Graphite

Projektdokumentation

TDP019 - Projekt: Datorspråk

Dennis Abrikossov, denab905@student.liu.se
Henrik Hårshangen, henha806@student.liu.se

2024-05-07

Innehållsförteckning

Innehåll

1 Inledning

Graphite är ett programmeringsspråk som utvecklats som en del av ett projektarbete i kursen TDP019 Projekt: Datorspråk, under andra terminen på IP-programmet. Projektet är en del av den akademiska utbildningen och syftar till att ge praktisk erfarenhet i design och implementering av programmeringsspråk.

1.1 Syfte

Syftet med detta projektarbete var att specificera och framställa ett programmeringsspråk. Genom att utforma och bygga ett eget språk från grunden, avsåg projektet att fördjupa förståelse för dem teoretiska och praktiska aspekterna av datorspråk. Denna erfarenhet är avsedd att stärka färdigheter i problemlösning och programvarudesign, samtidigt som den ger insikt i komplexiteten och dem överväganden som krävs vid utvecklingen av programmeringsteknologier.

1.2 Introduktion

Det programmeringsspråk som valdes att skapas, Graphite, är inspirerat av både C++ och Python. Där den strikt typade naturen hos C++ uppskattas, som hjälper till att minska antalet buggar genom att kräva explicit deklaration av variabeltyper. Samtidigt beundras Python för dess lättlästa och enkla syntax, vilket gör programmering snabbt och effektivt. Ambitionen var att kombinera dessa egenskaper för att skapa ett nytt språk som erbjuder det bästa av båda världar. Resultatet, Graphite, balanserar strikt typning med enkel syntaktisk struktur, vilket gör det attraktivt för både nya och erfarna programmerare som söker ett robust, men användarvänligt programmeringsspråk.

1.3 Målgrupp

Graphite är främst riktat mot programmerare med en viss tidigare erfarenhet av kodning, speciellt dem som har en förkärlek till strikt typade språk och som uppskattar ordning och struktur i sin kod. Det är idealiskt för dem som vill ha en stark typsäkerhet kombinerat med en klar och koncis syntax. Programmerare från olika bakgrunder, som strävar efter att förbättra sina färdigheter inom mjukvaruutveckling och som vill utforska nya språk, kommer att finna Graphite särskilt användbart.

2 Användarhandledning

I det aktuella kapitlet presenteras en detaljerad genomgång av dem fundamentala konstruktionerna inom programmeringsspråket Graphite. Genom att utnyttja en kombination av teoretiska förklaringar och praktiska exempel, syftar denna text till att förmedla en djupgående förståelse för hur dessa konstruktioner kan tillämpas i komplexa programmeringsmiljöer.

2.1 Datatyper

Graphite hanterar datatyperna olika enligt tabell 1 där deras motsvarigheter och exempel kan observeras.

Tabell 1: Beskrivning av datatyper i Graphite

Datatyp	**Motsvarighet**	**Exempel**
int	Heltal	10
float	Flyttal	10.5
char	UTF-8 karaktär	'a'
bool	Booleanskt uttryck	true *eller* false
array	Behållare av godtycklig datatyp	[1,2,3,4]

Datatypen "int" representerar heltal, som används för att lagra och bearbeta numeriska data utan decimaler. Ett exempelvärde är 10. Samtidigt som "float" används för att representera flyttal med decimaler, vilket är användbart i beräkningar som kräver högre precision, som 10.5. Enskild UTF-8 kodad karaktär representeras "char" , exempelvis 'a'. Denna datatyp används för att hantera text på teckennivå. Boolesk datatypen "bool" kan anta värdena true eller false, vilket används för logiska uttryck och villkorskontroller. Den komplexa datatypen "array" är behållare som kan innehålla flera element av en godtycklig datatyp, vilket demonstreras med exempelvärdet [1,2,3,4]. Denna datatyp kommer referas till som en "lista" under resterande dokumentet. Listor används för att lagra samlingar av värden i en enda variabel och är viktiga för att hantera datamängder effektivt.

2.2 Variabler och variabeltilldelning

I Graphite kan ett godtyckligt värde tilldelas ett namn och lagras som en variabel. Utan att specificera något annat är variabler konstanta i Graphite, vilket innebär att när ett värde har tilldelats variabeln kan den inte förändras. För att kunna ändra en variabels värde måste nyckelordet "mod" användas innan variabeltilldelningen. En konstant variabeltilldening, en modulär tilldening samt en omtilldelning av den modulära variabeln kan se ut på följande vis:

```
int var_a = 10;
mod int var_b = 10;
var_b = 5;
```

Nykelordet auto tillåter språket att dynamiskt interpretera vilken datatyp som ska tilldelas till den relevanta variablen. "auto" tillåter endast dynamisk typhärledning vid variabel deklaration. Detta betyder att en variabel kommer ha en bestämd datatyp efter deklaration och det går inte att tilldela variabeln ett värde av en annan datatyp.

```
auto var_a = 10;
mod auto var_b = 5;
```

Listor i Graphite är en behållare som kan innehålla element av en enda av dem tidigare nämnda datatyperna (int, float, char, bool), men alla element i en specifik lista måste vara av samma datatyp. Nedan visas hur en lista kan deklareras och initialiseras i Graphite:

```
int[] arr_a = [1,2,3];
mod int[] arr_b = [4,5,6];
```

I ovanstående kod anger hakparenteserna efter datatypen att variabeln "arr_a" och "arr_b" är listor som endast innehåller heltal (int). Nyckelordet mod indikerar att listan "arr_b" kan vara modifierbar.

Graphite stöder även skapandet av mer komplexa strukturer där listor kan innehålla andra listor. Detta möjliggör konstruktioner som kan representera flerdimensionella datastrukturer. Ett exempel på detta visas i följande kod:

```
int[] arr_a = [[1],[2,3], 4];
```

Data typer har också basvärden som tillåter deklarationen av variabler utan tilldelning av initialt värde. Nedan visas ett exempel på hur olika datatyper kan deklareras utan explicita initialvärden i programmeringsspråket:

```
int var_a;
float var_c;
bool var_e;
char var_g;
int[] var_a;
```

Värden som dem respektive datatyperna får som basvärde kan observeras i tabell 2 nedan.

Tabell 2: datatypers bas värde

Datatyp	Värde
int	1
float	1.0
char	"a"
bool	true
array	[]

En variabel av typen "int" får automatiskt värdet 1. På samma sätt tilldelas variabler av typen "float" värdet 1.0. Karaktärsvariabler ("char") initialiseras med värdet 'a', och booleska uttryck sätts till "true" som standard. Slutligen representeras en tom "array" med "[]" för att indikera en tom behållare.

2.3 Kontrollstrukturer

Kontrollstrukturer ger möjlighet att skapa ett flöde i programkoden och Graphite tillåter "if"-satser och "while"-satser. Nedan finns ett exempel på hur dessa kan användas:

```
mod int counter = 10;
if(true){
    counter = 0;
}
while(counter < 10){
    counter = counter + 1;
}
```

För båda kontrollstrukturer skrivs den kod som ska köras om dess villkorssats blir uppfyllt mellan klammerparenteserna. Villkorssatserna mellan parenteserna kan vara ett godtyckligt uttryck enligt sektion 2.6.

2.4 Funktioner

Funktioner låter användaren spara ett kodblock och köra denna med en godtycklig mängd parametrar. För att en funktion ska kunna returnera ett värde till funktionsanropet måste nyckelordet "return" följt av vad som ska returneras användas. I Graphite ser en funktionstilldening samt ett funktionsanrop ut på följande sätt:

```
def fun_name(int par_a, char par_b){
    if(par_b == 'a'){
        return par_a;
    }
    return par_b;
}
fun_name(10, 'a');
```

Funktioner kan också definieras med en datatyp, godtycklig datatyp som definierats i tabell 1 kan användas. För att implementera detta, anges datatypen direkt efter nyckelordet "def" och före funktionens

namn i funktionsdeklarationen. Detta garanterar att dem värden som returneras av funktionen är kompatibla med den angivna datatypen, vilket hjälper till att upprätthålla typsäkerhet genom programmet. Ett exempel på hur detta kan göras är implementerat nedan.

```
def int fun_name(int par_a){
    return par_a;
}
fun_name(10);
```

Det är möjligt att definiera funktioner som inte returnerar något värde genom att använda nyckelordet "void" istället för att ange en specifik datatyp. Nedan följer ett exempel på hur en sådan funktion kan definieras och användas:

```
def void fun_name(int par_a){
    print(par_a);
}
fun_name(10);
```

För listor har Graphite inbyggda funktioner som kallas på följande vis:

```
mod int[] arr = [10];
arr.add[5];
arr.remove(1);
arr[0];
```

Funktionen ".add" lägger till ett element i slutet av listan, ".remove" tar bort elementet på den angivna indexpositionen och "[x]" returnerar värdet på elementet med index x. Funktionerna ".add" och ".remove" fungerar endast på listor som är modifierbara.

När en lista innehåller flera listor, även känd som en flerdimensionell lista, kan värden från dem inre listorna enkelt hämtas genom att använda ytterligare hakparenteser efter variabelnamnet. Detta tillåter selektiv åtkomst till element på olika nivåer i datastrukturen. Denna funktionalitet illusteras i exemplet nedan.

```
int[] arr_a = [[1],[2,3], 4];
arr[0][0];
```

Ytterligare en inbyggd funktion är "print" som tar emot ett godtyckligt uttryck och skriver ut dess evaluerade värde i terminalen. För att skriva ut fler uttryck kan symbolen "," för att separera uttrycken och skriva ut dem i följd.

```
print('h','e','l','l','o','_','w','o','r','l','d');
```

Ett uttryck av en godtycklig datatyp kan konverteras till en annan datatyp enligt tabell 3 och kan konverteras på följande vis:

```
to_i('5');
```

Tabellen 3 nedan presenterar en översikt över olika funktioner som är tillgängliga för typkonvertering i det angivna programmeringsspråket. Varje funktion är designad för att omvandla en ingångsvariabel, "x", till en specifik datatyp. Dem accepterade ingångstyperna för varje funktion är listade för att ge en tydlig bild av deras flexibilitet och användningsområde.

Tabell 3: Typkonverteringar

Funktion	**Accepterad datatyp för x**
to_i(x)	int, float, bool, char(om symbolen är en siffra)
to_f(x)	int, float, bool, char(om symbolen är en siffra)
to_b(x)	int, float, bool, char
to_c(x)	int, float, bool, char
to_a(x)	int, float, bool, char

2.5 Omfångshantering

I Graphite hanteras omfång, eller omfång, för att säkerställa att variabler och funktioner endast är tillgängliga inom definierade delar av koden och förhindra oavsiktlig åtkomst eller ändringar av data.

Programkod i Graphite initieras med ett globalt omfång. Detta är det yttersta omfånget där variabler och funktioner som deklareras är tillgängliga inom hela programmet, såvida dem inte överskrids av lokala deklarationer inom funktioner eller kodblock.

Varje gång en funktion anropas i Graphite, skapas ett nytt lokalt omfång. Alla variabler och funktioner som definieras inom denna funktion är endast tillgängliga inom dess kontext. När funktionen avslutas, avslutas också dess omfång, vilket innebär att alla lokalt deklarerade variabler försvinner.

Graphite skapar även omfång inom kontrollstrukturerna "if" och "while". Variabler som skapas inom dessa block är isolerade från resten av programmet och deras livsläng är begränsad till kontrollstrukturens kodblock.

Den sista metoden för att skapa ett omfång är ett fristående omfång, detta kan göras på godtyckligt ställe i programkoden och definieras med fristående klammerparenteser.

Om en variabel med samma namn deklareras i ett inre omfång kommer denna att överskrida eventuella variabler med samma namn i ett yttre omfång. Ett variabelanrop med det namnet kommer kalla på variabeln som är skapad i det inre omfånget så länge omfånget är aktivt. Om en variabel anropas i ett inre omfång vars namn inte har deklarerats som en variabel, kommer Graphite rekursivt söka i omfånget över det nuvarande, tills en variabel med namnet hittats eller skriva ut ett felmeddelande om den inte hittas. Värdet på en variabel i ett högre omfång kan även ändras i ett inre omfång så länge flaggan "mod" har angetts enligt sektion 2.2.

Ett sammanställt exempel på hur Graphite hanterar omfång visas nedan:

```
int var_a_global = 10;
mod int var_b_global = 20;

if (var_a_global < 20){ var_b_global = 30; }
while (var_b_global < 50){ int var_c_local = 10; var_b_global = var_b_global +
  var_c_local }

{ int var_d_standalone_scope = 90; }

def void fun_a( int par_a ){ int var_a_global = 70; print (var_a_global);}
```

I exemplet ovan definieras två globala variabler: "var_a_global" som är en konstant "int" med värdet 10, och "var_b_global" som är en modifierbar "int" med ett värde på 20. Inom "if"-satsen uppdateras "var_b_global" till 30 eftersom villkoret "var_a_global < 20" är sant. Sedan utför "while"-iteratorn upprepade ökningar av "var_b_global" med värdet av en lokalt deklarerad variabel "var_c_local" tills "var_b_global" når 50. En variabel
"var_d_standalone_scope" skapas och försvinner så fort omfånget avslutas. Sist definieras funktionen "fun_a" som lokalt överskrider "var_a_global" med värdet 70 och skriver ut detta.

2.6 Operatorer

Operatorer i Graphite representerar matematiska operationer och definierar vad som händer när två värden evalueras i samma uttryck.

Tabell 4 nedan presenterar dem grundläggande aritmetiska operatorerna som är tillgängliga i Graphite. Varje rad i tabellen ger information om en specifik operator, dess symbol och ett exempel på hur den kan användas i ett matematiskt uttryck.

Tabell 4: Beskrivning av aritmetiska operatorer i Graphite

Namn	**Symbol**	**Exempel**
Addition	+	2 + 2 = 4
Subtraktion	-	2 - 2 = 0
Multiplikation	*	2 * 2 = 4
Division	/	2 / 2 = 1
Potens	**	2**3 = 8
modelus	%	5 % 2 = 1

Addition (+) används för att addera två värden. Exemplet visar att addition av 2 och 2 ger resultatet 4. Subtraktion (-) används för att subtrahera ett värde från ett annat. Exemplet illustrerar att subtraktion av 2 från 2 resulterar i 0. Multiplikation (*) används för att multiplicera två värden. Exemplet visar att multiplikationen av 2 med 2 ger 4. Division (/) används för att dividera ett värde med ett annat. Exemplet visar att division av 2 med 2 ger 1. Potens (**) används för att upphöja ett tal till kraften av ett annat. Exemplet visar att 2 upphöjt till 3 blir 8. Modulus (%) används för att beräkna resten efter division mellan två värden. Exemplet visar att modulus av 5 och 2 är 1.

Tabell 5 nedan ger en översikt över dem jämförelseoperatorer som finns tillgängliga i programmeringsspråket Graphite. Varje rad i tabellen beskriver en specifik operator, dess symbol och ett exempel på användning där variablerna p och q har värdena 1 respektive 2.

Tabell 5: Beskrivning av jämförelse operatorer i Graphite

Namn	**Symbol**	**Exempel där variablerna p = 1 och q = 2**
Mindre än	<	p < q = true
Större än	>	p > q = false
Mindre än eller lika med	<=	p <= q = true
Större än eller lika med	>=	p >= q = false
Skilt från, inte lika med	!=	p != q = true
Lika med	==	p == q = false

Mindre än (<) denna operator jämför om ett värde är mindre än ett annat. I exemplet är p mindre än q, vilket resulterar i "true". Större än (>) testar om ett värde är större än ett annat. Här är p inte större än q, resultatet blir därför "false". Mindre än eller lika med (<=) jämför om ett värde är mindre än eller lika med ett annat. Eftersom p är mindre än q, är påståendet "true". Större än eller lika med (>=) kontrollerar om ett värde är större än eller lika med ett annat. I detta fall är p inte större än eller lika med q, vilket ger "false". Skilt från, inte lika med (!=) bestämmer om två värden är olika. Eftersom p och q är olika, är resultatet "true". Lika med (==) kontrollerar om två värden är lika. Här är p inte lika med q, svaret blir därför "false".

Tabell 6 nedan presenterar dem grundläggande logiska operatorerna som används i programmeringsspråket Graphite. Varje rad beskriver en operator, dess symbol, och ett exempel på användning med dem fördefinierade variablerna p och q, där p är "true" och q är "false".

Tabell 6: Beskrivning av logiska operatorer i Graphite

Namn	**Symbol**	**Exempel där variablerna p = true och q = false**
Konjunktion	and,&&	p and q = false
Disjunktion	or,||	p or q = true
Negation	not,!	!p = false

Konjunktion (and, &&) denna operator används för att bestämma om både p och q är sanna. I detta fall, eftersom q är falskt, är resultatet av p and q också "false". Disjunktion (or, ||) används för att avgöra om någon av operanderna p eller q är sann. Här är p sann, vilket resulterar i att p or q blir "true". Negation (not, !) denna operator inverterar sanningsvärdet av operanden. Eftersom p är "true", blir !p "false".

2.7 Större exempel

I det större exemplet nedan illustreras användningen av konstruktioner nämnda i sektion 2.

```
int radius = 5;
float pi = 3.14;
int rect_length = 10;
int rect_width = 20;

def float calculate_circle_area(int r) {
    return pi * (r * r);
}

def calculate_rectangle_area(int length, int width) {
    return length * width;
}

def void main() {
    float circle_area = calculate_circle_area(radius);
    int rectangle_area = calculate_rectangle_area(rect_length,rect_width);

    print(circle_area);
    print(rectangle_area);

    if (circle_area > rectangle_area) {
        print("B","I","G","G","E","R");
    }
    if(circle_area <= rectangle_area) {
        print("S","M","A","L","L","E","R");
    }

    mod int counter = 0;
    while (counter < 5) {
        print(counter);
        counter = counter + 1;
    }

    float initial = 5.25;
    int int_value = to_i(initial);
    print(initial);
    print(int_value);

    mod int[] number_array = [10, 20, 30];
    number_array.add(40);
    print(number_array);
    number_array.remove(0);
    print(number_array);
    print(number_array[0]);
}

main();
```

3 Systemdokumentation

Graphite är ett imperativt grundspråk byggt på Ruby. Språket använder Rdparse för att analysera programkod och Graphites egna lexer för att bryta ner programmet i dess beståndsdelar. Detta för att återskapa den körabara koden som ett syntaxträd där varje nod representerar en komponent i ett uttryck.

3.1 Språkgrammatik

Språkets grammatik specificeras enligt följande bnf-grammatik:

<program> ::= <statement-list>
<statement-list> ::= <statement> <statement-list> | <statement>
<statement> ::= <return-statement> ;
| <print-statement> ;
| <conversion> ;
| <array-function> ;
| <assignment> ;
| <re-assignment> ;
| <control>
| <function-call> ;
| <function-def>
| <logical-expression> ;
| <variable-call> ;
| <standalone-scope>
<return-statement> ::= return <logical-expression>
<print-statement> ::= print (<variable-list>)
<conversion> ::= to_c (<logical-expression>)
| to_i (<logical-expression>)
| to_f (<logical-expression>)
| to_b (<logical-expression>)
| to_a (<logical-expression>)
<array-function> ::= <variable-call> . add (<variable-list>)
| <variable-call> . remove (<expression>)
| <variable-call> . remove ( )
<assignment> ::= mod <assignment>
| auto <variable> = <logical-expression>
| <array> <variable> = <array-list>
| <array> <variable>
| <type> <variable> = <logical-expression>
| <type> <variable>
<re-assignment> ::= <variable> = <logical-expression>
<control> ::= <if-expression> | <while-expression>
<if-expression> ::= if (<logical-expression>) { <statement-list> }
<while-expression> ::= while (<logical-expression>) { <statement-list> }

<array-list> ::= [<variable-list>] | [ ]
<function-call> ::= <variable> (<variable-list>) | <variable> ( )
<variable-list> ::= <logical-expression> , <variable-list> | <logical-expression>
<function-def> ::= def <function-def-type> <variable> ( ) { <statement-list> }
| def <function-def-type> <variable> (<assignment-list>) { <statement-list> }
| def <variable> ( ) { <statement-list> }
| def <variable> (<assignment-list>) { <statement-list> }
<function-def-type> ::= <type> [] | <type> | void
<assignment-list> ::= <assignment> , <assignment-list> | <assignment>
<logical-expression> ::= <logical-term> <or> <logical-expression> | <logical-term>
<or> ::= || | or
<logical-term> ::= <logical-factor> <and> <logical-term> | <logical-factor>
<and> ::= && | and
<logical-factor> ::= <not> <logical-factor> | <comparison-expression>
<not> ::= ! | not
<comparison-expression> ::= <expression> <comparison-operator> <expression> | <expression>
<comparison-operator> ::= < | > | <= | >= | != | ==
<expression> ::= <term> + <expression>
| <expression> - <term>
| <term>
<term> ::= <factor> * <term>
| <factor> / <term>
| <factor>
<factor> ::= <factor> ** <unary>
| <unary> % <factor>
| <unary>
<unary> ::= - <unary> | <atom>
<atom> ::= (<expression>)
| <function-call>
| <variable-call>
| <array-list>
| <conversion>
| <bool>
| <float>
| <int>
| <text>
<variable-call> ::= <variable> [<expression>]
| <variable>
<text> ::= " <char> " | ' <char> ' | "" | ''
<variable> ::= <variable> <digit> | <letter> <variable> | <letter>
<array> ::= <type> []
<type> ::= int | float | bool | char | auto | void
<float> ::= <int> . <int> | . <int>
<int> ::= <digit> <int> | <digit>
<bool> ::= true | false
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> ::= a ... z | A ... Z | _
<char> ::= any UTF-8 character expect white-space characters

3.2 Systemkomponenter

Graphite är ett omfattande språk som implementerar följande komponenter för att tolka programkod:

3.2.1 Lexikalisk analys

Rdparse inleder processen med den lexikaliska analysen, där den första uppgiften är att omvandla koden till en serie av tokens. Dessa tokens består av tecken-sekvenser som definieras genom reguljära uttryck. Under denna initiala fas läser Rdparse koden och kategoriserar elementen: först identifieras dem mest specifika nyckelorden, följt av siffror, bokstäver, specifika tecken och slutligen alla andra tecken.

Efter att den lexikaliska analysen är slutförd, genereras en lista över alla tokens. Denna lista överförs sedan till själva parsern, som använder den för att fortsätta processen med att tolka och strukturera koden.

3.2.2 Parsing

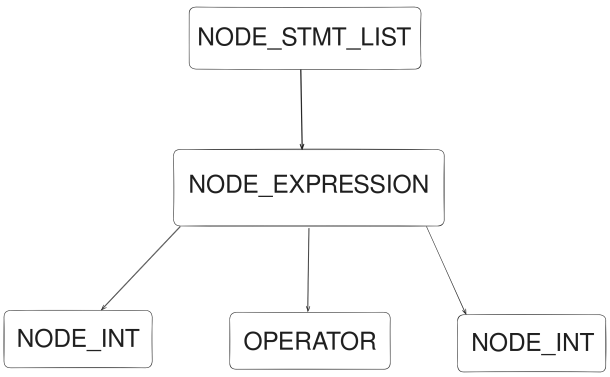
Rdparsern inleder parsningen genom att ta den token-sekvens som genererats under den lexikaliska analysen och matchar dessa tokens mot dem regler som specificerats i grammatiken. Genom att tillämpa dessa grammatikregler parsar Rdparsern koden och konstruerar ett abstrakt syntaxträd, vilket organiserar och representerar koden på ett strukturerat sätt.

3.2.3 Konstruktion av abstrakt syntaxträd

I ett abstrakt syntaxträd representeras varje del av uttrycken i den körda programkoden som enskilda noder. En nod består vanligtvis av två till tre komponenter. Den första av dessa komponenter är funktionen "initialize" som specificerar vilka in-parametrar som noden baserar sin evaluering på, vilket i dem flesta fallen är andra noder som är skapade längre ner i syntaxträdet. Detta kan i ett aritmetiskt uttryck exempelvis vara noderna som representerar högerledet respektive vänsterledet i ett uttryck. Den andra komponenten som dem flesta noder har är funktionen "get_type" som används när andra noder måste fastställa vilken datatyp värdet på den berörda noden har. Detta kan exempelvis vara vid variabeltilldelning, eftersom programspråket är strikt typat och måste säkerställa att det värde som en variabel ska tilldelas är av samma datatyp som variabeln. Den sista komponenten som alla noder har är funktionen "evaluate" som genomför den relevanta processen till det uttryck noden representerar.

När typen har fastställts i parsing-steget skapas en ny instans av den klass som motsvarar detta uttryck. Denna instans returneras sedan rekursivt upp genom dem tidigare reglerna. Dessa regler använder dem returnerade noderna och andra värden för att bygga upp nya noder som parametrar i trädet.

Ett illustrativt exempel på hur Graphite bygger upp ett abstrakt syntaxträd för den simpla aritmetiska uttrycket "1+2" kan observeras i figur 1:



Figur 1: Ett uppbyggt abstrakt syntaxträd som skapas av graphite i kod utrycket "1+2"

3.2.4 Evaluering

När syntaxträdet är konstruerat kallas "evaluate" på noden högst upp i syntaxträdet, som alltid kommer att vara en nod av klassen "Node_statement_list". Detta kommer sedan kalla på "evaluate" för dem uttrycks-noder som är lagrat i denna.

3.3 Klasser och relationer

Klassstrukturen för noderna i Graphite följer punktlistan nedan.

1. **Parser Klass: Language_parser**
   - Parsar en given sträng, skapar- och evaluerar ett abstrakt syntaxträd.
2. **Bas Klass: Node**
   - Fungerar som basklass för alla noder.
   - Har ärvda subklasser och hjälpfunktioner.
3. **Instruktionsklasser (Ärver från Node)**
   - **Node_statement_list**: Representerar en serie instruktioner. Kan hålla en instruktion och eventuellt nästa.
   - **Node_standalone_scope**: Kör en instruktion inom ett nytt omfång.
4. **Variabel- och Datatypklasser (Ärver från Node)**
   - **Node_datatype**: Abstrakt basklass för datatyper.
     - **Node_int**: Lagrar heltal.
     - **Node_float**: Lagrar flyttal.
     - **Node_char**: Lagrar tecken.
     - **Node_bool**: Lagrar boolean-värden.
   - **Node_variable**: Innehåller ett variabelnamn och hämtar dess värde från aktuellt omfång.
   - **Node_assignment**: Definierar en variabeltilldelning.
   - **Node_auto_assignment**: Tilldelar automatiskt en datatyp till en variabel.
   - **Node_re_assignment**: Uppdaterar värdet på en befintlig variabel.
5. **"Array"-klasser (Ärver från Node)**
   - **Node_array**: Representerar en lista av en viss datatyp.
   - **Node_array_accessor**: Åtkomst till ett list-element via index.
   - **Node_array_add**: Lägger till element i en befintlig lista.
   - **Node_array_remove**: Tar bort ett element från en lista med index eller det sista elementet.
6. **Kontrollstruktursklasser (Ärver från Node)**
   - **Node_if**: Villkorssatser.
   - **Node_while**: Itererar över en instruktion medan ett villkor är sant.
7. **Funktionsklasser (Ärver från Node)**
   - **Node_function**: Representerar en funktionsdefinition.
   - **Node_function_call**: Anropar en funktion.
   - **Node_return**: Representerar en retursats.
8. **Uttrycksklasser (Ärver från Node)**
   - **Node_expression**: Basklass för binära operationer.
     - **Node_logical_expression**: Logiska operationer (OCH, ELLER).
     - **Node_comparison_expression**: Jämförelseoperationer ($==$, $<$, $>$).
   - **Node_not**: Representerar den logiska INTE-operationen.
   - **Node_negative**: Representerar negation (unärt minus).
9. **Typkonverteringsklasser (Ärver från Node)**
   - **Node_to_char**: Konverterar värden till tecken.
   - **Node_to_int**: Konverterar värden till heltal.
   - **Node_to_float**: Konverterar värden till flyttal.
   - **Node_to_bool**: Konverterar värden till ett booleanskt uttryck.

3.4 Använda algoritmer

Koden implementerar i klassen med namnet Node_function_call använder caching som optimerar utvärderingen av rekursiva funktionsanrop. Denna teknik är särskilt värdefull i scenarier där rekursiva funktioner annars skulle behöva beräkna samma värden upprepade gånger, vilket kan påskynda beräkningarna betydligt.

När en instans av Node_function_call initialiseras, får den ett namn och en lista av variabelnoder, kallad variable_list. Dessutom inkluderar den ett cache-attribut, en tom hashkarta som fungerar som lagringsplats för resultat från tidigare beräknade funktionsanrop. Detta hjälper till att minska onödiga beräkningar genom att återanvända resultat.

Utöver initialiseringen sker utvärderingen genom flera steg i metoden "Evaluate". Först utvärderas argumenten för funktionsanropet genom att iterera över variable_list. Sedan skapas en unik cache-nyckel baserad på funktionsnamnet och dem utvärderade argumenten. Metoden kontrollerar först i cachen om ett resultat för denna nyckel redan finns. Om det gör det används det cachade resultatet direkt, vilket sparar tid genom att undvika ytterligare beräkningar.

Om resultatet inte redan finns i cachen, identifieras den relevanta funktionen genom dess namn, och ett nytt omfång skapas för att säkert utvärdera funktionens kropp. Om funktionen inte returnerar något, avslutas omfånget och nil returneras. Annars, efter att funktionens kropp har utvärderats, lagras resultatet i cachen för framtida användning och returneras till anroparen.

Denna metodik erbjuder signifikanta prestandafördelar. I rekursiva funktioner, där vissa indata ofta återanvänds, kan caching drastiskt minska antalet beräkningar som behövs. Detta är speciellt användbart i djupa rekursionsträd där samma beräkningar upprepas flera gånger. Ett exempel på detta är en rekursiv funktion som beräknar Fibonacci-tal. Utan caching skulle funktionen behöva beräkna alla tidigare Fibonacci-tal för varje anrop, vilket leder till exponentiell tidskomplexitet. Med caching, däremot, beräknas varje Fibonacci-tal endast en gång, och efterföljande förfrågningar för samma tal kan enkelt hämta resultatet från cachen, vilket reducerar tidskomplexiteten avsevärt.

3.5 Kodstandard

Dem kodstandarder som har tillämpats i projektet är noggrant utformade för att upprätthålla ordning och tydlighet i kodbasen. Först och främst används explicita return-satser, vilket undviker beroendet på implicita returvärden. Varje kodblock är korrekt indenterat enligt dess omfattning, vilket underlättar förståelsen för hur olika koddelar relaterar till varandra. Efter varje end-sats införs en ny rad, vilket bidrar till en tydlig separation av kodsegmenten. Dessutom infogas en ny rad även efter definitioner av klasser och funktioner, vilket hjälper till att visuellt separera dessa viktiga kodstrukturer. När det gäller namngivning av variabler, är dessa både beskrivande och följer snake_case-konventionen, med en begränsning på upp till 16 tecken för att undvika överdrivet långa namn. Slutligen inleds klassnamn med en stor bokstav för att skilja dem från variabler, vilket stärker kodens läsbarhet och struktur.

3.6 Packetering och distribution

Programspråket är byggt på ruby och kräver därför att ruby finns installerat på systemet. I Ubuntu 22.04 kan detta göras genom att köra följande kommando i terminalen

```
$ sudo apt install ruby--full
```

Graphites källkod finns att hämta på GitLab via följande länk: https://gitlab.liu.se/henha806/TDP019 eller genom följande kommandon i terminalen(förutsatt att Git är installerat på systemet):

```
$ git clone git@gitlab.liu.se:henha806/TDP019.git
```

3.6.1 Projektets filstruktur

Projektets filstruktur följer strukturen enligt tabellen nedan

```
.
├── docs
│   └── språkspecifikation_bnf_denab905_henha806.md
├── lan
│   ├── graphite.rb
│   ├── language.rb
│   ├── language_parser_test.rb
│   ├── node.rb
│   ├── rdparse.rb
│   ├── test2.gph
│   ├── test_fib.gph
│   └── test_program.gph
└── readme.md
```

3.6.2 Köra kod från fil

För att köra språket från en fil måste programkoden sparas i en ".gph" fil och körs därefter i foldern ./lan med kommandot:

```
$ ruby graphite.rb filnamn.gph
```

3.6.3 Köra programspråket i interaktivt läge

För att köra språket i interaktivt läge körs följande kod i ./lan:

```
$ ruby graphite.rb gph
```

För att avsluta interaktivt läge kan någon av följande kommandon köras:

‘quit‘, ‘exit‘, ‘bye‘, ‘done‘

4 Erfarenheter och reflektion

Under vårt projekt har vi genomgått en betydande utvecklingsresa. Projektet började med entusiasm och en stark vision om att skapa en imperativ klon som blandar element från både C++ och Python. Vår initiala specifikation var ambitiös, och under projektets gång har vi stött på både förväntade och oväntade utmaningar.

4.1 Vad gick lättare än förväntat?

Samtliga av projektets moment gick snabbare än initialt planerat, vilket var positivt därför att det gav utrymme för oväntad tidsförlust av skäl utanför projektet. En bidragande faktor till detta var saknaden av någon referens till hur lång utvecklingstid varje del av projektet skulle kräva. Ytterligare ett moment som gick lättare än förväntat var implementeringen av dem konstruktioner som krävdes för betyg 5. Detta var för att en robust typhantering utvecklades under projektets implementering och medförde att inget behövde skrivas om utanför implementeringen av "Node_auto_assignment"-klassen. Typhärledningen gynnades även av detta och var lätt att implementera.

4.2 Vad var svårare än förväntat?

Vid flera tillfällen under implementeringen av programspråket dök problem upp som krävde större omarbetning av koden. Den första av dessa var när omfång skulle implementeras. Initialt var tanken att parsern skulle skapa omfång genom att skapa nya instanser av en "Scope"-klass. Detta visade sig inte vara lämpligt med funktionalitet hos Rdparse och projektets krav om att skapa ett abstrakt syntaxträd. Detta låg till grund föra att koden omarbetades från start, vilket däremot ledde till en bättre kodstruktur.

Det andra problem som uppstod var returer inom funktioner och kontrollstrukturer. Problemet grundade sig i att få nyckelordet retur att bryta exekveringen av resterande kodblocket och returnera rätt värden till det föregående omfånget. Detta problem var svårare än förväntat och krävde en större tidsåtgång än önskat.

Allt eftersom programspråket fick funktionalitet utökas mängden enhetstester. Tidsåtgången för att genomföra enhetstesterna ökade fortare än förväntat och ett problem i dem grammatiska reglerna misstänkes vara orsaken till detta. Efter många tester fastställdes att regeln för "unary" låg på fel plats och orsakade överflödig bakåtspårning. Ännu ett problem relaterat till optimeringen av "parse"-momentet var vid djupt nästlade listor innuti listor. Detta problem kvarstår fortfarande.

Vidare problem kring listor uppstod när beslutet togs om att dessa skulle fungera i godtyckligt aritmetiskt- och logiskt uttryck. Implementeringen av detta var krånglig och krävde omfattande enhetstester för att säkerställa att alla tänkbara kombinationer fungerade som förväntat. Tillgång till värden i listor som var nästlade i andra listor var även detta svårt att implementera.

4.3 Ändringar relativt den första planen

Initialt skulle Graphite hantera "for"-iteratorn som repetetiv kontrollstruktur, men ersattes av "while"-iteratorn eftersom denna var lättare att implementera samt att den kunde återskapa alla tänkbara iteratorer.

Ytterligare en förändring som gjordes var att implementeringen av strängar valdes att inte utföras. Detta för att endast en datatyps-behållare krävdes och denna konstruktion fick därför en låg prioritet i projektimplementeringen.

Ännu en ändring som gjordes var till hur funktioner skulle definieras. Ursprungligen skulle ingen returtyp specificeras i funktionsdefinitionen. Detta ändrades efter att svårigheter kring implementeringen av denna funktion uppstod, och funktionsdefinitioner krävde därav att en returtyp specificeras. När den automatiska typhärledningen var implementerad gick den ursprungliga idéen att uppnå utan besvär och ett val gjordes om att det skulle vara frivilligt om en returtyp ska specificeras eller inte.

Den sista ändringen som skedde var att "else"- och "else-if"-satser valdes att inte implementeras eftersom dess funktionalitet inte var betydelsefull för betygskriterierna.

4.4 Avslutande reflektioner

Slutligen har detta projekt inte bara varit en teknisk utmaning utan även en värdefull läroprocess. Vi har konfronterats med betydande svårigheter, från djupgående problem med omfång och funktioners returer till optimeringar för prestanda. Varje utmaning har däremot bidragit till en större förståelse för språkdesign och systemutveckling.

Projektets dynamiska sida krävde en ständigt anpassning och omvärderade prioriteringarna, vilket i sin tur har skärpt våra färdigheter i problemlösning inom projektarbete. Våra initiala planer omfattade ambitioner som justerades under resans gång för att bättre passa våra utbildningsmål och dem praktiska begränsningarna vi stötte på.

Genom detta projekt har vi fått en omfattande förståelse för hur programspråk är strukturerade, vilket kommer att vara till stor hjälp i framtiden. Med denna kunskap kommer vi att kunna lära oss och tillämpa nya programspråk mycket snabbare, eftersom vi redan har en solid grund i att förstå deras generella uppbyggnad.