An alternative to reserving special names in the first position is to change the parentheses to special brackets. However, such a convention could quickly use up all available brackets and make bracket characters unavailable as selector names. I propose to reserve only ()[]{}' as special brackets, as in the current user syntax:

(r m a1...aN)
sends the message m to r with arguments a1...aN

[s1 ... sN]
executes s1 through sN in order, yielding sN as a value

[? e1 s1 ... eN sN s]
evaluates e1...eN until a non-false is found, then evaluates the next s; if all are false, evaluates the last s

[← v e]
assigns the value of e to the variable v

[↑ e]
returns e from the current method

[* e s]
evaluates e and s repeatedly until e is false, then stops before the next evaluation of s

{c f1:v1 ... fN:vN}
represents an object of class c with components named f1...fN having values v1...vN respectively.

{c v1 ... vN}
represents an object of class c with components implicitly numbered 1...N having values v1...vN respectively.

'a...z'
is an abbreviated form for a string

Remote evaluation is determined by open colons in the selector name. Cascading in the user syntax would be expanded in S-notation into a block of messages sent to the same recipient; if the recipient is computed, it must be assigned to a temporary variable. Comments would be "quoted".

It is possible to reduce all '...', (...), and [...] forms to the {...} literal form.

```
'abc' => {String 65 66 67}

(3 + 4) => {Message recipient: 3 selector: '+' arguments: {Vector 4}}

[s1 ... sN] => {Block statements: {Vector s1 ... sN}}

[? e1 s1 ... eN sN s] =>
        {Conditional
                conditions: {Vector e1...eN}
                consequences: {Vector s1...sN}
                alternative: s}

[← v e] => {Assignment variable: 'v' value: e}

[↑ e] => {Return value: e}

[* e s] => {Loop condition: e action: s}
```

Note that Message, Block, Conditional, Assignment, Return, and Loop must all be Smalltalk classes. Each of them as well as certain other classes would be able to print, read, compile, and interpret its part of the notation.

A method definition would be simply a literal of class Method:

```
{Method
    class: Number
    selector: 'max:'
    arguments: {Vector 'x'}
    program: [↑ [? (self > x) self x]]
}
```

Note that the selector is a single string, while the argument list is a vector of strings. (I use the word 'Vector' here for historical reasons; I think it indescriptive of an ordered set.)

Stalk-notation should map directly to and from any two-dimensional notation that we adopt, and can be compiled very rapidly into byte codes. It is as easy to produce from current user syntax as is the current compiler parse tree; however, in-place error message capability would have to be sacrificed unless we had coroutines or unless we installed backpointers in the S-expression to the source code. (Backpointers could be in a field of the major classes like Message and Block, but could be suppressed from the minor classes like String and from all printed forms.)

I welcome comments and further development of these notions.

Inter-Office Memorandum

To	Adele, Dave, Kim	Date	April 26, 1979
From	Larry Tesler	Location	Palo Alto
Subject	Classes for Tool	Organization	PARC/SSL

XEROX

Filed on: <Tesler>ToolClasses.press, .memo

Here is a possible plan for "Tool", mainly, the class definitions for late in chapter II and early in chapter III-A.

Classes Introduced in Chapter II

Class new title: 'True' fields: ''
There is one instance: (true).

Class new title: 'False' fields: ''
There is one instance: (false).

Class new title: 'Nil' fields: ''
There is one instance: (nil).

Class new title: 'List' fields: 'first rest'
The elements of a list (x) are (x first), (x rest first), (x rest rest first), etc., until (x rest) is (nil). Two different instances may be equal in value.

Class new title: 'Number' fields: 'bitList'
A number is a list of bits, low order first, sign last: true represents 1, false represents 0. (Two different instances may be equal in value.)

Class new title: 'Character' fields: 'ascii'
There are 128 instances of this class, each with a different ascii value between 0 and 127.

Class new title: 'String' fields: 'charList'
A string is a list of characters. (Two different instances may be equal in value.)

Class new title: 'Stack' fields: 'itemList'
A stack is a LIFO list of items.

All the above classes should have their methods defined, and some example LISP-like programs should be shown. No loops allowed, just recursion.

Classes Introduced in Chapter III

Class new title: 'Selector' fields: 'nameString'
Every instance of this class has an unequal nameString.

Class new title: 'Instruction' fields: 'kind which'
See next section for field meanings.

Class new title: 'Method' fields: 'instructionList literalList numTemps'
A method is a list of instructions and a list of literals. Its execution requires numTemps temporary variables initialized to nil.

Class new title: 'MessageBinding' fields: 'selector method'
A message binding associates a selector with a method.

Class new title: 'MessageDictionary' fields: 'bindingList'
A message dictionary is a list of message bindings, each with a different selector.

Class new title: 'Context' fields: 'receiver argumentList temporaryList evaluationStack method pc'
A context has a receiver (self), a list of argument variable values, a list of temporary variable values, an evaluation stack, a method, and a pc (program counter).

Class new title: 'Process' fields: 'contextList'
A (the) process is a LIFO list of contexts.

Class new title: 'Instance' fields: 'messageDictionary fieldList'
An instance has a pointer to the message dictionary of its class and has a list of its own fields. (This is somewhat circular, but we may need it to write the interpreter.)

Instruction kinds for the Interpreter

Pop
Pops the top item from the evaluation stack.

Load Self
Pushes (self) onto the evaluation stack.

Load Argument
Pushes the context's which'th argument onto the evaluation stack.

Store Argument
Replaces the context's which'th argument by the top of the evaluation stack.

Load Temporary
Pushes the context's which'th temporary onto the evaluation stack.

Store Temporary
Replaces the context's which'th temporary by the top of the evaluation stack.

Load Field
Pushes the receiver's which'th field onto the evaluation stack.

Store Field
Replaces the receiver's which'th field by the top of the evaluation stack.

Load Literal
Pushes the value of the method's which'th literal onto the evaluation stack.

Load Indirect Literal
Pushes the value of the global variable referenced by the method's which'th literal onto the evaluation stack.

Store Indirect Literal
Replaces the value of the global variable referenced by the method's which'th literal by the top of the evaluation stack.

Send Message
Sends a message to the object on top of the stack. The arguments are beneath the top of the stack. The selector is the method's which'th literal.

The Interpreter

We assume that a compiler exists that translates Smalltalk source code to object methods according to the above schema.

Then, we write an interpreter in Smalltalk using that schema. No loops, just recursion.

Next, we decide to store many of the supposed linked lists contiguously: numbers (16 bit limit), strings, methods (byte coded), dictionaries (two parallel vectors), contexts. The context lists are concatenated into a single stack. No interpreter can be written in Smalltalk at this point, because we haven't introduced variable length classes. But Kim's nice two page 8086 version can be written, and we do so after introducing Store Mode and the 8086 instruction set.

Chapter 5 will close the loop by introducing variable length classes.

Inter-Office Memorandum

To	Adele, Dave, Kim	Date	April 26, 1979
From	Larry Tesler	Location	Palo Alto
Subject	An Interpretive Tool	Organization	PARC/SSL

XEROX

Filed on: < Tesler > InterpretiveTool.press, .memo

Here is an alternative plan for "Tool". The class definitions for chapter II are the same, but those for chapter III are different. Mainly, no compiler is assumed: an S-expression interpreter is used instead. This version is even closer to Smalltalk-72 than the last one, but the source code is fully parenthesized as in LISP. We assume that a simple parser exists that translates Smalltalk source code to object methods according to the above schema. (Or we could even write that parser!) Then, we write an interpreter in Smalltalk using the above schema. No loops, just recursion. Next, we assume a compiler and move on to the plan of the previous memo, *"Classes for Tool"* .

Classes Introduced in Chapter III

Class new title: 'Name' fields: 'nameString'
Every instance of this class has an unequal nameString.

Class new title: 'Variable' fields: 'name value'
This class is introduced to enable the interpreter to resolve variable bindings at run time

Class new title: 'Selector' fields: 'name'

Class new title: 'Literal' fields: 'value'

Class new title: 'Message' fields: 'receiver selector argumentList'
The receiver is a (variable) name or a message or a literal.

Class new title: 'Method' fields: 'numTemps messageList'

Class new title: 'MessageBinding' fields: 'selector method'

Class new title: 'MessageDictionary' fields: 'bindingList'

Class new title: 'Context' fields: 'receiver argumentList temporaryList evaluationStack pc'
A context has a receiver (self), a list of argument variables, a list of temporary variables, an evaluation stack, and a pc (program counter: a direct pointer into the method structure).

Class new title: 'Process' fields: 'contextList'

Class new title: 'Instance' fields: 'messageDictionary fieldList'

SIM

Simula 67 and Smalltalk

Simula 67

Simula 67 [refs] is a popular programming language for simulation and other applications. The language is an extension of Algol 60. The most significant extension is the *class declaration.*

A Simula 67 class declaration looks very much like a procedure declaration but when it is invoked an *object* is allocated. The value returned by the invocation is a *reference* to the new object. The body of a class declaration declares procedures and variables that become part of every object created by the class. An object may be thought of as a packet of data (variables) with associated programs (procedures).

A Simula 67 class may be declared as a variant, or *subclass*, of a previously declared class. The subclass inherits the procedures and variables of its *superclass.* The variables and procedures in each object of the subclass are the concatenation of those declared in the superclass and subclass declaration.

Smalltalk

Smalltalk borrows the class concept from Simula 67. However, unlike Simula 67, Smalltalk is not built on an Algol 60 base. The class concept of Simula has been enriched and generalized enough in Smalltalk to serve as a complete language semantics. The syntax of Smalltalk was designed from scratch.

The way Simula semantics was collapsed was to implement many primitive concepts of Simula by class declarations in Smalltalk. The Algol data types *real* and *integer* as well as the data structuring concepts *array* and *string* are implemented by class declarations in Smalltalk. Even Simula's novel concepts -- *class* and *object* -- are implemented by class declarations in Smalltalk (this may seem circular, but it works out very well).

The syntax of Smalltalk is not based on any other language. It was designed to support the object-oriented semantics while allowing somewhat more concise programs than in most languages (but not as much conciseness as in APL). As a result, the syntax appears strange to most experienced programmers when they first learn Smalltalk.

Terminology of Smalltalk

The variables named in a class declaration that appear in every instance of the class are called *fields* to emphasize the internal structural similarity between an object in Smalltalk and a *record* in a business-oriented language. Asking an object to call one of its procedures is known as *sending a message to the object* to emphasize the external behavioral similarity between an object in Smalltalk and an autonomous process in a multi-process operating system. The procedures declared in a class body are called *methods* because they define how the object will respond to various messages.

Specific Omissions in Smalltalk

In Smalltalk, the state of an object can only be accessed directly by procedures of that object. Other objects are required to send messages requesting access to the state. All interfaces between objects are procedural. It is easy to change the representation of an object while maintaining the same external interface because it is certain that no external object depended on the particular representation. (Of course, a change in representation may lead to a change in performance. Improved performance is a common motivation for wanting to change a representation.)

There are no type declarations in Smalltalk. The value of every variable is a reference to an object. The syntax provides no way to constrain the class of the referenced object. Although it would be possible and in fact helpful to allow some sort of type declaration capability, it is far from essential. The reason is that Smalltalk programs are developed interactively and incrementally and thus errors due to type mismatch during assignment to variables and arguments are generally caught and fixed just a few minutes after a method is compiled.

Simula allows a class declaration to have a *main part.* The main part is a set of unnamed statements after the procedure and variable declarations. The main part of an object of the class can be started and stopped during the lifetime of the object. This facility permits a degree of

concurrent execution sufficient for implementing discrete simulations. In Smalltalk, class declarations do not have a main part. A similar effect can be obtained by declaring methods called *start* and *stop* that in turn invoke process-manipulation methods.

Simula allows an object executing the main part of a superclass declaration to execute the main part of the subclass declaration by use of the *inner* construct. Smalltalk has no main parts, but it allows an object executing a method defined in a superclass to invoke a method of another name defined in the subclass by sending a message to the pseudo-variable `self`. It also allows an object executing a method defined in a subclass to invoke a method of the same or another name in the superclass by sending a message to the pseudo-variable `super`

Simula allows a name to be declared *virtual* in a class declaration as long as an actual declaration is supplied in each subclass. Since Smalltalk has no type declarations, there is no need for a *virtual* construct. All names are intrinsically virtual. This means that all procedures are generic; i.e., when a message is passed to an object, there is a run-time determination of which of possibly several methods of that name will be invoked, depending on the class of the receiver of the message.

In Simula, a class is a language construct whose uses and methods are fixed by the manual and the compiler. Smalltalk has a class named Class that describes the behavior of all classes, and that can be augmented even at execution time. The Simula construct for creating an object of class Q is NEW Q and requires a reserved word in the language, namely, NEW. The Smalltalk construct for creating an object of class Q is Q `new`, and simply sends the message `new` to Q, invoking a (primitive) method of class Class.

The NEW construct of Simula allows for arguments to be passed to the class to accomplish initialization of the new object. New Smalltalk objects have all their fields initialized to `nil`. Subsequent initialization is accomplished by sending messages as usual.

In Simula, an object, like a class, is a language construct whose uses and methods are fixed by the manual and the compiler. Smalltalk has a class named Object that is the ultimate superclass of all other classes, and which thus describes behavior common to all objects (except those having override procedures). The hierarchy of subclasses in Smalltalk has a single root, class Object.

In Simula, a stack, like a class or an object, is a language construct whose uses and methods are fixed by the manual and the compiler. Smalltalk has classes named Context and Process that allows access to the internal representation of the execution stacks and thus makes the programming of debugging tools straightforward. By manipulation of contexts, a wide variety of multi-process mechanisms can be implemented. Thus, Smalltalk has no concurrency constructs built into the language itself.

Syntactic Differences

Procedure call.

x.p

```
x p
```

x.k(a)

```
x k: a
```

x.k1k2(a1, a2)

```
x k1: a1 k2: a2
```

x.sub(y)

```
xoy
```

x.assignp(v)

```
x p ← v
```

x.assignk(a, v)

```
x k: a ← v
```

x.assignk1k2(a1, a2, v)

```
x k1: a1 k2: a2 ← v
```

x.assignsub(y, v)

```
xoy ← v
```

Similarities

Smalltalk was largely inspired by Simula. Both have objects, classes, and subclasses.

Terminology Differences

Procedures are called methods.

Calling a procedure is called sending a message.

??? are called fields.

Semantic Differences

No access to state. All interfaces are procedural.

No type declarations. Type errors are caught at run-time.

No main part or START keyword. Use a method instead.

No INNER. Use self and super instead.

No VIRTUAL. All procedures are generic.

Class is a class.

No NEW keyword and no ??? (class decl args). Use a method instead.

Object is a class.

The subclass hierarchy has a single root (class Object).

Context is a class.

No concurrency constructs. Manipulate contexts instead.

Integer, Array, String, etc. are classes.

Syntactic Differences

Procedure call.

x.p

```
x p
```

x.k(a)

```
x k: a
```

x.k1k2(a1, a2)

```
x k1: a1 k2: a2
```

x.sub(y)

```
x○y
```

x.assignp(v)

```
x p ← v
```

x.assignk(a, v)

```
x k: a ← v
```

x.assignk1k2(a1, a2, v)

```
x k1: a1 k2: a2 ← v
```

x.assignsub(y, v)

```
x○y ← v
```

Block.

begin *s1*; *s2*; ... *sn* **end**

```
[ s1. s2. ... sn ]
```

Conditional.

if a **then** b **else** c

```
[a ⇒ [b] c]
```

Iteration.

for i ← a **step** b **until** c **do begin** ... **end**

```
fors i from: a to: c by: b dos [ ... ]
```

Method declaration.

```
real procedure p; begin real v, w; ... return(z); end;
```

```
p | v w [ ... ↑z]
```

Class declaration.

```
D class C;
    begin
    real f1, f2;
    real procedure p; begin real v, w; ... end;
    real procedure q; begin ... end;
    ...
    end;
```

```
Class new title: 'C' subclassof: D fields: 'f1 f2' declare: ''; asfollows
    p | v w [ ... ]
    q [ ... ]
```