OEXPL SPECIFICATION

Download as PDF(pending)

Introduction

An informal specification for a minimal object oriented extension of the ExpL languge with support for Inheritance and subtype polymorphism is outlined here.

Class Definitions

Classes extend the notion of ExpL types. A class encapsulates *member fields* and *member functions* (called methods in OOP jargon). The following is an example for a class definition. Standard syntax and semantics conventions followed in languages like Java and C++ are assumed.

```
1   class
2   Person{
3
4           str name;
5           int age;
6           decl
7                   int printDetails();
8                   str findName();
9                   int createPerson(str name, int age);
10          enddecl
11          int printDetails(){
12                  decl            /*  local variables, if any */
13                  enddecl
14                  begin
15                          write(self.name);
16                          write(self.age);
```

```
17                          return 1;
18                  end
19
20          }
21          str findName(){
22                  decl
23                  enddecl
24                  begin
25                          return self.name;
26                  end
27          }
28          int createPerson(str name, int age){
29                  decl
30                  enddecl
31                  begin
32                          self.name=name;
33                          self.age=age;
34                          return 1;
35                  end
36          }
37  }       /*end of Person class*/
38  endclass        /* end of all class definitions */
```

person.txt hosted with ❤ by **GitHub** **view raw**

All the class definitions in a program must be placed together, between the keywords *class* and *endclass*. Declaration of class variables follow exactly the same syntax as declaration of user defined type variables in ExpL. The member fields of a class must be declared before member functions are declared. Class definitions must be placed after type definitions, but before global declarations. (Further example programs are provided at the end).

Since OExpL is designed for pedegogical purposes, the following restrictions are imposed to simplify implementation.

1. The member fields of a class may be of type integer, string, user defined types, previously defined classes or of the same class. Thus the language supports both *Composition and Inheritance.*
2. Member fields of a class are private to the class in the sense that they can be accessed only by the methods defined in the class. The language does not permit *access modifiers* like public or protected.
3. Class variables can be declared only globally. Class variables cannot be arguments to functions; a function cannot have a class as its return type and class variables cannot occur as local variables within functions.
4. In methods defined within a class, the special keyword self refers to the instance of the class through which the method was invoked. (The usage is

similar in spirit to this in C++.)

5. The methods defined in a class may have parameters as well as local variables. The syntax and semantics rules are similar to other ExpL functions. However, there are some important differences:
   a) Methods of a class, apart from its arguments and local variables, have access only to the member fields of the corresponding class.
   b) A method can invoke only functions of the same class or functions inherited from its parent class or methods of class variables occuring as member fields of the class. (Be aware of the *fragile base class problem*).

To put in short

1. class variables can only be global.
2. Member functions of a class can access only its member fields, methods, local variables, arguments and methods of member fields.
3. Member fields of a class can be accessed from outside only through member functions of the class.

Inheritance

The language supports class extension. A class defined by extension of another class is called a *derived class* (or *child class*) of the *parent class* (sometimes called the *base class*). The following example demonstrates the syntax of class extension. The language does not support multiple inheritance .

```
class
Person{
        decl
                str name;
                int age;
                int printDetails();
                str findName();
                int createPerson(str name, int age);
        enddecl
        int printDetails(){
                decl
                enddecl
                begin
                        write(self.name);
                        write(self.age);
                        return 1;
                end
        }
```

```
19          str findName(){
20                  decl
21                  enddecl
22                  begin
23                          return self.name;
24                  end
25          }
26          int createPerson(str name, int age){
27                  decl
28                  enddecl
29                  begin
30                          self.name=name;
31                          self.age=age;
32                          return 1;
33                  end
34          }
35  }     /*end of Person class */
36  Student extends Person{
37
38          decl
39                  int rollnumber;               /*  The members name and a
40                  str dept;
41                  int printDetails();
42                  int createPerson(str name, int age,int rollNo, str dept)
43          enddecl
44          int createPerson(str name, int age,int rollNo, str dept){  /*  c
45                  decl
46                  enddecl
47                  begin
48                          self.name =name;
49                          self.age = age;
50                          self.rollNo = rollNo;
51                          self.dept = dept;
52                          return 1;
53                  end
54          }
55          int printDetails(){  /* This function is also overridden in the
56                  decl
57                  enddecl
58                  begin
59                          write(self.name);
60                          write(self.age);
61                          write(self.rollnumber);
62                          write(dept);
63                          return 1;
64                  end
65          }         /**  The derived class inherits the findName() functio
```

```
66   }  /* end of student class */
67   endclass
```

Inheritance.txt hosted with ❤ by **GitHub** **view raw**

The semantics of class extension can be summarized as follows:

1. The derived class inherits all member fields of the parent class automatically. If additional fields are defined, they will be specific to the derived class. The derived class cannot re-declare a member field already declared in the parent class.
2. The derived class inherits only those methods of the parent class which are not re-defined (overridden). If a method is overridden, the new definition will be the only valid definition for the derived class. All the overridden methods must be declared again in the derived class. The signature of the overridden method must match exactly in both number and types of arguments with the signature of the function in the parent class. Only one function of the same name is permitted in a class. Thus, the language does not permit function overloading.

Class variables and Instantiation

Class variables are declared just like other variables in the global declaration section after type definitions and class definitions.

Example:

```
 1   /* Type definitions */
 2   /* Class definitions */
 3
 4    decl
 5       int n,temp;
 6       string name;
 7       Person first;
 8       Student second;
 9       Person arbitrary;
10    enddecl
```

f1.txt hosted with ❤ by **GitHub** **view raw**

Object instance is created for a variable of a class with the built-in function *new*. The language does not support constructors and destructors. Hence intitialization of objects has to be done explicitly. An object can be deallocated using the built-in

function *delete*. The function *new* will create an object of a specified class at run time, and assigns a *reference* to the object into a variable. A variable of a given class may be assigned a reference to an object of any desendent class using *new*. Access semantics of class variables is similar to ExpL user-defined-types, except for the details associated with methods defined within classes. These details are described below.

Subtype Polymorphism

A variable of a parent class can refer to any object in its inheritance hierarchy. That is, if class B extends class A and class C extends class B, then a variable of class A can refer to objects of classes A, B or C, whereas a variable of class B can refer to any object of class B or C. (References are set using the ExpL *assignment statement* in the normal way, as with user defined types. The function *new* can also be used to set the reference for a variable). When a method is invoked with a variable of class B or C, if the method is inherited from an ancestor class, the ancestor's method is invoked. This is illustrated by the following examples.

```
1   begin
2       first=new(Person);
3       temp = first.createPerson(“Rogers”, 37)              /* invokes met
4       second=new(Student);
5       temp = second.createPerson(“Mathew”, 35, 999, “CS”);  /*invokes meth
6       name = first.findName();                             /* invokes met
7       name = second.findName();                            /* invokes inh
8       delete(first);
9       delete(second);
10  end;
```

main.txt hosted with ❤ by **GitHub** **view raw**

Run time Binding

Suppose that a variable declared to be of a parent class in an inheritance hierarchy holds the reference to an instance of a derived class. On invocation of a method of the parent class variable, if the method is over-ridden by the derived class, the method of the derived class is invoked.

This pivotal feature of object oriented programming compilcates the compiler implementation because at the time of generating code for a method invocation, the correct method to call may not be known, as illustrated in the example below.

```
1   begin
2       first=new(Person);
3       temp = first.createPerson(“Rogers”, 37)              /* invokes meth
4       second=new(Student);
5       temp = second.createPerson(“Mathew”, 35, 999, “CS”); /*invokes metho
6       read(n);                                             /* The run time
7       if (n>0) then
8           arbitrary = first;
9       else
10          arbitrary = second;
11      endif;
12      n = arbitrary.printDetails();                        /* Function not
13      delete(first);
14      delete(second);
15    end;
```

mainex2.txt hosted with ❤ by **GitHub** **view raw**

In the above code, the value of the variable n read from the input at run-time determines whether the variable *arbitrary* refers to an object of the class - person or the class - student. Consequently, at compile time, we cannot decide whether the call *arbitrary.printDetails()* must be translated to an invocation of the function in the parent class or the child class.

To resolve such function invocations, the method of dynamic binding must be implemented using the The virtual function table method .A tutorial explaining the implementation of virtual function tables is given here .

Sample Programs

Example Programs are here.

Appendix

Keywords

The following are the reserved keywords in OExpL and it cannot be used as identifiers.

class	endclass	extends	new	delete	self