CIS311 Programming project 3

due before Final week Dec 2nd

You will extend the interpreter you built in Programming project 2. The extensions go in three parts.

Part C: add objects

First, you add object declarations and objects. The objects are "structs" that hold var and proc declarations, like we saw in the Lecture Note 6. A sample program will look and work like this:

```
===================================================

python run.py
Type program; OK to do it on multiple lines; terminate with  !
  as the first symbol on a line by itself:

int a = 1;
ob counter = new {int val = a;
                  proc inc(x) : val = ((val + x) + a);  print val; end;
                 };
proc p(): a = (a + counter.val)  end;
ob nothingyet = nil;

counter.inc(1);
p();
nothingyet = counter
!

===================================================
```

Here is the execution of the above program:

```
===================================================

Parse tree:
[[['int', 'a', '1'], ['ob', 'counter', ['new', ['struct', [['int', 'val', ['deref', 'a']], ['proc', 'inc', ['x'], [], [['=', 'val', ['+', ['+', ['d
Execution:
3
activation stack = ['h0', 'h4']
heap = {
  h0 : {'a': 1, 'parentns': 'nil', 'counter': 'h1', 'nothingyet': 'nil', 'p': 'h3'}
  h1 : {'parentns': 'h0', 'val': 3, 'inc': 'h2'}
  h2 : {'body': [['=', 'val', ['+', ['+', ['deref', 'val'], ['deref', 'x']], ['deref', 'a']]], ['print', ['deref', 'val']]], 'params': ['x'], 'type
  h3 : {'body': [['=', 'a', ['+', ['deref', 'a'], ['deref', ['dot', 'counter', 'val']]]]], 'params': [], 'type': 'proc', 'link': 'h0', 'locals': []
  h4 : {'x': 1, 'parentns': 'h1'}
}
Successful termination.
activation stack = ['h0']
heap = {
  h0 : {'a': 4, 'parentns': 'nil', 'counter': 'h1', 'nothingyet': 'h1', 'p': 'h3'}
  h1 : {'parentns': 'h0', 'val': 3, 'inc': 'h2'}
  h2 : {'body': [['=', 'val', ['+', ['+', ['deref', 'val'], ['deref', 'x']], ['deref', 'a']]], ['print', ['deref', 'val']]], 'params': ['x'], 'type
  h3 : {'body': [['=', 'a', ['+', ['deref', 'a'], ['deref', ['dot', 'counter', 'val']]]]], 'params': [], 'type': 'proc', 'link': 'h0', 'locals': []
  h4 : {'x': 1, 'parentns': 'h1'}
  h5 : {'parentns': 'h0'}
}

===================================================
```

In the example, `h0` is the handle to the global variables' namespace. Variable `counter` is bound to the handle of a new namespace/object, `h1`, that holds `val`, `inc`, and `parentns` (which is needed when evaluating the expressions that appear in the object's `int` declarations).

When `counter.inc(1)` is called, it works just as you implemented it in Programming project 2: a new activation record, `h4`, is constructed for the call to `inc`, and `h4` is pushed onto the activation stack. (Note that `h4` holds a `parentns` link that is set to `h1`, which is `inc`'s "parent object".)

Once `inc`'s code finishes, the stack is popped. Then `p()` gets called, and its activation, `h5`, is pushed then popped.

You must implement these new parts of the language:

```
D ::=  ...  |  ob I = E
E ::= ... |  new T  |  nil
T ::=  { DL }
L ::=  I  |  L . I
```

Here is the complete syntax, with the new parts included:

```
===================================================

 P : Program               E : Expression
 CL : CommandList          T : TypeTemplate
 C : Command               L : LefthandSide
 DL : DeclarationList      IL : VariableList
 D : Declaration           I : Variable
 EL : ExpressionList       N : Numeral
```

```
P ::=  DL CL

DL ::=  D;*
D ::=  int I = E  |  ob I = E  |  proc I ( IL ) : CL end

CL ::=  C;*
C ::=  L = E  |  if E : CL1 else CL2 end  |  print E  |  L ( EL )

EL ::=  E,*
E ::=  N  |  ( E1 OP E2 )  |  L  |  new T  |  nil
    where  OP ::=  +  |  -
T ::=  { DL }
L ::=  I  |  L . I

N ::=  string of digits
IL ::=  I,*
I ::=  strings of letters, not including keywords

===================================================
```

To be safe, please use the `a231ex.py` and `a23pars.py` files included and use them to replace what you used for project 2. These two files were revised.

Interpreter input format

The input to the interpreter is the list-based parse tree constructed by the parser. The new constructions are:

```
DTREE ::=   ... |  ["ob", ID, ETREE]
ETREE ::=  ...  |  "nil"  |  ["new",  TTREE]
TTREE ::=  ["struct", DLIST]
LTREE ::=  ID  |  ["dot", LTREE, ID]
```

The syntax of all forms of parse trees goes like this:

```
===================================================

PTREE ::=  [DLIST, CLIST]
DLIST ::=  [ DTREE* ]
           where  DTREE*  means zero or more DTREEs
DTREE ::=  ["int", ID, ETREE]  |  ["proc", ID, IDLIST, CLIST, DLIST]  |  ["ob", ID, ETREE]

CLIST ::=  [ CTREE* ]
CTREE ::=  ["=", LTREE, ETREE]  |  ["if", ETREE, CLIST, CLIST]
        |  ["print", ETREE]  |  ["call", LTREE, ELIST]

ELIST ::=   [ ETREE* ]
ETREE ::=  NUM  |  [OP, ETREE, ETREE] |  ["deref", LTREE]  |  "nil"  |  ["new",  TTREE]
      where  OP ::=  "+"  | "-"
TTREE ::=  ["struct", DLIST]
LTREE ::=  ID  | ["dot", LTREE, ID]

NUM   ::=  a nonempty string of digits
IDLIST ::= [ ID+ ]
ID    ::=  a nonempty string of letters

===================================================
```

Interpreter operation

You start from the interpreter you built for project 2. (*If you didn't complete at least project 2, Part A, you must finish it before you start this one* --- please talk to me as soon as possible to make an appointment to fix your project 2 submission. Sorry, but it's useless to do this assignment without learning how to do project 2, Part A.)

You have these structures to implement: `"nil"`, `["new", TTREE]`, `["ob", ID, ETREE]`, `["struct", DLIST]`, and `["dot", LTREE, ID]`.

1. Within `interpretETREE`, implement `"nil"` to have itself as its value. Implement `["new", T]` to call `interpretTTREE(T)`, whose job is to allocate an object, fill it with `T`, and return the object's handle.

   You define `def interpretTTREE(ttree)`. It receives arguments of the form, `["struct", DLIST]`. The function does this: *(i)* allocates a new namespace and *pushes the namespace's handle on the activation stack*; *(ii)* evaluates `DLIST`; *(iii)*pops the activation stack and returns the popped handle as its answer.

2. Within `interpretDTREE`, implement `["ob", I, E]`, which *(i)* computes the meaning of `E`, *(ii)* validates that `E` is either a handle to an object or is `nil`, and *(iii)* binds `I` to the meaning in the active namespace (provided that `I` is not already declared there).

3. Within `interpretLTREE`, implement `["dot", L, I]`. This means you compute the handle named by `L`, call it `h`, and then check if the pair, `(h,I)` is a valid L-value (that is, variable `I` is a field inside the object named by `h`). *Big Hint:* read the code for the virtual machine in Chapter 2, Section 2.2.1.

As usual, enforce declaration checking --- the same name cannot be declared twice in the same namespace and no name can be referenced or assigned to if it is undeclared (this includes fields within objects). Also enforce type checking in commands and expressions: *Only a handle (or `nil`) can be assigned to an object variable, and only an int can be assigned to an int variable*. (Hint: use Python's `instanceof` or `type` operator to check types. See the notes, Terse notes on lists and dictionaries on the CIS505 web page.) Only ints can be used in arithmetic; only procs can be called; and only objects can be indexed with dot notation.

Remember to document appropriately your modified interpreter.

Testing

The `Ex3` folder contains a file of test cases that you should use for testing Part C. Use at least these tests to check your implemetation. You should also devise 2-3 additional tests to see if the interpreter detects program errors and prints appropriate messages.

Place all the test cases and their output in a file named `tests.txt`

Part D: add classes

Don't start this part unless you have built and tested successfully Part C.

Now you extend the interpreter with classes. A sample program looks like this:

```
===================================================

python run.py
Type program; OK to do it on multiple lines; terminate with  !
  as the first symbol on a line by itself:

int a = 2;
class counter : {int val = 8;
                 proc inc(x) : val = (val + x); end;
                };
ob c = new counter;
proc p(): a = (a + c.val)  end;
c.inc(1);
p();
!

===================================================
```

Here is the execution of the above program:

```
===================================================

Parse tree:
[[['int', 'a', '2'], ['class', 'counter', ['struct', [['int', 'val', '8'], ['proc', 'inc', ['x'], [], [['=', 'val', ['+', ['deref', 'val'], ['deref
Execution:
Successful termination.
activation stack = ['h0']
heap = {
  h0 : {'a': 5, 'parentns': 'nil', 'c': 'h2', 'counter': 'h1', 'p': 'h4'}
  h1 : {'body': ['struct', [['int', 'val', ['deref', 'a']], ['proc', 'inc', ['x'], [], [['=', 'val', ['+', ['deref', 'val'], ['deref', 'x']]]]]]],
 'type': 'class'}
  h2 : {'parentns': 'h0', 'val': 3, 'inc': 'h3'}
  h3 : {'body': [['=', 'val', ['+', ['deref', 'val'], ['deref', 'x']]]], 'params': ['x'], 'type': 'proc', 'link': 'h2', 'locals': []}
  h4 : {'body': [['=', 'a', ['+', ['deref', 'a'], ['deref', ['dot', 'c', 'val']]]]], 'params': [], 'type': 'proc', 'link': 'h0', 'locals': []}
  h5 : {'x': 1, 'parentns': 'h2'}
  h6 : {'parentns': 'h0'}
}

===================================================
```

The syntax has these two additions:

```
D ::=  ...  |  class I : T
T ::=  ...  |  L
```

That is, we can declare a class and call it. The parser already implements the new constructions.

Interpreter input format

You must implement in your interpreter,

```
DTREE ::=  ...  |  ["class", ID, TTREE]
TTREE ::=  ...  |  ["call", LTREE]
```

Interpreter operation

There are two steps:

1. Within `interpretDTREE`, implement `["class", I, T]`, which behaves like procedure declaration, that is, `I` is bound to a closure containing `T` and its link to global variables. Please study the above example program.
2. Within `interpretTTREE`, implement `["call", L]`. This works like procedure call, where `L` is computed to a handle, the closure labelled by the handle is extracted from the heap, and provided that the closure holds a class, the `TTREE` within the closure is extracted and executed.

Testing

The `Ex3` folder contains a file of test cases that you should use for Part D. Use at least these tests to check your implemetation. Place your tests and their outputs in your `tests.txt` file. You should also devise additional tests to see if the interpreter detects program errors and prints appropriate messages.

Part E: add subclasses

Finally, we allow subclasses to extend super-classes. A subclass can hold methods that *override* methods in the superclass. The syntax of declarations and templates become

```
D ::=  ...  |  override I ( IL ) : DL CL end
T ::=  extend T with { DL }  |  { DL }  |  L
```

and the operator trees have two new constructions:

```
DTREE ::=  ... |  ["override", ID, ILIST, DLIST, CLIST]
TTREE ::= ...  |  ["extend", TTREE, DLIST]
```

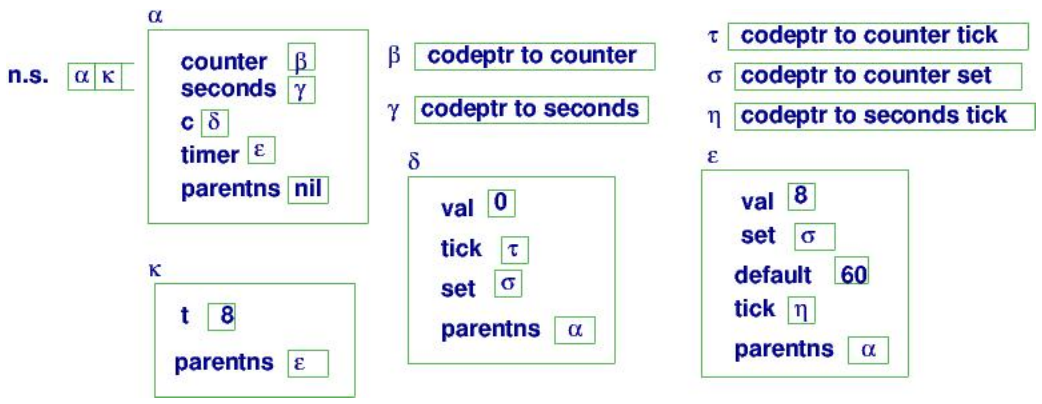
Here is an example:

```
===================================================

class counter : { int val = 0;
                  proc tick() : val = (val + 1); end;
                  proc set(t) : val = t;  // (*)
                  end;
};
class seconds : extend counter with {
                 int default = 60;
                 override tick() :
                    if val : val = (val - 1);
                    else : set(default)
                 end;
};
ob c = new counter;
ob timer = new seconds;
c.tick();
print timer.val;
print timer.default;
timer.set(8);  timer.tick();

===================================================
```

Here is a diagram of the storage layout when the execution is doing the call, `timer.set(8)`, at point `(*)`:



The picture shows these two important modifications:

- `timer` names an object that holds the fields and methods of both classes `counter` and `seconds`, but *the override method, `tick`, "erased" the same-named method declared in the superclass.*
- Closures no longer hold handles to nonlocal variables. Instead, when a closure is called, the handle is calculated at the point of call:

  For example, the call, `timer.set(8)`, calculates the L-value coordinates, `(ε, 'set')`, fetches `σ` at those coordinates, and executes the closure code at handle `σ`. *`ε` is used as the value of `parentns`* for the call to `timer.set`.

The layout looks a bit like Java's, and it is simpler than the one in the Lecture Notes --- no need for a `super`-link, and `parentns` is the same as the `this`-link. (I am trying to be nice to you and give you less work to do.)

Here are the semantics of the new constructions:

1. The semantics of `override I(...): CL end` is to declare `I` in the current, active namespace, *provided that `I` is already declared there*. Otherwise, the interpreter signals an error (that there is no method to override).
2. The semantics of `extends T with { DL }` is to invoke the TypeTemplate, `T`, which constructs an object and returns its handle. Next, the declarations in `DL` are added to the object just constructed.
3. In addition, you must alter the semantics of procedure call: For `L(EL)`, do these steps *(i)* Compute `interpretLTREE(L)`, to get the L-value coordinates, `(han,name)`. The value at `(han,name)` is fetched from the heap --- it should be the handle of a proc closure. Extract from that closure `IL` and `CL`. *(ii)* evaluate `EL` to a list of values *(iii)* Allocate a new namespace and push the new namespace's handle onto the activation stack. *(iv)* Within the new namespace, bind `parentns` to `han`(!!!); bind the values from `EL` to the corresponding names in `IL`. *(v)* Execute `CL`, and upon completion pop the activation stack.

An advantage of this implementation is that you do not add `this` and `super` links to objects --- you use `parentns` as before. Also, you need not recode `interpretLTREE`. Also, all the previous examples in Stages A-D work the same as before. (What you *lose* with this simplified implementation is the ability to distinguish between fields and methods in the superclass from those in the subclass. This becomes a problem when you extend classes that other people wrote --- your code can "mess" with their fields and "erase" their methods. The very last required test case exposes this flaw.)

Testing

The `Ex3` folder contains a file of test cases that you should use for Part E. Use at least these tests to check your implemetation. Place your tests and their outputs in your `tests.txt` file.

Submission and grading

Place your versions of `heapmodule.py`, `interpret.py`, and `tests.txt` in the `Ex3` folder. (Don't alter and don't submit the other files.) Zip the folder into a `.zip` file and submit the `.zip` file to Blackboard or another better place. *Each group just submit together.*

I will study your work and your tests and apply some additional tests, and arrange a meeting with each group to see each of your understanding to grade.