

AST-basierte Interpreter: Basics

Carsten Gips (FH Bielefeld)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

Aufgaben im Interpreter

```
int x = 42;
int f(int x) {
    int y = 9;
    return y+x;
}

x = f(x);
```

- Aufbauen des AST
- Auflösen von Symbolen/Namen
- Type-Checking und -Inference
- Speichern von Daten: Name+Wert vs. Adresse+Wert
- Ausführen von Anweisungen
- Aufruf von Funktionen und Methoden

AST-basierte Interpreter: Visitor-Dispatcher

```
def eval(self, AST t):  
    if t.type == Parser.BLOCK : block(t)  
    elif t.type == Parser.ASSIGN : assign(t)  
    elif t.type == Parser.RETURN : ret(t)  
    elif t.type == Parser.IF : ifstat(t)  
    elif t.type == Parser.CALL : return call(t)  
    elif t.type == Parser.ADD : return add(t)  
    elif t.type == Parser.MUL : return mul(t)  
    elif t.type == Parser.INT : return Integer.parseInt(t.getText())  
    elif t.type == Parser.ID : return load(t)  
    else : ... # catch unhandled node types  
    return None;
```

Auswertung von Literalen und Ausdrücken

- Typen mappen: Zielsprache => Implementierungssprache
- Literale auswerten:

```
INT: [0-9]+ ;
```

```
elif t.type == Parser.INT : return Integer.parseInt(t.getText())
```

- Ausdrücke auswerten:

```
add: e1=expr "+" e2=expr ;
```

```
def add(self, AST t):  
    lhs = eval(t.e1())  
    rhs = eval(t.e2())  
    return (double)lhs + (double)rhs  # Semantik!
```

```
ifstat: 'if' expr 'then' s1=stat ('else' s2=stat)? ;
```

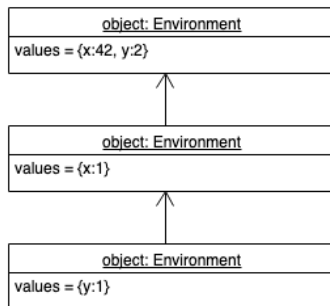
```
def ifstat(self, AST t):  
    if eval(t.expr()): eval(t.s1())  
    else:  
        if t.s2(): eval(t.s2())
```

Zustände: Auswerten von Anweisungen

```
int x = 42;  
float y;  
{  
    int x;  
    x = 1;  
    y = 2;  
    { int y = x; }  
}
```

Zustände: Auswerten von Anweisungen

```
int x = 42;  
float y;  
{  
    int x;  
    x = 1;  
    y = 2;  
    { int y = x; }  
}
```



Environment
enclosing: Environment values: Map<String, Object>
define(name: String, value: Object): void assign(name: String, value: Object): void get(name: String): Object

Detail: Felder im Interpreter

```
class Interpreter(BaseVisitor<Object>):  
    __init__(self, AST t):  
        BaseVisitor<Object>.__init__(self)  
        self.root = t  
        self.env = Environment()
```

Quelle: Eigener Code basierend auf einer Idee nach (Nystrom 2021) und angepasst auf ANTLR-Visitoren, Interpreter.java (MIT)

Ausführen einer Variablendeklaration

```
varDecl: "var" ID ("=" expr)? ";" ;
```

```
def varDecl(self, AST t):  
    # deklarierte Variable (String)  
    name = t.ID().getText()  
  
    value = None; # TODO: Typ der Variablen beachten (Defaultwert)  
    if t.expr(): value = eval(t.expr())  
  
    self.env.define(name, value)  
  
    return None
```

Ausführen einer Zuweisung

```
assign: ID "=" expr;
```

```
def assign(self, AST t):  
    lhs = t.ID().getText()  
    value = eval(t.expr())  
  
    self.env.assign(lhs, value)  # Semantik!  
}  
  
class Environment:  
    def assign(self, String n, Object v):  
        if self.values[n]: self.values[n] = v  
        elif self.enclosing: self.enclosing.assign(n, v)  
        else: raise RuntimeError(n, "undefined variable")
```

Blöcke: Umgang mit verschachtelten Environments

```
block: '{' stat* '}' ;
```

```
def block(self, AST t):  
    prev = self.env  
  
    try:  
        self.env = Environment(self.env)  
        for s in t.stat(): eval(s)  
    finally: self.env = prev  
  
    return None;
```

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

- 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 Blöcken und Variablen (Deklaration, Zuweisung)

LICENSE



Unless otherwise noted, this work is licensed under CC BY-SA 4.0.