INST Interpreter in Little-Smalltalk

May 17, 1996

This document explains the design of an interpreter for INST (given in hw2). The code given in this document is available in the file INST.st. We give a sample testcase for the interpreter, give informal semantics for INST, and explain the design of INST interpreter.

A Sample Program in INST

Figure 1 contains a program in INST. It consists of a list of types, a list of classes, a set of methods (or operations) and a main procedure. Types in INST can be an `Integer` or a user-defined type. User-defined types are implemented by classes in INST. Class declarations consist of a list of variables which are like instance variables in Smalltalk. In the figure example, we have a user-defined type `Cell` implemented by a class `cell` with an instance variable `x:Integer`.

Unlike Smalltalk, methods in INST are dispatched based on the class of all the arguments (methods in Smalltalk are dispatched on the class of their first argument or receiver). Methods use a specializer symbol, "`@`", to declare the class of formal parameters. The method declaration in the example, `method value(p@cell:Cell):Integer`, means that the actual argument to this method will handle arguments of class `cell`, which is to be of type `Cell`. (The language can check that the class `cell` implements the type `Cell`.) As in Smalltalk, methods can be overloaded, but the overloading only has to differ in one of the argument classes.

A main procedure in INST consists of a set of "in" variables (D_1) and a set of "out" variables (VD_1). Variables in the set VD_1 are used while executing statement S_1 and variables in the set $VD_1 \cup VD_2$ are used for executing statement S_2. These sets must be disjoint.

```
type Cell;
class cell implements Cell
        instance variables x: Integer;
end;
method createCell():Cell
  is skip
  return new cell;

method set_value(p@cell:Cell, i@Integer:Integer):Cell
  is p.x := i;
  return p;

method value(p@cell:Cell):Integer
  is skip
  return p.x;

observe a: Cell; b: Integer;
        b := 9;
        a := createCell();
        set_value(a, b);
by c:Integer;
   c := value(a);
```

Figure 1: Sample program in INST

To simplify the interpreter and to concentrate on language features rather than on parsing techniques we use abstract syntax in smalltalk to represent various constructs in INST. Table 1 compares the abstract syntax of INST (given in hw2) and the abstract syntax used in the smalltalk interpreter. Abstract syntax for the above program is given in the file: `~leavens/classes/cs541/public/st-interpreter/test1.st`.

Abstract syntax of INST in smalltalk for the program in Figure 1 is given in the file `test1.st` and the abstract syntax of used in the INST interpreter is summarized in Table 1. The following is the script of running Smalltalk code equivalent to the INST program in `test1.INST` file.

```
Script started on Wed Feb  1 15:29:55 1995
% st /usr/unsup/little-smalltalk/systemImage
Little Smalltalk, Version 3.05
Written by Tim Budd, Oregon State University
>       (File new) fileIn: 'INST.st'.
File
>       (File new) fileIn: 'test1.st'.
File
>       r <- Test new.
Test
>       r run.
value of c is 9
Test
>       %exit

script done on Wed Feb  1 15:30:35 1995
```

Design

Figure 2 shows the class hierarchy of the interpreter. Types, classes and methods of INST are compiled into a data structure, which we call algebra. This data structure (algebra) contains the set of types, the set of classes with the types they implement and with their instance variables, and a method dictionary.

Informal Semantics of INST

A program in INST is modeled by a sequence of states. Statements in INST may change the state of a program. Output of a program is the value of the "out" variables (VD_2) in resulting state after executing S_1 and S_2. Expressions in INST return an object and may change the state of the program. Statements and expressions in INST use a data structure called an "algebra" for method lookup.

Comments (in the form of pre- and post-conditions) in the code of the interpreter (see the statement of HW2 for where to find this) explain the meaning of different constructs of INST.

	INST Constructs	Abstract Syntax in smalltalk
TD	type T	#T
VD	I: T	v ← Varble new v varName: #I type: #T
CD	class C implements T instance variables VD	c ← Clss new c name:#C implementsType: #T instanceVar: vdList
F	I@C:T	f ← Formal new f varName: #I className: #C type:#T
B	method I ($\vec{f}$): T `is` S `return` E	methodI ← Operation new. method1 name: #I formals: fl returnType: #T body: cmdC retExp: expE
S	E I := E $I_1.I_2$:= E S_1; S_2 if E then S_1 else S_2 fi skip	s ← ExpStmt new. s stmtExp: expE s ← VarAssignStmt new. s stmtVar: varI withExp: expE s ← InstVarAssignStmt new. s stmtVar: varI_1 withIVar: #I_2 withExp: expE s ← SequenceStmt new. s stmtOne: s1 andStmtTwo: s2 s ← IfStmt new. s stmtBool: expE then: s1 else: s2 s ← SkipSmt new.
E	N I $I_1.I_2$ I($\vec{E}$) new C bot	e ← ValExpression new. e expressionAs: N e ← VarExpression new. e expressionAs: varI e ← DotExpression new. e expressionAs: varI_1 and: #I_2 e ← MessageExpresion new. e expressionAs: #I and: explistE e ← NewExpression new. e expressionAs: #C e ← BotExpression new.

Table 1: Abstract Syntax for constructs in INST

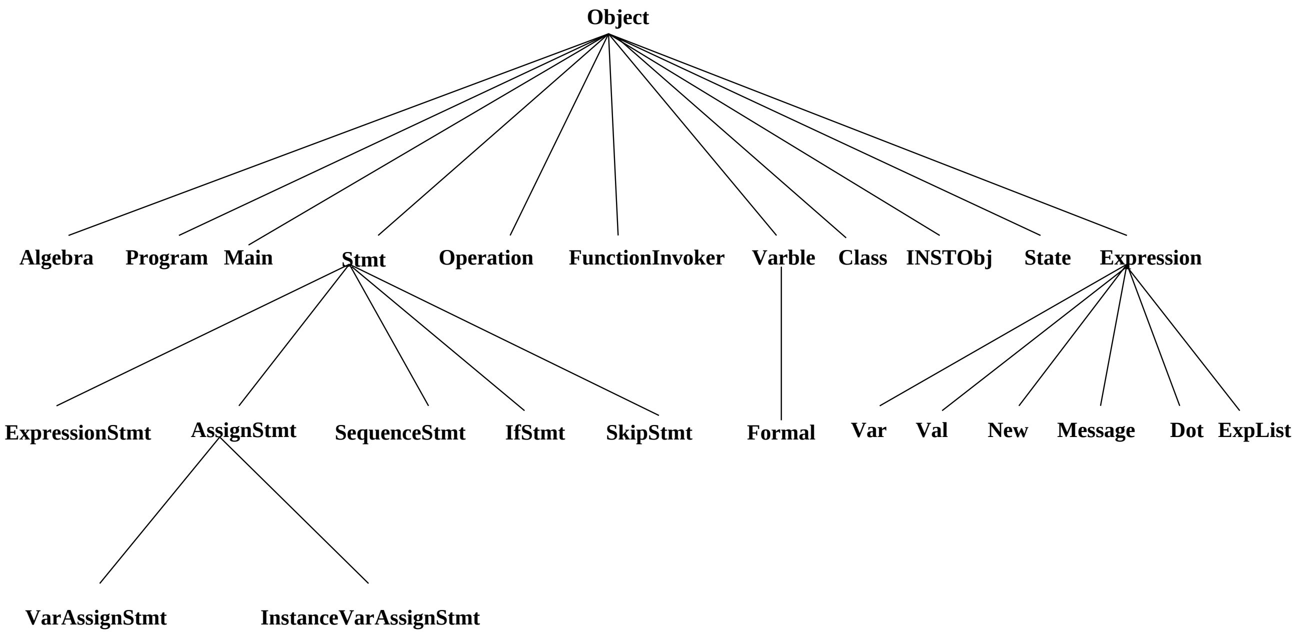

Figure 2: Class hierarchy in INST.st

Algebras

Algebras consists of a tuple of type names, class dictionaries, and method dictionaries.

Programs

Types, classes and operations of an INST program are compiled into an algebra as explained above and the main program is executed in a new state.

Main Procedure

The main procedure of a program consists of a set of "in" variables and a set of "out" variables, and two statements S_1 and S_2.

Statements

Statements in INST may effect the state during execution. This is reflected in the modifies clause of the `executeWithVars:` method. As shown in the Figure 2 different statements are modeled as subclasses of an abstract class `Stmt`.

Expressions

Expressions in INST return an object upon execution. (Note that this object is different from `smalltalk` object.)

Expression List

Executing expression lists return a list of objects.