MP 5 - Object Oriented Languages

CS421 Agha - Spring 2015

rev. 1.2

Introduction

Cooperation: You may work in pairs for this MP ***only***, your partner may not be necessarily the same as the one for last MP.

Assigned: April 10, 2015
Due: 11:59pm, April 24, 2015 **(the deadline is firm)**
Outline: This assignment covers material from chapter 9 of the EOPL book, 3rd edition.
Grading: This assignment will account for 10% of your grade.

Submission and academic honesty instructions can be found here:

`https://wiki.cites.illinois.edu/wiki/display/cs421sp15/Assignments?src=contextnavchildmode`

Pay special attention to the policies regarding the use of other people's code.

Notation. We use courier font to denote terminals; for example, `newpair`. Nonterminals are italicized, as in *Expr*. Parentheses are literals (i.e. part of the specified grammar), while braces are part of the meta-language we use to describe the grammar. We use the * symbol to denote zero or more repetitions; for example, $\{(Expr\ Var)\}^*$ means zero or more occurrences of $(Expr\ Var)$, including the parentheses. When it is clear from context, we omit the braces, so that Var^* means zero or more repetitions of *Var*.

Problems

For this MP, you need to write an interpreter for an object-oriented language without explicit class definitions. Here the grammar:

Expr	::=	*ArithmeticOp* (*Expr* {, *Expr*}*)
	|	*ComparisonOp* (*Expr* , *Expr*)
	|	`proc` ({*ID*}*) *Expr* `end`
	|	`(set` *Expr* *Expr* `)`
	|	*Integer*
	|	`true` | `false`
	|	*ObjectExp* { `.` *ID*}*
ObjectExp	::=	`begin` { *Expr* `;` }* `end`
	|	`if` *Expr* `then` *Expr* `else` *Expr* `end`
	|	`let` { *ID* = *Expr* }* `in` *Expr* `end`
	|	`letrec` { *ID* = *Expr* }* `in` *Expr* `end`
	|	`(` *Expr* { *Expr* }* `)`
	|	*ID*
	|	`self` | `super`
	|	`EmptyObj`
	|	`extend` *Expr* `with` { *MemberDecl* }*
MemberDecl	::=	{ `public` | `protected` } *ID* = *Expr* `;`

Here, *ID* represents identifiers, that is, names of variables and procedures. You may assume they are simply strings of characters; *ArithmeticOp* denotes the usual +, -, *, / operators; *ComparisonOp* denotes the comparison operators <, >, =. The arithmetic and comparison operators allow only integer operands. Note that the `let` here in our language works like `let*` in Scheme.

Operational Description of the language. Conceptually speaking, every object is an instance of an anonymous class, the definition of which matches that of the object at hand. For instance, `EmptyObj` denotes an object of an empty class which is not explicitly given. To clarify this, if you are a Java programmer, `EmptyObj` would be equivalent to an instance of the `Object` class. Look at the following example:

```
(object-interpreter
  "let ob = extend EmptyObj with public x = 1;
     in ob.x
   end")

> 1
```

Here, we are extending the anonymous, empty class associated with `EmptyObj`, add a field `x` which is initialized to the value 1, and *at the same time* construct an instance of this new class, which is then bound to `ob`. The `let` construct works as usual, and the value of the above expression will be the value bound to `ob.x` (which is the value bound to the field `x` of object `ob`).

Here is the prototypical example of inheritance in this language:

```
(object-interpreter
  "let ob1 = extend EmptyObj with protected x = 1;
     in let ob2 = extend ob1 with public getX = proc () self.x end;
            in (ob2.getX)
          end
   end")

> 1
```

The `extend` construct performs the following:

- It declares an anonymous class, which extends the class of the given object with the provided members.
- It instantiates a new object from the latter class.
- It copies the values of all non-overridden members from the given object.

In the example above, the definition of `ob1` creates an object that is an instance of a class that extends the class of `EmptyObj` with a protected field (`x`). Then `ob2` is an instance of a class that extends the previous one with a method, `getX`. In this language, a *protected* field is accessible in the class defining it, as well as its subclasses (think Java). As a result, `getX` in `ob2` has access to `x` which is "inherited" from `ob1`. Since `getX` is declared public in `ob2`, the call (`ob2.getX`) is valid, and results in the execution of the method.

Note: We said "inherited" (with quotes) in the previous paragraph because `ob1` and `ob2` are not classes; they are objects, that each define a class in an implicit manner. From now on, we will use the term inherit(ance) as if we are talking about classes.

Following is an example of re-defining an inherited field:

```
(object-interpreter
  "let ob1 = extend EmptyObj with public x = 1;
    in let ob2 = extend ob1 with public x = 2;
           in ob2.x
         end
   end")

> 2
```

Here is an example demonstrating dynamic dispatching of methods, as on page 334 of EOPL v3 (Figure 9.6).

```
(object-interpreter
  "let ob1 = extend EmptyObj with
          public m1 = proc() (self.m2) end;
          public m2 = proc() 1 end;

        ob2 = extend ob1 with public m1 = proc() (super.m1) end;
                              public m2 = proc() 2 end;

     in (ob2.m1)
   end")

> 2
```

Contrary to the language used in the book, in our language there is no distinction between fields and methods when it comes to scoping rules: they are both *dynamically* dispatched:

```
1  (object-interpreter
2    "let ob1 = extend EmptyObj with
3
4              protected x = 1;
5              public getX = proc() self.x end;
6
7       in let ob2 = extend ob1 with
8                    protected x = 2;
9                    public getX = proc() (super.getX) end;
10
11            in (ob2.getX)
12          end
13     end")
14
15 > 2
```

To make this clearer: when calling `ob2.getX` (line 11), the call eventually reaches the one defined in `ob1` (line 5). Now there is a difference between Java and this language. In Java, the `x` referred to on line 5 would be the one defined on line 4. In our language however, `self` is dynamically resolved even now, which means it refers to `ob2`. Hence the `x` returned is actually that of line 8.

However: Dynamic dispatching applies when it comes to object members. In all other cases, use static (lexical) scoping.

Our language uses implicit references, as described in Chapter 4.3 in the EOPL book v3. This allows for some cool things, as in the following example:

```
1  (object-interpreter
2    "let ob1 = extend EmptyObj with public foo = 1; in
3         begin
4           (set ob1.foo proc() 3 end);
5
6           let ob2 = extend ob1 with
7                 public foo = proc() 4 end;
8                 public superFoo = proc() (super.foo) end; in
9
10              begin
11                (ob2.foo);
12                (ob2.superFoo);
13              end
14          end;
15        end
16     end")
17
18 > 3
```

First, `ob1` is created, with a public field `foo`, initialized to the value 1. Then, it is set to a procedure that returns the number 3. Then `ob2` is created, that overrides `foo` to a procedure that returns the number 4. It also defines `superFoo` to be a procedure that calls the inherited `foo`. The value of the expression on line 11 is 4, because the method actually called is the one defined on line 7. On line 12, one would expect to use the definition of line 2; however, its value was re-set on line 4, *before* `ob2` was created (before `ob1` was extended). Hence the procedure actually called is the one of line 4, returning 3.

More examples follow. Notice that in the first two cases below, `p` is *not* a member of `ob1`; it is merely another definition in the enclosing `let`. This is where indenting your code properly will help a lot.

```
(object-interpreter
  "let ob1 = extend EmptyObj with
          public foo = proc () 1 end;

       p = proc (x) (x) end

     in (p ob1.foo)
   end")

> 1

(object-interpreter
  "let ob1 = extend EmptyObj with
        public foo = proc () (self.x) end;
        public x = proc() 1 end;

       p = proc (x) (x) end

     in (p ob1.foo)
   end")

> 1

(object-interpreter
  "let ob = extend EmptyObj with

        public foo = proc(y) let x = 1
                              in +(x, y)
                             end
                     end ;

     in (ob.foo 2)
   end")

> 3

(object-interpreter
  "let factObj = extend EmptyObj with

        public fact = proc (n)
          if =(n, 0)
            then 1
            else *(n, (self.fact -(n, 1)))
          end
        end ;

    in (factObj.fact 5)
  end")

> 120
```

```
(object-interpreter
  "let a = extend EmptyObj with
        public b = extend EmptyObj with
          public c = extend EmptyObj with public x = 5 ; ; ;
     in a.b.c.x
   end")

> 5


(object-interpreter

  "let node = extend EmptyObj with

        protected next = 0 ;
        protected element = 0 ;
        public setElement = proc (x) (set element x) end;
        public setNext = proc (x) (set next x) end;
        public getElement = proc () element end;

        public apply = proc (f)
          begin
            (set self.element (f element)) ;
            if =(next, 0) then 0 else (next.apply f) end ;
          end
        end;

      in let head = extend node with
             a = extend node with
             b = extend node with
         in
            begin
              (head.setNext a) ;
              (a.setNext b) ;
              (head.apply proc (x) +(x, 1) end) ;
              (b.getElement) ;
            end
         end
    end")

> 1
```

The last example above is an implementation of a linked list. The list includes an `apply` method that takes a function as its argument and applies it to each element in the list. The list then contains the newly produced elements in place of the old ones.

The value of erroneous programs should be the *symbol* `undefined`. This is with regard to errors other than syntax errors – in which case just let SLLGEN output its own thing. For example, accessing a protected member outside the defining class (or a subclass) should result in the value `'undefined`:

```
(object-interpreter
  "let ob = extend EmptyObj with
              protected x = 1;
     in ob.x
   end")

> 'undefined
```

Notes on the language syntax. In order to use SLLGEN, the grammar has to be LL(1). As a result, there are a few noteworthy points, including a few changes to the syntax of your familiar constructs.

- Every major construct ends with the keyword `end`. This includes procedure definitions, if-statements, `let`, etc.
- Object member declarations end with a semi-colon.
- Every expression that is part of a begin-end construct has to be followed by a semi-colon, including the last one.
- To implement `set` in SLLGEN, include the opening parenthesis in the same string as "set" itself. In other words, the rule should start with "(set". If you separate the parenthesis out, you will get a "not LL(1)" error (because there is another rule that starts with a left parenthesis).

Submission reminder. Name your interpreter "object-interpreter" and put it in your mp5-1.scm file.

Good luck!