# Notizen

Inhalt

[Notizen 1](#_Toc65919183)

[Variablen 2](#_Toc65919184)

[Funktionsweise des Stacks 11](#_Toc65919185)

[SymbolTable 12](#_Toc65919186)

## Variablen

Alle Variablen sollen später durch IDs identifizierbar und lesbar sein

Variablen werden immer in einem Scope gespeichert und gelten auch nur in diesem

Die Variable a ist

nun genau so in der Entry geltend.

Findet der Parser nun einen Identifier, schaut er, ob dieser im aktuellen Scope deklariert

wurde.

In diesem Beispiel:

result = a + b

Findet er b im aktuellen Scope, aber a nicht.

Nun schaut er im nächsthöheren Scope nach, ob die

Variable dort deklariert wurde.

Das höchste und letzte Scope ist immer die

Attributebene. Dort findet er auch die deklaration von a und nutzt diese.

cls Test():

int a = 10

ntr ():

int b = 2

int result = a + b

print(result)

Parameter werden beim Methodenaufruf wie normale Variablen erzeugt.

D.h. der Parameter a ist in dem Scope der gesamten Entry gültig.

Sollte nun eine Variable mit dem Namen a aufgerufen werden,

So würde der Parser zuerst den Parameter a finden und diesen nutzen.

Soll jedoch das Attribut mit dem gleichen Namen genutzt werden, kann

man self/this benutzen.

Beispiel:

cls Test():

int a = 10

ntr (int a):

int b = 2

int result = a + b #Der Parameter wird genutzt

print(result)

cls Test():

int a = 10

ntr (int a):

int b = 2

int result = self.a + b #Das Attribut der Klasse wird genutzt

print(result)

#=======================================

# Speichern und aufrufen von Variablen

#=======================================

Objekte haben immer die gleiche Struktur.

Beispiel:

cls Addition:

int a

int b

int result

fnc add(int a,int b):

self.a = a

self.b = b

result = a+b

Die Klasse Addition hat immer die gleiche Anzahl an

Attibuten, sowie Funktionen.

Wenn man alle Attribute also als Array[extends Object]

speichert, so kann man davon ausgehen dass:

for(0,10,1):

new Addition()

Alle hier erzeugten Objekte der Klasse Addition an Stelle [0] des

Arrays die Variable a(int) gespeichert haben.

Zwar haben alle Instanzen der Klasse eigene Attribute. Diese sind

aber lokal in gleicher Reihenfolge gespeichert und somit voraussehbar.

Globale Variablen sind eindeutig identifizierbar. Egal wie viele

Instanzen man erzeugt die Variable bleibt immer bestehen.

//Variablen in Methoden

Variablen die in Methoden erzeugt werden, sowie die Parameter sind auch immer

in der selben Reihenfolge

Beispiel:

fnc test (int a,Test t):

str text = t.getText()

int b = 12 \* pi

Die Funktion Test hat folgende Einträge:

[0] int a

[1] Test t

[2] str test

[3] int b

Alle variablen haben eine ID.

Diese gibt die Speicheradresse und somit die Variable zurück

Da Statische Variablen immer existend sind,können sie durch die ID genau identifiziert werden.

Bei Objekten sieht das ganze anders aus.

Man kann tausende Objekte vom Typ Addition machen und jedes hat seine eigene Version von (a,b und result)

Da Objekte während laufzeit erstellt und gelöscht werden können variieren auch die Speicheradressen.

Deshalb kann man sie mit der ID nicht eindeutig identifizieren.

Da sie aber lokal in ihrem Objekt immer die gleiche Position im Speicher einnehmen, bekommen sie eine lokale

ID.

Beispiel:

cls Addition:

\*int a

int b

int result

fnc add(int a,int b):

Addition.a = a

self.b = b

result = a+b

Die Variable a ist nun statisch.

Sie existiert immer zu Programmbegin und erhält

deshalb auch immer die gleiche ID.

Die lokale Variablen b und result hingegen existieren erst,

wenn ein Instanz von Addition erzeugt wird.

Und da dies Zeitlich variieren kann, vorallem wenn Threads genutzt werden.

Variiert auch die vergebene Speicheradresse.

Dennoch haben die beiden Variablen lokal in ihrer Instanz immer den festen Platz:

[0] -> Zeiger zu statischer variable

[1] int b

[2] int result

also wissen wir egal wann und wieviele Objekte wir davon erzeugen.

dass Addition[1] immer int b, und

Addition[2] immer int result ist.

=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

!Alle Statischen Variablen kommen auf den Stack!

=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

#=======================================

# Symbol Table

#=======================================

Um zu schauen, welche Variablen existieren und welche nicht.

Erstellt und verwaltet der Parser für jedes Scope einen Symobltable.

Im Symboltable sind alle Verweise mit ihren Namen, Art(Variable/Methode) und Datentyp hinterlegt

Beispiel:

cls Test:

int var1

fnc func\_1():

int var2

if():

int var3

int var4

fnc func\_2()->int:

int var5

if():

int var6

if():

int var7

int var8

->1

=====Test=====

var1 Variable int

func\_1 Function void

func\_2 Function int

=====func\_1=====

var2 Variable int

var4 Variable int

=====func\_1, Scope 1=====

var3 Variable int

=====func\_2=====

var5 Variable int

var8 Variable int

=====func\_2, Scope 1=====

var6 Variable int

=====func\_2, Scope 2=====

var7 Variable int

Das ganze ist in Form eines Trees gespeichert.

Somit hat jede Klasse und Methode sowie jedes Scope der Methode einen eigenen Symboltable

So kann man nun rekursiv bis in den obersten Symboltable nach Variablen, oder Funktionen suchen

Der "Wurzel-" Symboltable ist der Globale Symboltable. Jede Klasse/Funktion hat Zugriff auf die

globalen Variablen/Funktionen

!!! Eventuell sollte im Symbol table auch direkt die zugehörige ID gespeichert werden,

Dabei muss aber zwischen der globalen und der lokalen ID unterschieden werden !!!

#=======================================

# RAM Management

#=======================================

Static-Heap:

Beinhaltet alle statischen Variablen/Funktionen/Objekte.

Die größe bleibt immer gleich.

Heap:

speichert alle nicht-statischen Objekte inklusive ihrer Attribute

Method-Heap:

Speichert alle nicht-statischen Methoden.

Methoden werden hinzugefügt/entfernt, sodass nur Methoden von

Objekten, die gerade Instanziert sind aufrufbar sind.

Beispiel:

Test t = new Test()

#Speichert Objekt t in dem Heap und lädt alle damit verbundenen nicht-statischen Methoden in den Method-Heap

Test t2 = new Test()

Test t3= new Test()

#werden nun weitere Objekte des gleichen Datentypen erzeugt, können sie auf die schon erstellten Methoden zugreifen.

t = null

t1 = null

t2 = null

#Werden alle Objekte des Typs gelöscht. So werden auch die damit verbundenen nicht-statischen Methoden aus dem Method-Heap gelöscht

#Eventuell sollten auch nicht-statische Methoden immer existieren

## Funktionsweise des Stacks

cls test:

ntr():

int a = 10

int b = 4

int result = add(a,b)

fnc add(int a,int b)->int:

-> a+b

stackframe für entry wird erstellt

entry:

int a

int b

int result -> call add(a,b)

stackframe für Methodenaufruf wird erstellt.

stackframe nimmt die parameter an.

Inhalt der Methode wird ausgeführt.

Sobald die Methode fertig abgelaufen ist wird der stackframe gelöscht

und das ergebnis übergeben.

ist in der Entry alles abgelaufen, so wird auch der dazugehörige stackframe gelöscht.

Dass Programm ist nun beendet und sämtliche Heaps werden befreit.

//Erstmals ist alles statisch

Also nur Static-Heap einfügen

## SymbolTable

-> wenn eine Variable initialisiert/deklariert wird, soll sie im dazugehörigen

symbol table gespeichert werden

-> wenn eine Variable genutzt/ geändert wird, muss im symboltable nachgeschaut weden, ob sie vorhanden ist .. falls nicht programmabbruch

------------------------------------------------------------------------

Zuerst wird immer der static SymbolTable erstellt.

Wenn eine Klasse gefunden wird, wird ein Symboltable erstellt und mit dem root-SymbolTable verknüpft

Wenn ein neues Scope anfängt (Sich die Anzahl an Tabs erhöht), wird an dem aktuellen SymbolTable ein neuer

verknüpft

Wenn ein bestehendes Scope aufhört (Sich die Anzahl an Tabs verringert), wird der aktuelle Symboltable verlassen und

der parent-SymbolTable aufgerufen

Da SymbolTables sich rekursiv aufrufen, wenn sie vollständig sind, sollte man am Ende wieder beim rootTable rauskommen

und diesen verlassen, da er vollständig ist

!BESONDERHEIT!

Der Root-Table kann als einziger jederzeit gefüllt werden, da er Variablen und Funktionen aus verschiedenen Klassen zusammenführt

TODO

Statische variablen

Attribute

Statischer symboltable, der immer der Root-SymbolTable ist

Was hat alles ein Symboltable:

Alles was ein Body hat:

* Klassen
* Funktionen (inkl. Entry)
* Loops
* If-Abfragen
* Switch

Eine Funktion hört da auf, wo eine neue anfängt, oder die Datei endet

Die Ids von Variablen sind lokal und jeder symboltable zählt von 0 auf hoch. Ebenfalls gibt die ID die reihenfolge an, wie Datentypen in Objekten gespeichert sind.

Wenn Werte aus Variablen gelesen werden sollen unterscheidet der Parser zwischen lokalen und statischen Variablen und teilt diese auch in zwei verschiedene Nodes ein:

Beide Nodes:

Haben die ID der Variable gespeichert, die gelesen werden soll.

StaticNode

Die ID ist in dem Fall ein Verweis auf den Static-SymbolTable. Somit ist sie eindeutig identifizierbar.

LocalNode

Die ID weist auf ein Attribut in der eigene Klasse

HeapNode

Zusätzlich wird noch eine Objekt-ID gespeichert, welche auf das Objekt zeigt, wessen Variable gelesen werden soll.

Die Normale ID weist nun auf ein Attribut des Objektes.

Beispiel:

cls Starter:

Human h = new Human()

h.name = “Hugo”

Print(h.age)

cls Human:

str name

int age

Zuerst wird ein Objekt der Klasse Human erzeugt, dieses hat Folgende Struktur:

Human

name var str 0

age var int 1

danach wird der Name verändert.

Es wird also eine HeapNode erstellt, die auf das Objekt h zeigt, und die Variable mit der ID 0 in „Hugo“ ändert

Das gleiche Prinzip folgt dann bei der Print Anweisung. Eine HeapNode, welche auf Objekt h zeigt und die Variable mit der ID 1 wiedergibt.

Aus der Sicht des Parsers:

Liest der Parser etwas wie:

Class.attribute

So weiss er direkt bescheid, dass die Variable „attribute“ eine Statische Variable der Klasse „Class“ ist.

Sieht er etwas wie:

Object o

o.name = „Terence“

Merkt er, dass es sich bei o nicht um eine Klasse, sondern um ein Objekt dieser handelt. In dem Fall kann es sich bei der Variable „o“ nur um ein public attribute handeln.

Regel:

Ist der erste Token ein Identifier, welcher den Namen einer Klasse trägt, so muss es sich um ein Statischen Attribut handeln. Andernfalls bleibt nur Option b und es ist ein Attribut eines Objektes.

cls Starter:

Human h = new Human() # NewNode

h.name = “Hugo” # HeapNode (h,0)

Print(h.age) # HeapNode (h,1)

cls Human:

int humans

str name

int age

con ():

name = “default” # LocalNode

humans++ # StaticNode

# Wie und Wann StackFrames erstellt werden

Da sich der Ablauf einer Funktion strukturell nicht ändern kann,

können die StackFrames vorgefertigt werden.

Lexer übersetzt in Tokens

Parser baut AST,SymbolTable

Nachdem der Parser alle Tokens ausgewertet hat,

baut er für jede Funktion/Konstruktor/Entry einen StackFrame

Der StackFrame beinhaltet:

* Alle Variablen, die von der Funktion aus genutzt werden können(Also alle Scopes in der Funktion, sowie alle Attribute der zugehörigen Klasse)
* Eine Returnadresse, die angibt wo der Rückgabewert gespeichert werden soll, falls gegeben

Beispiel:

int add(int a,int b):

->a+b

ntr ():

int result = add(5,2\*3)

add(10,4)

Der StackFrame der Funktion add hat „a“ und „b“ als Variablen gespeichert.

Die Returnadresse wird beim Aufruf der Methode angegeben. In diesem Fall übergibt der StackFrame der Entry, dem StackFrame der aufgerufenen Funktion als Rückgabeadresse die der Variable result.

Anschließend wird dieselbe Funktion nochmals aufgerufen. Diesmal wird aber keine Variable als speicher angeboten. Der StackFrame der Funktion add sieht, dass die Returnadress mit „null“ gekennzeichnet ist verzichtet einfach auf eine Rückgabe.

In beiden Fällen wird nach dem Abarbeiten der StackFrame gelöscht, da die benötigte Arbeit getan ist. Der pointer fällt nun auf den Vorherigen StackFrame zurück (der von der Entry) und arbeitet dort den Code weiter ab. Sobald auch dieser Abgearbeitet ist, wird er ganz normal gelöscht.

Der gesamte Stack ist nun leer und das Programm ist fertig abgelaufen.

Nun werden alle Statischen Variablen/Funktionen gelöscht und der Heap geleert.