PROMETHEUS

Ett typat, objektorienterat programmeringsspråk av Tim Andersson.

Linköpings universitet
TDP019 Projekt: Datorspråk
Examinator: Anders Haraldsson, IDA

Tim Andersson
timan976@student.liu.se
Utskriftsdatum: 2012-05-29

INNEHÅLLSFÖRTECKNING

Inledning 1

Användarhandledning 1

Installation 1

Att använda Prometheus 1

Kodkonstruktioner 1

Datatyper/Tilldelning 1

Funktioner och metoder 2

Block 2

Klasser (Ej implementerat) 3

Properties (Ej implementerat) 4

Villkorsstrukturer 4

Upprepningsstrukturer 5

Operatorer 5

Systembeskrivning 6

Lexikalisk analys och parsning 6

Abstrakt syntaxträd 6

Runtime – Typkoll 7

Runtime – Metod- och funktionsanrop 7

Runtime - Scope 7

Erfarenheter och reflektion 8

Tokenizern 8

Ruby 9

PROMETHEUS

Inledning

Prometheus är tänkt att vara ett objektorienterat programspråk med det bästa (enligt mig) från C, Objective-C, Ruby och Python. Några exempel på dessa saker är stark typning från (Objective-)C, block från Ruby/Objective-C och literaler för arrayer och dictionaries från Ruby och Python. Den huvudsakliga målgruppen är programmerare som har en del tidigare erfarenhet.

Användarhandledning

Om du är intresserad av att prova på Prometheus bör du följa denna användarhandledning. Först tas installation av Prometheus upp och sedan användningen av själva språket.

Installation

För att använda Prometheus måste du först ladda ner källkoden för Prometheus och sedan se till att du har senaste versionen av Ruby installerad (vilken kan laddas ner här). När du har gjort detta skall du öppna en terminal och navigera till */path/to/prometheus_root/src/*. Väl inne i mappen kan du starta en interaktiv Prometheus-tolk genom att skriva *ruby PrometheusParser.rb*, eller köra en fil genom att skriva *ruby PrometheusParser.rb filename.pro*.

Att använda Prometheus

När du har installerat Prometheus har det blivit dags att börja koda! Om du har någon tidigare erfarenhet av C, Python eller Ruby kommer Prometheus förhoppningsvis kännas någorlunda bekant. Det som kan vara lite jobbigt om man inte är van vid det är den statiska typningen, men det vänjer man sig vid tämligen fort. Skapa en testfil (exempelvis *playground.pro*) och testa att köra konstruktionerna som nämns under nästa rubrik.

Kodkonstruktioner

Datatyper/Tilldelning

Allting i Prometheus är objekt, även saker som är skalära i andra språk. Nedan följer en lista över datatyper och hur man skapar dem i Prometheus.

```
Integer count = 3;
Float pi_approx = 3.141592;
String name = “Tim”;
Object foo = 0; // Object är basklassen för alla objekt
Bool count = true;
Array nums = [0, 0, 7];
Dict map = @[“one”: 1, “two”: 2, “three”: 3];
Block callback = ^(Bool completed) { /* Do something */ }
```

Funktioner och metoder

I Prometheus skiljer man på procedurer som tillhör objekt (*metoder*) och procedurer som inte gör det (*funktioner*). En funktion kan deklareras var som helst och tillhör inte ett objekt:

```
Number add(Number n1, Number n2);
```

Metoder kan endast deklareras i en klassdeklaration och skiljer sig lite från funktioner syntaktiskt sett:

```
- Number add(Number n);
+ Object new();
```

Ett minus (-) framför en metod visar att det är en instansmetod, medan ett plus (+) visar att det är en klassmetod.

Anrop

Funktioner och metoder anropas precis som man hade förväntat sig:

```
Number sum = add(3, 6);
Number five = 5;
five.add(10); // I praktiken samma som five + 10
```

Parenteserna måste alltid vara med, även om funktionen eller metoden inte tar några argument.

Block

Prometheus har som många andra språk stöd för anonyma funktioner. I Prometheus kallas dessa *block* och är objekt precis som allting annat. Hur man använder block följer nedan.

```
// Returnerar [0, 0, 49]
[0, 0, 7].map(^(Number n) {
     return n.exp(2);
});

// Det går även att spara blocket för att använda det vid ett
// senare tillfälle:
Block double = ^(Number n) { return n * 2; }
[1, 2, 3].map(double); // Returnerar [2, 4, 6]
double(4); // Returnerar 8
```

Klasser (Ej implementerat)

Prometheus har även stöd för att skapa nya klasser. I likhet med Objective-C måste man först skapa ett så kallat *interface* för klassen och sedan *implementera* den. Ett exempel på en klass ser vi här:

```
// Klassens interface
@interface Person < Object {
     // Instansvariabler
     Number age;
     String name;
}

// Metoddeklarationer
- Person init(Number a, String n);
- Void set_age(Number a);
- Void set_name(String n);
- Number age();
- String name();

@end
```

```
// Klassens implementering
@implementation Person

- Person init(Number a, String n) {
     self.age = a; // Syntaktiskt socker för self.set_age(a);
     self.name = n; // Syntaktiskt socker för self.set_name(n);
     return self;
}

// Setters
- Void set_age(Number a) {
     age = a;
}

- Void set_name(String n) {
     name = n;
}

// Getters
- Number age() {
     return age;
}

- String name() {
     return name;
}

@end
```

Properties (Ej implementerat)

Som du kanske såg blev det rätt mycket boilerplate-kod för getters och setters i klassen. Prometheus kan skapa getters/setters automatiskt åt en genom användning av @*property* i interfacet av en klass:

```
@interface Person < Object {
      Number age;
      String name;
}

// Properties
@property (readonly) name; // skapar name()
@property age; // readwrite – skapar set_age() och age()

- Person init(Number a, String n);

@end
```

Villkorsstrukturer

Villkorsstrukturerna i Prometheus ser precis ut som de gör i C:

```
Integer count = 3;
if(count < 3) {
      print "count är mindre än 3!";
} else if(count > 3) {
      print "count är större än 3!";
} else {
      print "count är lika med 3!";
}

// Det går även att utelämna måsvingarna…
if(count == 3)
      print "count är lika med 3!";
else
      print "count är inte lika med 3!";

// …eller att skriva det på en rad
if(count == 3) print "count är lika med 3!";

// Prometheus har även stöd för "ternary operator" (ej implementerad)
String result = count == 3 ? "är lika med 3" : "är inte lika med 3";
```

Upprepningsstrukturer

Prometheus har stöd för *while-* och *for-loopar*. Dessa ser till stor del ut som i C, men det finns en skillnad i for-loopen:

```
// En klassisk for-loop
for(Integer i = 0; i < 10; i++) {
      print “<i>”; // 0, 1, 2, 3…
}

// En klassisk while-loop
Integer i = 0;
while(i < 10) {
      print “<i>”; // 0, 1, 2, 3…
      i++;
}

// Iterera genom en samling (ej implementerad än)
for(Integer i in [0, 1, 2, 3, 4]) {
      print “<i>”; // 0, 1, 2, 3…
}
```

Operatorer

Här följer en lista på operatorerna som finns i Prometheus. Tabellen är sorterad i fallande ordning efter prioritet.

Operator	Beskrivning
++ --	Ökning och minskning (suffix)
++ --	Ökning och minskning (prefix)
! -	Logisk NOT Unär minus (ej implementerad)
* / %	Multiplikation, division och modulo
+ -	Addition och subtraktion
< <= > >=	Relationsoperatorerna < och ≤ Relationsoperatorerna > och ≥
== !=	Jämförelseoperatorerna = och ≠
&&	Logisk AND
||	Logisk OR

Operator	Beskrivning
= += -= *= /= %=	Tilldelning Tilldelning via addition och subtraktion (ej imp.) Tilldelning via multiplikation, division och modulo (ej imp.)

Systembeskrivning

Lexikalisk analys och parsning

Prometheus är implementerat med hjälp utav den recursive descent parser (RDParser) som vi tillhandahöll genom TDP007-kursen.
Det första som sker är den lexikaliska analysen där parsern skapar tokens; dessa tokens är antingen strängar eller instanser av min `Node`-klass. Jag har valt att göra så att det bildas separata tokens av följande:

- Operatorer
- Nyckelorden *true* och *false*
- Strängar
- Variabelnamn
- Klassnamn
- Heltal och flyttal

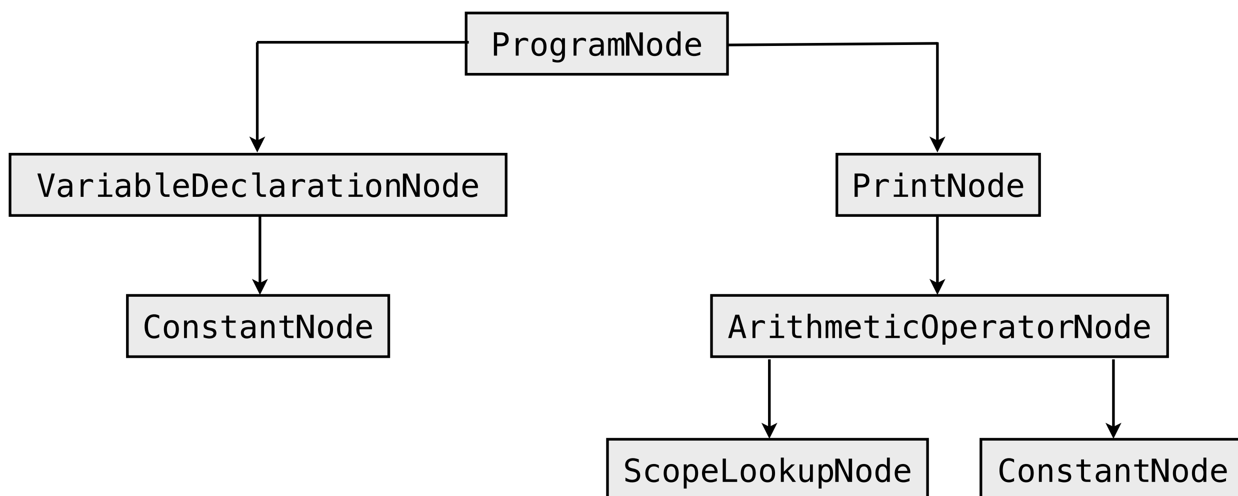
När den lexikaliska analysen är klar och det har skapats tokens påbörjas parsningen. Parsern använder sig utav reglerna i språkets grammatik (se bilaga 1), vilken är skriven i ett BNF-liknande format och uttrycker vilka sammansättningar av tokens som är giltiga i språket.

Abstrakt syntaxträd

För varje konstruktion i språket skapas det en motsvarande nod[1], vilket bildar ett så kallat *abstrakt syntaxträd* (AST). Exempelvis skapas det en instans av `ConstantNode` när flyttal, heltal, booleska värden eller strängar hittas. När det abstrakta syntaxträdet har konstruerats kommer man ha ett träd där rotnoden är en instans av `ProgramNode`. Nästa steg som sker är att `evaluate`-metoden (som alla `Node`-subklasser implementerar) anropas på rotnoden, vilket kommer resultera i att alla undernoder evalueras i tur och ordning. Nedan följer ett exempel på en bit kod och dess motsvarande AST.

```
Number foo = 3;
print foo + 10;
```

[1] En instans av en `Node`-subklass



Runtime – Typkoll

Då Prometheus är ett starkt typat språk som är implementerat i ett dynamiskt typat språk (Ruby) var jag tvungen att implementera ett eget typsystem. Detta förverkligas genom att runtimen håller reda på typen på variabler, argument och returvärden. Varje gång man tilldelar en variabel, försöker anropa en funktion eller liknande kontrolleras typen på alla inblandade objekt; om ett objekt inte är av den förväntade typen kastas en exception.

Runtime – Metod- och funktionsanrop

Runtimens andra uppgift är att anropa metoder på objekt baserat på metodsignaturer, samt att anropa funktioner.
När en funktion deklareras i Prometheus skapas en instans av `NAFunction`, vars syfte är att hålla reda på funktionens namn, returvärde, parametrar och body. `NAFunction` implementerar även en `call`-metod som kallas när ett funktionsanrop hittas.

Metoder och metodanrop skiljer sig lite från funktioner och funktionsanrop. En väsentlig skillnad mellan en metod och en funktion är att en metod anropas på ett objekt – en mottagare. I runtimen finns en funktion, `msg_send()`, som tar en mottagare, en metodsignatur och ett godtyckligt antal argument som skall användas vid metodanropet. En metodsignatur håller reda på metodens namn, returtyp, typerna på parametrarna, samt om det är en klass- eller instansmetod. Tillsammans med den specificerade metodsignaturen och metodtabellerna för varje klass kontrollerar `msg_send()` om mottagaren implementerar metoden. Gör mottagaren det kontrolleras det att de angivna argumenten är av rätt typ och rätt antal; stämmer allt överens anropas metoden på mottagaren.

Runtime - Scope

Den tredje uppgiften runtimen har är att sköta hantering utav scope. Detta görs genom `NAScopeFrame`-klassen, vilken representerar en räckvidds*ram*. Varje ram har en hash där nycklarna är namn på variabler eller funktioner, och värdena är motsvarande variabel (`NAVariable`) eller funktion (`NAFunction`). Varje ram har även en *parent*-ram, vilket innebär att en räckvidd (scope) är en hierarki av räckviddsramar. När en variabel eller funktion skall hittas i ett scope börjar man längst ner i hierarkin och traverserar sedan uppåt.

För att kod alltid skall evalueras i rätt scope skickas den aktuella räckviddsramen med som argument till `evaluate`-metoden på noderna.

Erfarenheter och reflektion

Att designa och implementera ett eget programmeringsspråk har varit en oerhört rolig och lärorik upplevelse. Dels får man en djupare förståelse för hur (vissa) programmeringsspråk är uppbyggda, och dels får man bättre grepp om vad det är som gör att man föredrar vissa programmeringsspråk framför andra.
I slutändan blev jag nöjd med Prometheus, trots att jag inte implementerade riktigt allt jag hade hoppats på. Tanken var att det skulle vara ett starkt typat objektorienterat programmeringsspråk; att Prometheus är detta går inte att förneka. Jag hoppas dock att jag kommer ha tid i framtiden att implementera några saker jag känner att det saknar.

För att runda av följer här nedan ett par problem och reflektioner.

Tokenizern

Ett av de mest frustrerande problemen jag stötte på hade med tokenizern att göra. Problemet uppstod om man hade regler som liknade följande…

```
token(/foo/) { |s| s }
token(/d+/) { |d| d.to_i }
```

… och försökte tokenize:a exempelvis:

```
“123foo”
```

Det förväntade resultatet är att både *123* och *foo* blir tokens, men i praktiken blir *foo* en token medan *123* hoppas över. Anledningen till detta är att tokenizern fungerar på följande sätt:

1. Undersök om det reguljära uttrycket matchar strängen.
2. Om det reguljära uttrycket matchar:
    1. Plocka ut delsträngen som matchade uttrycket.
    2. Ta bort allting från början av strängen upp till slutet av delsträngen.

Det tog ett tag innan jag konstaterade vad det var som orsakade problemet, men när jag väl hade gjort det var lösningen enkel – se till att alla reguljära uttryck matchar början av strängen:

```
token(/^(foo)/) { |s| s }
token(/^(d+)/) { |d| d.to_i }
```

Ruby

Prometheus är, som tidigare nämnt, implementerat i det dynamiskt typade programmeringsspråket Ruby. Överlag tycker jag att Ruby är ett väldigt smidigt och trevligt språk, men det finns en sak jag har lärt mig att avsky under projektets gång – dynamisk typning. Att inte kunna se direkt vilka typer det skall vara på argumenten till en funktion är något jag har blivit smått frustrerad på vid ett antal tillfällen.