Homework 2: Object-Oriented Programming in Smalltalk

Due: Wednesday, Feb. 9.

This is a group homework.

For all Smalltalk programs, you must run your code with little Smalltalk. You must also provide evidence that your program is correct (for example, test cases). Hand in a printout of your code (print the file produced by `fileOut`) and your testing. (You can run Smalltalk from within an emacs shell to get test output captured, or you can use the Smalltalk printing primitives to print test output to a file.)

Don't use designs or write code that look like C! We'll take points off for poor designs or style.

Choose one (1) problem of the following (only) to do for this homework. You should choose one that you and your group are interested in; see the track information on each problem.

1. (applications—general) Design your own homework problem to be coded in Smalltalk; the idea is that it would be useful for your application interest). You are not to do some program that is already published, although it is okay to extend something you find published (and this is a good idea if you are a bit unsure about Smalltalk and OOP, see also the last problem in this homework); but you must tell us if you are doing that.

   You must discuss a written description of your problem and an initial design (describe the classes) with a staff member by February 4. You may change the problem after this meeting, but you should get our approval for any major changes in the problem. Also you are encouraged to discuss your thoughts about your problem and its design with us before then.

2. (applications—artificial intelligence, semantics) Write a Smalltalk program to learn a context-free grammar from examples. Your program can use both positive and negative information. You can narrow down the problem by making additional assumptions, but let us know what they are. See Section D5e of the *Handbook of Artificial Intelligence* for algorithms and references.

3. (semantics) Design a static type system for Smalltalk. That is, propose additions to Smalltalk syntax that would allow type checking and describe how to do type checking for Smalltalk programs. Be sure to include examples of what can and can't be type-checked with your ideas, and state the limitations of your type system clearly. (That is, it doesn't have to be perfect, try to do 95% of the programs well and have a clean type system).

4. (semantics, programming) Compare the semantics of Smalltalk and C++ for: object creation, classes, inheritance, and message passing. Describe a translation that can take a Smalltalk program and produce an "equivalent" C++ program. Be sure to include examples of what can and can't be translated with your ideas, and state the limitations of your translation clearly. (That is, it doesn't have to be perfect, try to do 95% of Smalltalk programs well and have an elegant translation). One strategy for describing the translation is to describe it by means of the grammar for Smalltalk (that is, describe how to translate instances of each production).

5. (semantics, programming) Write an interpreter for the language mini-Cecil, whose grammar is given in Figure 1. In this interpreter you can ignore type checking. You don't have to do parsing either; your interpreter's input will be an abstract syntax tree. See the staff for the meaning of the various constructs in this language, and for hints and some code to get you started (you'll "fill in the blanks").

6. (programming, applications—game theory) Object-oriented programming techniques are touted as being good for simulation. In this problem you will investigate an iterated prisoner's dilemma, using simulation techniques and Smalltalk.

Abstract Syntax:

$M \in$ Main-Command
$T \in$ Type-Declaration
$T^* \in$ Type-Declaration-List
$C \in$ Class-Name
$N \in$ Name
$P \in$ Program
$D \in$ Declaration
$D^* \in$ Declaration-List
$C \in$ Command
$E^* \in$ Expression-List
$E \in$ Expression
$V \in$ Variable
$i \in$ Index
$g \in$ Operation
$T \in$ Type-Symbol
$F \in$ Formal-Arg-Declaration
$F^* \in$ Formal-Arg-Declaration-List
$Cl \in$ Class
$Op^* \in$ Operation-List
$Val \in$ Value-Expression
$Num \in$ Number
$Bool \in$ Boolean

P ::= T* Op* M
Cl ::= `Class` N `implements` T
`instance variables` D*
end;
T ::= type N
Op* ::= | N (F*):T C E; Op*
F ::= N@C:T
M ::= `program` (D^*_1) `returns` D^*_2 `;` C
D* ::= | D`,` D*
D ::= V `:` T
C ::=E | V `:=` E | V.i `:=` E | `begin D*;` C `end` | C_1; C_2 | `if` E_1 `then` C_1 `else` C_2 `fi`
E ::= V | g`(`E*`)` | $\perp$ | `new` N | V.i | Val
E* ::= | E* E
Val ::= Num | Bool
Num ::= 0 | 1 | $\cdots$
Bool ::= true | false

Figure 1: Syntax of mini-Cecil.

The prisoner's dilemma is a model of certain kinds of social interactions. The idea is that people are often faced with situations where they may either "cooperate" or "defect". As the standard example, consider two criminals, Bonnie and Clyde, who have been arrested on a minor charge but are suspected of committing a major crime. The police make the following offer to each: "if you confess and implicate your partner, then you will go free and your partner will get 5 years in jail, but if you do not confess, you will have to serve one year in jail on the minor charge. Oh by the way, if you both confess, you will each serve 3 years in jail." In this case cooperation (among the prisoners) means keeping silent, defection means implicating the partner. The police offer can be represented by the following payoff matrix:

Bonnie\Clyde	cooperate	defect
cooperate	(-1,-1)	(-5,0)
defect	(0,-5)	(-3,-3)

where the entry (0,-5) means that if Bonnie defects and Clyde cooperates, then Bonnie gets a jail term of 0 years, while Clyde gets 5 years in jail (-5). Each prisoner wants to maximize his or her payoff (i.e., each wants to get the fewest years in jail).

The iterated prisoner's dilemma is like the prisoner's dilemma, only repeated over many encounters. Imagine Bonnie and Clyde both cooperate, and after a year in jail, they get together again, rob more banks, and then wind up in the same situation. More to the point, imagine a world of agents who are continually interacting. For example, consider graduate students and their use of disk space. One may cooperate by deleting unused files, but this takes some time and effort. Hence there is a cost to cooperation. If others use disk space without deleting unneeded files (the defectors), you are just subsidizing their lethargy with your time! However, if everyone cooperates, then there will usually be plenty of disk space for all. Similar examples can be found in many social interactions, from driving a car to trade relations among nations.

This problem has two parts.

(a) Files containing Smalltalk source code for the iterated prisoners dilemma are in the directory

   `~leavens/classes/cs541/public/prisoners`

   In the code there are some methods you have to add, and some classes you have to design and add. Look in the **`README`** file in that directory for further directions. For this part you are to extend the simulation of the iterated prisoner's dilemma, by adding and testing at least three (3) different, history-sensitive strategies for the behavior of prisoners to see which is best. A strategy is an algorithm for deciding whether to cooperate or defect, based on internal state such as a history of how the other agent has cooperated or defected in previous rounds, and some random possibilities, (using a random number generator). A strategy is *history-sensitive* if it does different things depending on what its partner did (i.e., it is not completely random or thick-headed).

   Include comments in your program to explain your strategies, and also show the output of test cases. You should submit an English explanation of your program and the strategy you found to be the best, along with your experimental results.

(b) Let us call a run of the program in the preceding problem a *generation*. In this problem you will make another subtype of **`World`**, call it **`EvolvingWorld`** that "evolves" between generations. The idea is to compute a "fitness index" for each strategy, based on its winnings in the previous generation, and to populate the next generation so that the more fit a strategy is, the more descendents it has in the next generation. Your task is to make the program work with the **`EvolvingWorld`**. Run the simulation for, say 5 or 10, generations, and then print the results. (It may be helpful to define another class **`TraceEvolvingWorld`** that prints more debugging information about what happened at each generation.) You will have to come up with a suitable formula for determining the "fitness index" and for assigning strategies to the next generation. (Hint: figure out what strategy did the worst, and watch out for when they all do equally well.)