AST-basierte Interpreter: Funktionen und Klassen

Carsten Gips (FH Bielefeld)

Funktionen

```
int foo(int a, int b, int c) {
    print a + b + c;
}

foo(1, 2, 3);
```

Funktionen

```
int foo(int a, int b, int c) {
    print a + b + c;
}

foo(1, 2, 3);
```

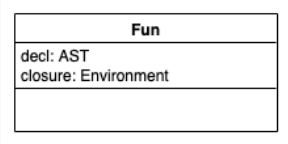
```
def makeCounter():
    var i = 0
    def count():
        i = i + 1
        print i
    return count;

counter = makeCounter()
counter()   # "1"
counter()   # "2"
```

Ausführen einer Funktionsdeklaration

```
funcDecl : type ID '(' params? ')' block ;
funcCall : ID '(' exprList? ')' ;
```

```
def funcDecl(self, AST t):
    fn = Fun(t, self.env)
    self.env.define(t.ID().getText(), fn)
```



Quelle: Eigener Code basierend auf einer Idee nach (Nystrom 2021), LoxFunction.java (MIT)

Ausführen eines Funktionsaufrufs

```
funcDecl : type ID '(' params? ')' block ;
funcCall : ID '(' exprList? ')' ;
```

```
def funcCall(self, AST t):
    fn = (Fun)eval(t.ID())
    args = [eval(a)  for a in t.exprList()]

    prev = self.env;  self.env = Environment(fn.closure)
    for i in range(args.size()):
        self.env.define(fn.decl.params()[i].getText(), args[i])

    eval(fn.decl.block())
    self.env = prev
```

Quelle: Eigener Code basierend auf einer Idee nach (Nystrom 2021), LoxFunction.java (MIT)

Funktionsaufruf: Rückgabewerte

```
def funcCall(self, AST t):
    ...

    eval(fn.decl.block())

    ...
    return None   # (Wirkung)
```

Funktionsaufruf: Rückgabewerte

```
def funcCall(self, AST t):
    ...

    eval(fn.decl.block())

    ...
    return None   # (Wirkung)
```

```
class ReturnEx(RuntimeException):
    __init__(self, v): self.value = v

def return(self, AST t):
    raise ReturnEx(eval(t.expr()))

def funcCall(self, AST t):
    ...
    erg = None
    try: eval(fn.decl.block())
    except ReturnEx as r: erg = r.value
    ...
    return erg;
```

Native Funktionen

```
class Callable:
    def call(self, Interpreter i, List<Object> a): pass
class Fun(Callable): ...
class NativePrint(Fun):
    def call(self, Interpreter i, List<Object> a):
        for o in a: print a  # nur zur Demo, hier sinnvoller Code :-)

# Im Interpreter (Initialisierung):
self.env.define("print", NativePrint())

def funcCall(self, AST t):
    ...
#    prev = self.env;  self.env = Environment(fn.closure)
#    for i in range(args.size()): ...
#    eval(fn.decl.block()); self.env = prev
    fn.call(self, args)
    ...
```

Quelle: Eigener Code basierend auf einer Idee nach (Nystrom 2021), LoxCallable.java, LoxFunction.java (MIT)

Klassen und Instanzen I

```
classDef : "class" ID "{" funcDecl* "}" ;
```

```
def classDef(self, AST t):
    methods = HashMap<String, Fun>()
    for m in t.funcDecl():
        fn = Fun(m, self.env)
        methods[m.ID().getText()] = fn

    clazz = Clazz(methods)
    self.env.define(t.ID().getText(), clazz)
```

Quelle: Eigener Code basierend auf einer Idee nach (Nystrom 2021), Interpreter.java, (MIT)

Klassen und Instanzen II

```
class Clazz(Callable):
    __init__(self, Map<String, Fun> methods):
        self.methods = methods

    def call(self, Interpreter i, List<Object> a):
        return Instance(self)

    def findMethod(self, String name):
        return self.methods[name]

class Instance:
    __init__(self, Clazz clazz):
        self.clazz = clazz

    def get(self, String name):
        method = self.clazz.findMethod(name)
        if method != None: return method.bind(self)
        raise RuntimeError(name, "undefined method")
```

Quelle: Eigener Code basierend auf einer Idee nach (Nystrom 2021), LoxClass.java, LoxInstance.java (MIT)

Zugriff auf Methoden (und Attribute)

```
getExpr : obj "." ID ;
```

```
def getExpr(self, AST t):
    obj = eval(t.obj())

    if isinstance(obj, Instance):
        return ((Instance)obj).get(t.ID().getText())

    raise RuntimeError(t.obj().getText(), "no object")
```

Methoden und *this* oder *self*

```
class Fun(Callable):
    def bind(self, Instance i):
        e = Environment(self.closure)
        e.define("this", i)
        e.define("self", i)
        return Fun(self.decl, e)
```

Quelle: Eigener Code basierend auf einer Idee nach (Nystrom 2021), LoxFunction.java, (MIT)

Wrap-Up

- Interpreter simulieren die Programmausführung
  - Namen und Symbole auflösen
  - Speicherbereiche simulieren
  - Code ausführen: Read-Eval-Loop
- Traversierung des AST: `eval(AST t)` als Visitor-Dispatcher
- Scopes mit `Environment` (analog zu Symboltabellen)
- Interpretation von Funktionen (Deklaration/Aufruf, native Funktionen)
- Interpretation von Klassen und Instanzen

LICENSE