Smalltalk (ST) is a pure object oriented programming language. In ST, all values are considered objects. The interpreter that we write is for is a small approximate subset of ST (See the grammar for it below).

Unlike mini lisp which had only one value (integer), our ST has three type of values (objects). They are int, sym, and object instance.

So our ST value would be packaged in the following way:

```c
struct STVALUEREC
{
    CLASS owner; //every value is owned by some one
    STVALUETYPE vtype; //0 for INT, 2 for SYM, 3 for USER
    int intval; //if its INT
    NAME symval; //if its SYM
    ENV userval; //if its USER
};
```

Hence the EXPREC has to be modified:

```c
struct EXPREC
{
    EXPTYPE etype; //what type of expression
    STVALUE valu;
    NAME varble;
    NAME optr;
    EXPLIST args;
};
```

**PART ONE DUE Nov 02**

Do the following:

a) Modify the symbol table
b) Create a record to store small talk values.
c) Create a linked list to store class records (come to class for details)

d) Write a function ParseClass() that will parse a class, create a class record and link it in the linked list of class records.

PART TWO DUE NOV 16

Finish the interpreter by modifying eval and a few other functions (not much), please pay attention in class.

The Mini Smalltalk Interpreter

The interpreter is interactive. The user enters three kinds of inputs

Function definitions such as

\[ \text{(define double (x) (+ x x))} \]

expressions, such as:

\[ \text{(double 10)} \]

and class definitions such as:

\[ \text{(class FinancialHistory Object} \]
\[ \text{ (cashOnHand incomes expenditures) } \]
\[ \text{(define initFinancialHistory (amount) } \]
\[ \text{ (begin} \]
\[ \text{ (set cashOnHand amount) } \]
\[ \text{ (set incomes (mkDictionary)) } \]
\[ \text{ (set expenditures (mkDictionary)) } \]
\[ \text{ self))} \]
\[ \text{(define receive:from: (amount source) } \]
\[ \text{ (begin} \]
\[ \text{ (at:put: incomes source (+ (totalReceivedFrom: self source) amount))} \]
\[ \text{ (set cashOnHand (+ cashOnHand amount)))} \]
\[ \text{(define spend:for: (amount reason) } \]
\[ \text{ (begin} \]
\[ \text{ (at:put: expenditures reason (+ (totalSpentFor: self reason) amount)))} \]
(set cashOnHand (- cashOnHand amount)))
(define cashOnHand () cashOnHand)
(define totalReceivedFrom: (source)
  (if (includesKey: incomes source)
      (at: incomes source)
      0))
(define totalSpentFor: (reason)
  (if (includesKey: expenditures reason)
      (at: expenditures reason)
      0))
)

Function and class definitions are simply remembered by the interpreter, and expressions are evaluated. Evaluating an expression is the same as running program in most other languages.

Note that a class definitions has methods, these belong to the class that defines the method. However, the syntax for the method is exactly the same as for a function. Hence, we can make use of our previous “ParseDef” to parse a method.

**Syntax**

```
input --> expression | fundef | classdef

classdef --> (class class class inst-vars methoddef+)

inst-vars --> (variable*)

methoddef --> fundef

fundef --> (define function arglist expression) ;same as lisp

arglist --> (variable*)

expression --> value | variable
  | (if expression expression expression )
  | (while expression)
  | (set variable expression) ;same as lisp
  | (begin expression+ )
  | (optr expression*)

optr --> function | value-op

value --> integer | #symbol

value-op --> new | + | - | * | / | | = | < | > | print ;same as lisp

function --> name

variable --> name
```
class --> name

integer--> sequence of digits, possibly preceded by a minus sign

name --> any sequence of characters not an integer, and not containing (, ), ;, or space

A function cannot be one of the keywords define, if, while, begin or set or any of the value-ops.

Comments are introduced by the character ';' and continue to the end of the line.

A session is terminated by entering quit.

Semantics

symbols: Can be either integers, symbols, or class instances (objects). The symbol #nil is used to represent “bad” or “undefined” values.

self: The definition of a method always has one implicit argument, which can be referred to by the name self.

new: This function creates a new object of class C when applied to the variable called C, initializing its instance variables to the integers zero.

Redefinition of value-op’s: A small extension not previously noted is that value-ops are allowed to be used as messages.

Inheritance: The instance variables representing an object of class C are those named in C or in any of its ancestors. Object of a class C can respond to messages defined in as ancestor of C.

Method search: when a message m is sent to an object of class C, the interpreter first looks in C for a method m, if it fails, it looks in the supper class and so on.

The class Object: We start with one class called Object, which has instance variable self and as its objects all the integers and symbols. All other classes inherit the instance variable self, which is initialized whenever a new object is created.

Scope rules: For the variables is static and for functions it is dynamic.
**Functions and messages:** There are two types of functions calls, global and message sends. First check to make sure its a mesage if not it must be a global function else error.

**Working Example:**

```
(define mkFinancialHistory (amount)
    (initFinancialHistory (new FinancialHistory) amount))

(set myaccount (mkFinancialHistory 1000))
(spend:for: myaccount 50 #insurance)
(receive:from: myaccount 200 #salary)
(cashOnHand myaccount)
1150
(spend:for: myaccount 100 #books)
(cashOnHand myaccount)
1050
(set myaccount (mkDeductibleHistory 1000))
(spend:for: myaccount 50 #insurance)
(receive:from: myaccount 200 #salary)
(cashOnHand myaccount)
1150
(totalDeductions myaccount)
0
(spend:Deduct: myaccount 100 #mortgage)
(cashOnHand myaccount)
1050
(totalDeductions myaccount)
100
(spend:for:deduct: myaccount 100 #3-martini-lunch 50)
(cashOnHand myaccount)
950
(totalDeductions myaccount)
150
```