LINGUAGGI DINAMICI E PYTHON

0. Introduzione

Questo documento propone nozioni fondamentali per imparare a conoscere il linguaggio di programmazione Python. Attualmente la crescita riguardante l'utilizzo di questo linguaggio è notevole, in quanto ha superato il tanto utilizzato Java. Spesso Python lo si mette in competizione a Java in quanto sono due linguaggi di programmazione ad oggetti.
È importante conoscere Python nel 2020 in quanto è molto utilizzato dalle grandi aziende in ambito Web, e inoltre è il più utilizzato e più famoso in ambito di "Intelligenza Artificiale".
Il documento viene fornito in modalità gratuita e non è possibile effettuare qualsiasi attività economica di lucro su di esso.

Questo testo è scritto da Daniel Rossi, studente di ingegneria informatica presso il dipartimento di ingegneria dell'università di Modena e Reggio emilia.

Contatti:

Instagram: @DanielrossiTV / @OfficialProjecto

YouTube: Https://youtube.com/c/ProjectoOfficial?sub_confirmation=1

Al link del canale YouTube potete trovare videocorsi e progetti riguardanti il linguaggio di programmazione Python.

1. Caratteristiche e differenze dagli altri linguaggi

Innanzi tutto Python è un linguaggio dinamico a differenza ad esempio del C che è un linguaggio statico:

- *Linguaggio statico:* è un linguaggio ad alto livello in cui le operazioni effettuate a run-time sono legate quasi esclusivamente all'esecuzione del codice del programmatore (un'eccezione notevole è la gestione dello stack). Alcuni esempi possono essere il linguaggio C e il linguaggio Assembly;
- *Linguaggio dinamico:* è un linguaggio di alto livello in cui le operazioni effettuate a run-time non sono legate esclusivamente all'esecuzione del codice. Alcuni

linguaggi dinamici sono il Perl, Python (come stiamo dicendo), Ruby, PHP, Javascript.

In realtà non esiste una definizione univoca che differenzia un linguaggio statico da uno dinamico, nonostante ciò si possono elencare le più importanti differenze:

- Tipizzazione dei dati (attribuire ad una variabile un tipo, es float, int… viene svolta a run-time)
- Metaprogramming (svolta a run-time)
- Gestione dinamica della memoria e degli errori (svolta a run-time)
- Modello di generazione del codice (prodotto intermedio es. bytecode)

Alcune di queste caratteristiche si possono ritrovare in altri linguaggi non propriamente considerati dinamici, ad esempio la gestione della memoria in Java.

Caratteristiche di un linguaggio statico

Esiste una fase detta *"di compilazione"*, in cui il codice sorgente viene tradotto in un formato esclusivo per l'architettura considerata.
La traduzione da codice sorgente a codice macchina è 1:1, questo significa che è una rappresentazione molto fedele.
Permette l'esecuzione ad una velocità elevata.
I tipi di dati sono identificati a tempo di compilazione (non quando il programma è in esecuzione come accade in python) e non sono mutabili a tempo di esecuzione.
Non sempre fornisce strumenti di controllo, né semplificazioni; quasi tutto è lasciato al programmatore (memoria, tipizzazione). Basti pensare alla funzione malloc del C per allocare memoria a un puntatore.

Caratteristiche di un linguaggio dinamico

Nella fase di compilazione il codice sorgente viene tradotto in un formato intermedio, indipendente dall'architettura (es. bytecode)

[Bytecode: è più astratto che si pone in mezzo tra il nostro linguaggio di programmazione e il vero linguaggio macchina, e viene usato per descrivere le operazioni che costituiscono un programma. Viene chiamato così perché spesso le operazioni hanno un codice che occupa un solo Byte, anche se la lunghezza dell'istruzione può variare perché ogni operazione ha un numero specifico di parametri sui quali operare. Questo linguaggio viene utilizzato per creare indipendenza dall'hardware e facilita la creazione di interpreti]

Il formato intermedio è interpretato. Questo significa che un linguaggio dinamico è un linguaggio portatile, che funziona su qualsiasi macchina.

L’interprete sfrutta funzioni interne per gestire la memoria e gli errori in modo automatico a run-time.
Ha una tipizzazione dinamica dei dati: il tipo di una variabile (ma anche quello di una funzione) può cambiare a run-time.

Inoltre include meccanismi che permettono al programma di “analizzarsi” e modificarsi durante l’esecuzione (metaprogramming). Il Meta-Programming dà la possibilità di costruire funzioni e classi il cui obiettivo principale è la manipolazione del codice:

- Permette di *generare* nuovo codice
- Effettua *modifiche* e *wrapping* di codice esistente

[Wrapping: alcune funzioni svolte dal wrapper sono: allocare e deallocare risorse, controllare le pre-condizioni e post-condizioni, memorizzazione nella cache / riciclaggio a seguito di un calcolo lento]

Alcuni meccanismi fondamentali per il meta-programming sono:

- Decoratori
- Metaclassi
- Decoratori di Metaclassi

Altre caratteristiche

Dà la possibilità di creare strutture di dati, anche eterogenee, variabili nel tempo.
Inoltre esistono tante librerie esterne e facilmente utilizzabili per diversi compiti:

- Calcolo scientifico
- Interfacce grafiche complesse
- Supporto per il Web

Linguaggio dinamico VS statico

Come *VANTAGGI* abbiamo in primis che è più semplice da imparare per anche i neofiti. La scrittura del codice, di un software, diventa la scrittura effettiva del suo scheletro. Esistono tante librerie esterne e quindi si ha un forte riutilizzo del codice. La prototipazione, ovvero creare modelli di sviluppo del software, è veloce. Infine è *flessibile* e *portatile*.

Come *SVANTAGGI* possiamo notare la lentezza (a causa delle molte operazioni a run-time). Per via della semplicità di scrivere codice, può portare a scrivere pessimo codice (soprattutto per i principianti); è da sottolineare che python ha proprie metodologie di programmazione.

2. Tipizzazione dei dati

Come già sappiamo, ogni dato/variabile ha un tipo (ad es. float, int, double …). In un programma, ad ogni entità è associata un informazione su:

- *Valore*: valore (attuale, calcolato, …)
- *Tipo*: indicatore del tipo di dato associato
- *Dimensione*: l'area di memoria occupata

In Python tutte le variabili sono oggetti, esiste la classe Int, la classe Float, ecc… .
Il tipo di dato ha conseguenze sull'insieme di valori che un entità può assumere (valori ammissibili), la sua dimensione in memoria e e le operazioni che su tali valori si possono effettuare.

Dall'inglese: type checking, la tipizzazione è un processo che cerca di 'capire' qual è il tipo di un dato, sia esso dichiarato o prodotto/calcolato, in modo da poter effettuare i relativi controlli, e quindi

- definire i vincoli sui valori ammissibili (non puoi associare 3,14 a una variabile Int)
- definire le operazioni consentite (non puoi sommare 3,14 a una variabile Int)

Queste operazioni garantiscono che un programma sia type-safe.

Tutti i linguaggi di programmazione di alto livello hanno un proprio sistema per la tipizzazione dei dati (ma con significative differenze).

Che vantaggi porta la tipizzazione?

Ha conseguenze sulla *SICUREZZA*, quindi l'uso della tipizzazione permette di trovare codice privo di senso o illecito.
A livello di compilazione riduce i rischi di errori o risultati inattesi a run-time, ad esempio consideriamo:

variabile = 3 / "Hello World"

il primo operando è un intero, il secondo è l'operatore di divisione, e il terzo è una stringa. È un'operazione che non ha senso e quindi viene riportato un "*type error*".

Ha conseguenze sull'*OTTIMIZZAZIONE*, quindi in uno stadio iniziale (a livello di compilazione), le informazioni di type checking possono fornire informazioni utili per applicare tecniche di ottimizzazione del codice.

Es. istruzione x * 2 può essere ottimizzata nel seguente modo per produrre uno shift più efficiente:

mul x,2 → shift_left x

Questo è possibile solo se si sa con certezza che x rappresenta un valore di tipo intero.

Ha conseguenze sull'*ASTRAZIONE* dove il meccanismo dei tipi di dato permette di riprodurre programmi ad un livello di astrazione più alto di quello di linguaggi a basso livello. Un caso limite di assenza di tipizzazione è un codice macchina nativo, quindi sequenze di bit o byte che rappresentano un unico tipo.

Ad esempio una stringa di caratteri è vista a basso livello come una sequenza di byte. In Python una stringa è un oggetto che ha metodi e attributi.

Ha conseguenze sulla *DOCUMENTAZIONE* e quindi nei sistemi di tipizzazione più espressivi, i nomi dei tipi di dato possono servire come fonte di documentazione del codice. Il tipo di dato mostra la natura di una variabile, ed in ultima analisi l'intento del programmatore.

[siccome in Pyhon esistono le "variabili senza tipo" può essere un problema utilizzarle in quanto chi va a rileggere il codice può non capire cosa contengono e di che tipo sono]

Ad esempio i Booleani, Timestamp o marcatore temporale (solitamente interi a 21/64 bit)
Dal punto di vista della comprensione del codice è diverso leggere:
int a = 32; o timestamp a = 32

è inoltre possibile la definizione di nuovi tipi di dato (typedef in C) per aumentare l'espressività del linguaggio.

Ha conseguenza sulla *MODULARITÀ* sull'uso appropriato dei tipi di dato che costituisce uno strumento semplice e robusto per definire interfacce di programmazione (API = Application Program Interface). Le funzionalità di una data API sono descritte dalle signature (ovvero i prototipi) delle funzioni pubbliche che la compongono. Leggendo le signature, il programmatore si fa immediatamente un'idea di cosa può o non può fare!
Boolean somma(int a, int b)
somma(a,b)

Quando avviene il Type Checking?

Le operazioni di type checking che stabiliscono il tipo del dato associato a porzioni di codice e ne verificano i vincoli (range di valori ammissibili e operazioni consentite) possono avvenire:

1. a tempo di compilazione (compile time)
2. a tempo di esecuzione (run-time)

Classificazione non esclusiva: sono possibili anche metodi intermedi.

Type Checking statico

Il meccanismo di type checking è *statico* se le operazioni di type checking sono eseguite **solo** a tempo di compilazione!

I linguaggi che effettuano type checking statico sono:

C , C++ , Fortran , Pascal , GO

Il type checking statico permette:

- di individuare parecchi errori con largo anticipo (inteso come una forma più sicura di verifica dell'integrità un programma)
- migliori prestazioni (ottimizzazione e mancanza di controlli a run-time)

Type Checking dinamico

Il type checking è dinamico se la maggior parte delle operazioni (non per forza tutte) di type checking vengono effettuate a tempo di esecuzione. Questo implica che non è necessario esplicitare il tipo di una variabile, ci penserà l'interprete a tempo di esecuzione ad associare alla variabile il tipo corretto in base al dato che le stiamo assegnando.

Linguaggi a type checking dinamico:
Javascript , Perl , PHP , Python , Ruby

Questa dinamicità permette più flessibilità rispetto al modello statico, con maggior overhead (più lavoro da parte dell'interprete nel gestire le risorse)

- le variabili possono cambiare tipo a run-time
- gestione della struttura dei dati a run-time (problemi di gestione della memoria, processi, tempo di esecuzione …)

Tramite il meccanismo di tipizzazione a run-time, si possono effettuare assegnamenti "arditi", ad esempio: var = <token letto da input>

dove token può avere un qualunque tipo.
Il type checker dinamico assegna il tipo corretto a run time, questo ovviamente non è possibile in linguaggi come il C o Java.

Nonostante ciò, tutta questa flessibilità ha un prezzo. Un type checker dinamico dà minori garanzie a priori perché opera (la maggior parte del tempo) a run-time, quindi è molto possibile incorrere in tante situazioni di errore a causa di input inaspettati.
Sarà quindi necessario:

- gestire le situazioni di errore a runtime con un meccanismo di gestione delle eccezioni (try: / except: … import traceback as t, t.print_stack())
- verificare la correttezza di un programma effettuando tanti test

Type checking ibrido

Alcuni linguaggi fanno un uso ibrido di tecniche di type checking statico e dinamico. In Java il type checking è completamente statico sui tipi ma dinamico per le operazioni di binding tra metodi e classi (legato al polimorfismo nella identificazione di una signature valida nella gerarchia delle classi).
Nonostante ciò, Java viene ritenuto lo stesso a type checking statico.

Vantantaggi a confronto

Type checking statico:

- identifica gli errori a tempo di compilazione (più veloce perché ci sono meno controlli)
- permette la costruzione di codice che esegue più velocemente

Type checking dinamico:

- permette una prototipazione più libera e rapida
- è più flessibile perché permette l'uso di costrutti considerati illegali nei linguaggi statici (Es. var = 5)
- permette la generazione di nuovo codice a runtime (metaprogrammi – es. funzione eval())

Tipizzazione forte

Un linguaggio di programmazione si dice di tipizzazione forte se impone regole rigide e impedisce usi incoerenti dei tipi di dato specificati (ad esempio le operazioni effettuate con operandi aventi tipo di dato non corretto).

Alcuni esempi possono essere la somma di interi con caratteri o un'operazione in cui l'indice di un array supera la sua dimensione.
Un linguaggio completamente a tipizzazione forte non esiste, sarebbe inutilizzabile.

Tipizzazione debole

D'altra parte esiste la tipizzazione debole, ovvero, quando un adotta la tipizzazione debole dei dati non impedisce operazioni incongruenti (ad esempi con operanti aventi tipi di dato non corretto). La tipizzazione debole fa spesso uso di operandi di conversione (casting implicito) per rendere omogenei gli operandi.

Spesso vengono usati nel C: si considerino le operazioni 'a' / 5 e 30 + "2"
Java è fortemente tipizzato, più del C e del Pascal, mentre Perl ha una tipizzazione molto debole.
Con la tipizzazione debole il risultato di una operazione può cambiare a seconda del linguaggio scelto per programmare, ad esempio:

var x:=5; var y:="37"; y + x

quanti risultati diversi possono esserci?

- In C si basa tutto sull'aritmetica dei puntatori
- in Java/Javascript, x viene convertito a stringa (x + y = "537")
- in Visual Basic e in Perl, y viene convertito a intero (x + y = 42)
- in Python non viene accettato ed esce con errore

Tipizzazione safe

Un linguaggio di programmazione adotta una tipizzazione safe (sicura) dei dati se non permette ad una operazione di casting implicito di produrre un crash, ad esempio Python.

Tipizzazione unsafe

Un linguaggio di programmazione adotta una tipizzazione unsafe (non sicura) dei dati se consente operazioni di casting che possono produrre eventi inaspettati o crash (ad esempio il C).

DUCK TYPING: Tipizzazione e polimorfismo

Il *Polimorfismo* è la capacità di differenziare il comportamento di parti di codice in base all'entità a cui sono applicati, implica quindi il riuso del codice.

Nello schema classico (ad esempio Java) di un linguaggio ad oggetti, il polimorfismo è di solito legato al meccanismo di ereditarietà. L'ereditarietà garantisce che le classi possano avere una stessa interfaccia.
In Java le istanze di una sottoclasse possono essere utilizzate al posto delle istanze della superclasse (polimorfismo per inclusione).
L'overriding dei metodi permette che gli oggetti appartenenti alle sottoclassi rispondano diversamente agli stessi utilizzi (un metodo dichiarato nella superclasse viene implementato in modi diversi nelle sottoclassi – metodi polimorfi).

In questo caso, applicare il polimorfismo e individuare la signature valida di un metodo implica:

- individuare la classe di appartenenza dell'oggetto, e cercarvi una signature valida per il metodo
- in caso di mancata individuazione, ripetere il controllo per tutte le superclassi (operation dispatching – potenzialmente risalendo fino alla classe radice – es. Object). Questo implica risalire tutta la gerarchia dell'ereditarietà e cercare la prima "classe giusta", quella con la signature valida per il metodo.

[Signature: la signature di un metodo sono un insieme di informazioni che lo identificano. Solitamente queste informazioni sono: *nome*, *numero di parametri*, *tipi di parametri*. Es. se ho due metodi con lo stesso nome che svolgono operazioni diverse, posso distinguerli a livello macchina in base al numero di parametri]

La ricerca della classe adatta nella gerarchia delle classi può essere effettuata:

- A tempo di compilazione
- A tempo di esecuzione

In molti linguaggi ad oggetti (ad esempio C++) avviene a tempo di compilazione, in quanto il costo a tempo di esecuzione è ritenuto troppo elevato.

In Java e nei linguaggi dinamici, avviene a tempo di esecuzione. Questo implica che può essere molto costoso in termini di overhead.
Nei linguaggi dinamici però esiste una alternativa: si va a sfruttare il *Duck Typing* che permette di realizzare il concetto di polimorfismo senza dover necessariamente usare meccanismi di ereditarietà (o di implementazione di interfacce condivise). "Se istanzio un oggetto di una classe e ne invoco metodi/attributi, l'unica cosa che conta è che i metodi/attributi siano definiti per quella classe". Questo meccanismo è reso possibile grazie al Type Checking dinamico, ovvero il controllo (duck test) viene effettuato dall'interprete a runtime.

Ad esempio: Animale, Cane e Gatto

Una funzione *funz()* prende in ingresso un parametro formale (non tipizzato, quindi senza tipo) e ne invoca il metodo *CosaMangia()*. Non è necessario che le classi *Cane* e *Gatto* siano sottoclassi di *Animale* per essere passate a *funz()* , basta che abbiano un metodo chiamato *CosaMangia()*.
All'interprete interessa solo che i tipi di oggetto espongano un metodo con lo stesso nome e con lo stesso numero di parametri in ingresso.

Esempio Python

```
class Duck:
        def quack(self)
                print("quack quack")


def is_animal(var)
        var.quack()
```

definiamo la classe *Duck* con il metodo *quack* e all'esterno della classe creiamo una funzione chiamata *is_animal(var)* alla quale possiamo passare un qualsiasi dato, ma che effettuerà la print di "quack quack" soltanto se quel dato è un istanza della classe *Duck,* la quale possiede il metodo *quack()*.

In generale il polimorfismo a livello di tipi di dato permette di:

- Differenziare il comportamento dei metodi in funzione del tipo di dato a cui sono applicati
- evitare di dover predefinire un metodo, classe o struttura di dati appositi per ogni possibile combinazione di dati

tutto questo implica sempre il riuso del codice

[da notare che il riuso del codice implica minori costi di sviluppo del software al costo, in caso di errori, di propagare un bug all'interno di tutto il sistema]

In un linguaggio tipizzato dinamicamente, tutte le espressioni sono intrinsecamente polimorfe.

In conclusione a questo capitolo, quasi tutti i linguaggi dinamici adottano un meccanismo di gestione degli attributi e dei metodi degli oggetti basato su duck typing. Questo perché permette di ottenere polimorfismo senza dover per forza incorrere nell'overhead dovuto all'ereditarietà. Ovviamente anche qui si deve considerare la

possibilità di errori a tempo di esecuzione: in questo modo non è possibile imporre che gli oggetti rispettino una interfaccia comune.

3. Architettura

Gli strumenti per la generazione di codice eseguibile sono classificabili in due principali categorie distinte:

- Compilatori
- Interpreti

Il *Compilatore* è un software che traduce un codice (sorgente) scritto in un linguaggio di programmazione in un altro linguaggio (codice oggetto) di più basso livello. Ad esempio nel C, il compilatore traduce il codice in linguaggio macchina binario.

L'*Interprete* è un software che considera porzioni limitate (statement) di codice sorgente o intermedio, le traduce in codice macchina e le esegue direttamente.

Lo strumento utilizzato per la generazione di codice eseguibile *non* è una caratteristica del linguaggio.
Gli svantaggi di un linguaggio interpretato sono:

- la maggiore lentezza nell'esecuzione dovuta a una traduzione a runtime
- la richiesta di più memoria
- la richiesta della presenza del software interprete sul computer

Gli svantaggi di un linguaggio compilato sono:

- fornisce minore portabilità
- è meno flessibile (ad esempio non permette meccanismi di type checking dinamico)
- supporto per debugging a runtime meno flessibile e potente

Gli obiettivi dei linguaggi dinamici come Python sono la portabilità e la rapidità di prototipazione e inoltre la velocità maggiore di un linguaggio completamente interpretato (ad esempio la Shell).
Nonostante ciò, esiste un approccio ibrido ovvero *Compilazione + interpretazione,* con un formato di rappresentazione intermedia del codice indipendente dall'architettura. A seconda del tipo di formato intermedio, si distinguono due diversi modelli di esecuzione:

- Modello "*Abstract Syntax Tree*" (AST)
- Modello "*Bytecode*"

Modello “Abstract Syntax Tree”

Questo modello prevede l’utilizzo di un compilatore e di un interprete.
Il *compilatore* traduce il codice sorgente in una rappresentazione ad albero sintattico (Abstract Syntax Tree). In questa fase (a seconda del linguaggio) è possibile l’interpretazione o l’esecuzione di speciali sezioni del codice (ad esempio le inizializzazioni: BEGIN/END in Perl).
L’interprete, attraverso algoritmi di vista dell’AST, considera “porzioni di albero”, le traduce in istruzioni corrispondenti e le esegue (statement di codice). Esiste la possibilità di inserire durante questa fase, una fase di compilazione del codice.

Modello “Bytecode”

Il modello Bytecode prevede l’utilizzo di un compilatore e di una macchina virtuale (interprete). Il *compilatore* traduce il codice sorgente in un linguaggio intermedio di basso livello, portatile (Bytecode o p-code) che rappresenta un passo successivo (potenzialmente più ottimizzato) rispetto all’AST. Il Bytecode viene installato sui sistemi di destinazione, dove viene tradotto in codice macchina ed eseguito.
La *macchina virtuale*, o *interprete*, fornisce una astrazione di un sistema operativo dove il Bytecode può essere considerato come il ‘codice macchina’ della macchina virtuale. Esso traduce le istruzioni e le richieste di sistema espresse in Bytecode nelle richieste concrete al sistema operativo realmente installato sulla macchina. Le funzionalità di base della macchina virtuale sono contenute nella libreria di funzioni detta *RunTime Library.*

RunTime Library + MV = RunTime Environment

Compilazione Just-In-Time (JIT)

La compilazione Just-In-Time o traduzione dinamica (dynamic translation) viene utilizzata nei linguaggi che adottano un modello a Bytecode (Java, Python, Ruby, PHP, …). Questa viene utilizzata da quasi tutte le recenti implementazioni di macchine virtuali.
L’ambiente di esecuzione (macchina virtuale) incorpora un compilatore interno just-in-time che, a runtime:

- compila porzioni di codice bytecode traducendole nel linguaggio macchina nativo del computer ospite
- le memorizza per il riuso

Viene ovviamente impiegato per ragioni di efficienza come le ottimizzazioni su porzioni di codice eseguite frequentemente.

Compilazione completa

Le operazioni principali di un compilatore C-like sono:

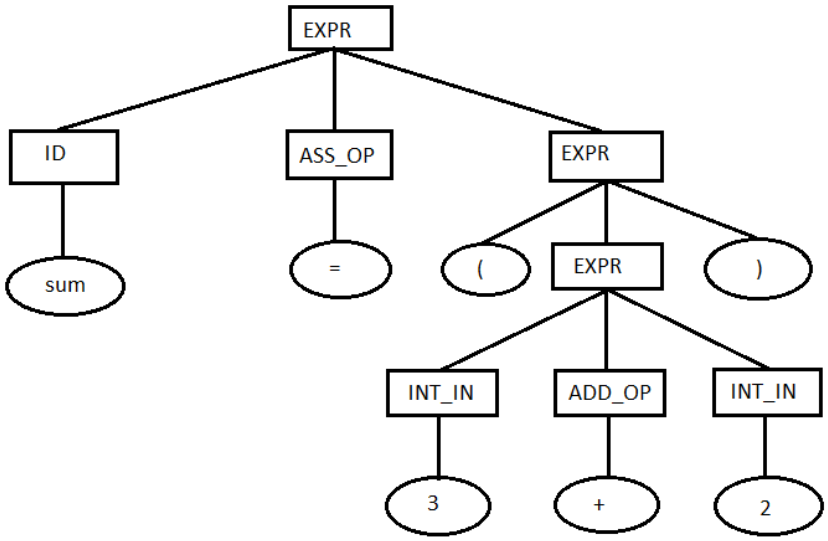
- *Analisi (lessicale, sintattica e semantica):* il testo viene diviso in unità di base e analizzato con diversi tipi di controlli per poi generare in output una rappresentazione intermedia del codice detta Abstract Syntax Tree (AST)
- *Generazione del codice:* la rappresentazione intermedia viene tradotta nel formato finale
- *Ottimizzazione del codice:* il codice risultante viene ottimizzato secondo un qualche criterio specifico (velocità, uso memoria, consumo di energia)

L'Analisi Lessicale ha come obiettivo il dividere il flusso di caratteri in ingresso (codice sorgente) in tante unità chiamate *Token* . Un Token è un elemento minimo, non ulteriormente divisibile, di un programma (ad esempio keyword come for, while, nomi di entità, operatori).
La fase di *Tokenizzazione* è eseguita da un analizzatore lessicale detto *Scanner* (o *Lexer*). Lo *Scanner* è una macchina a stati finiti che riconosce i possibili token definiti tramite espressioni regolari (ad esempio un numero intero è un carattere seguito da una sequenza di cifre).
Ad ogni token identificato è associata una categoria (significato nel lessico del linguaggio) assegnata in base all'espressione regolare che ha riconosciuto il token. Un piccolo esempio di tabella può essere:

TOKEN	**CATEGORIA**
Sum	IDENTIFIER
:=	ASSIGN_OP
3	INT_NUMBER
+	ADD_OPERATOR
2	INT_NUMBER

L'*Analisi sintattica* prende in ingresso la sequenza di token ed esegue il controllo sintattico: verifica che i token formino una espressione valida. Ad esempio, l'espressione "a = + = b" è riconosciuta non valida solo a questo livello: prima lo scanner avrebbe targato ogni operatore come identifier, assign o add.

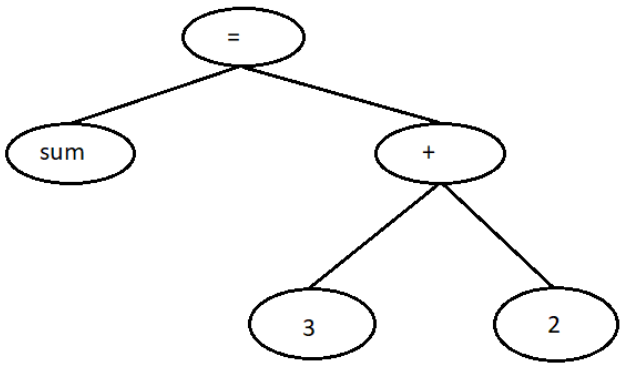
Questa analisi viene svolta da un analizzatore sintattico o parser. Il risultato è un albero di sintassi o *Parser Tree*, ovvero un albero che rappresenta la struttura sintattica di una espressione in accordo alla grammatica usata. I token sono tutti rappresentati da nodi foglia.
Esempio di Parse Tree generato per sum = (3 + 2)



L'*Analisi Semantica* ha come obiettivo il rilevamento di (sequenze di) istruzioni non corrette, ovvero senza un significato valido a livello semantico. I controlli tipici in questa fase sono:

- il controllo che gli identificatori siano stati dichiarati e inizializzati
- il Type Checking (controllo di tipo) sulle operazioni possibili e assegnamenti di valori corretti

Durante questa fase viene creata una tabella dei simboli con informazioni su ciascun simbolo (nome, scope, tipo, …). Il risultato dell'analisi semantica è sempre un AST dove operatori e keyword *NON* sono nodi foglia. È astratto in quanto non rappresenta esplicitamente tutti i dettagli rappresentati nella sintassi, che vengono però rappresentati in modo implicito (ad esempio con le parentesi).



Generazione del codice da parte del Compilatore

Il modulo di generazione del codice:

1. Prende in input l'AST prodotto dall'analisi semantica
2. Converte la struttura ad albero in una sequenza di codice/istruzioni
3. Spesso integra alcune tecniche di ottimizzazione
4. La sequenza prodotta può ancora essere intermedia, in quanto potrebbe essere prevista una fase separata di ottimizzazione, seguita da generazione del codice target/macchina

Un esempio di linguaggio intermedio può essere il linguaggio Assembly.

La generazione del codice avviene attraverso tre processi: *Instruction Selection, Instruction Scheduling e Register Allocation.*

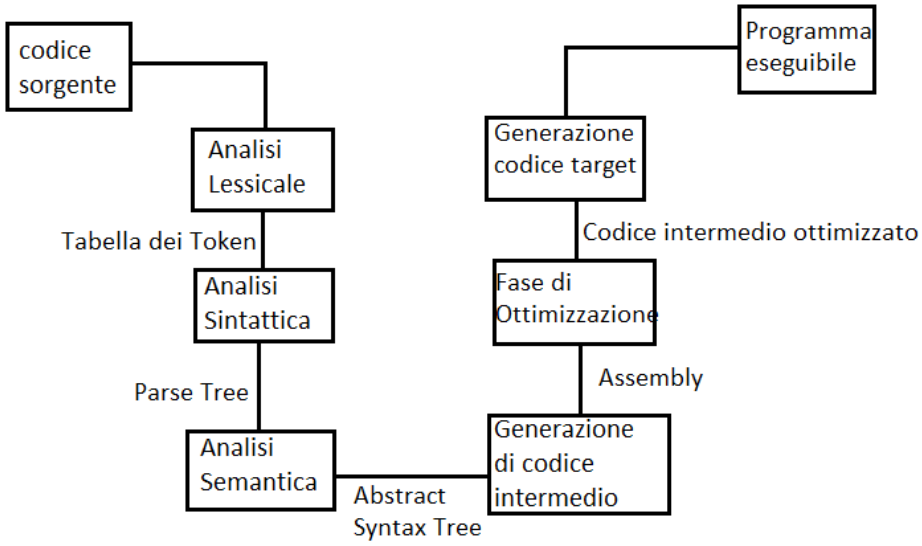
Nell'*Instruction Selection* vengono definite le istruzioni che rappresenteranno la conversione dalla rappresentazione ad albero. Si procede spezzando la rappresentazione ad albero nel numero più basso possibile di *tile*, dove un *tile* è una porzione di albero che può essere implementato con una singola istruzione nel codice generato.

L'ottimizzazione è integrata nella fase di instruction selection, ad esempio una traduzione naive può generare codice inefficiente: può esserci la necessità di eliminare accessi ridondanti alla memoria, riordinando e fondendo istruzioni, e sfruttando l'uso dei registri.

Nell' *Instruction Scheduling* si sceglie la sequenza in cui sistemare le istruzioni. L'obiettivo è quello di migliorare il parallelismo a livello di istruzioni (con maggior throughput di istruzioni). La tecnica di ottimizzazione tipica è quella dell' *Instruction Pipeline* (classica pipeline su architetture RISC – Reduced Instruction Set Computer – applicazioni embedded). Il tentativo è quello di utilizzare in modo parallelo accessi a livelli diversi di memoria e risorse di calcolo.

Nella *Register Allocation* si decide come allocare le variabili dentro ai registri della CPU. L'obiettivo è quello di cercare di assegnare quante più variabili possibili ai registri della CPU per velocizzare l'accesso stesso. Ovviamente l'accesso nei registri è più veloce rispetto all'accesso in RAM ma spesso le variabili contemporaneamente in uso

sono in numero maggiore dei registri disponibili. L'ottimizzazione integrata va a scegliere le variabili che vengono accedute più frequentemente (scrittura vs lettura)



4. Architettura del Perl

Il modello di esecuzione del Perl è ad Abstract Syntax Tree, composto da Compilazione e Interpretazione. Entrambe le fasi sono gestite dall'eseguibile Perl (o interprete Perl): il compilatore produce un AST e successivamente l'interprete prende in input l' AST, lo esegue ed esegue operazioni primitive del Perl (opcode) secondo un ordine preciso.

La costruzione dell'AST deve soddisfare due requisiti fondamentali:

- Elencare gli opcode
- specificarne l'ordine di sequenza

L' AST è un albero in cui ogni nodo non foglia è un opcode interno del Perl (eseguibile in una istruzione dall'interprete); ogni nodo foglia è un operando e l'ordine di visita indica la precedenza degli operatori. L'albero viene visitato in post-ordine (prima il sottoalbero sinistro, poi quello destro ed infine il nodo radice).

Durante la fase di compilazione, come prima cosa si effettua l'analisi del programma per determinare la presenza di eventuali errori, quindi analisi lessicale, sintattica e semantica (type checking parziale: nessun controllo di dichiarazione, identificazione di tipo limitata a valori costanti già noti). Successivamente può essere eseguito il codice: con la direttiva *use* si effettua il caricamento dei moduli (interpretata subito dopo la sua scansione); i blocchi di codice contrassegnati con particolari parole chiave (ad esempio BEGIN) sono interpretati ed eseguiti subito dopo la loro scansione.

Analisi

L’analisi avviene tramite l’uso di tre moduli distinti:

- *Lexer:* identifica i token all’interno del codice sorgente (lexer analyzer)
- *Parser:* associa gruppi contigui di token a costrutti (espressioni, statement) in base alla grammatica del Perl(analisi sintattica e semantica)
- *Optimizer:* riordina e riduce i costrutti prodotti dal parser, con l’obiettivo di produrre sequenze di codice equivalenti più efficienti

Nel Perl la fase di ottimizzazione viene inserita all’interno dell’analisi che produce l’AST.

I moduli Parser e Optimizer sono responsabili della costruzione dell’AST e della sua ottimizzazione attraverso 3 fasi:

- Bottom-Up Parsing
- Top-Down Optimizer
- Peephole Optimizer

Non tutte le fasi sono sequenziali e consecutive, alcuni passi sono interallacciati. Ad esempio l’optimizer a volte non può entrare in azione fino a quando il parser non ha raggiunto un certo punto della costruzione dell’albero.

Bottom-Up Parsing

Nella fase di Bottom-Up Parsing, il parser riceve in ingresso i token prodotti dal lexer, dai quali costruisce l’AST. Il parsing è di tipo “bottom-up” perché l’AST viene costruito partedno dalle foglie, quindi parte dagli operandi e segue con gli opcode.
Dopo la costruzione di un nodo non foglia (opcode), si verifica se la semantica dell’opcode relativo è congruente (ad esempio, numero corretto di parametri per una funzione o per un operatore). Poi si costruisce l’ordine di visita dei nodi per quel sottoalbero.
L’ordine di visita viene memorizzato nell’AST stesso (acceduto tramite puntatori), necessario per garantire maggior efficienza all’interprete.

Non appena un sottoalbero dell’AST è generato, viene creata una struttura ciclica che collega i nodi secondo l’ordinamento di esecuzione (visita in Post-Order) a partire dalla radice. Il nodo radice viene collegato al primo nodo da visitare, poi il primo nodo da visitare viene collegato al secondo nodo da visitare, e così via… Quando viene aggiunto un altro sottoalbero AST al precedente, il ciclo radice-primo nodo si spezza e viene ricostruito con il nuovo albero.
In questo modo l’interprete potrà individuare il prossimo nodo da visitare in maniera efficiente, (con costo O(1)) a partire dal nodo radice.

Top-Down Optimizer

La fase di Top-Down Optimizer è interallacciata con la precedente. Non appena un sottoalbero dell'AST viene prodotto, viene scandito dalla radice alle foglie per eventuali ottimizzazioni, ovvero ottimizzazioni locali.
Una volta identificato il "contesto" di un nodo, esso viene propagato verso il basso ai nodi figli (*Context Propagation* top-down).

Peephole Optimizer

La fase Peephole (spioncino) Optimizer svolge un ottimizzazione su porzioni di codice piccole ed inizia dopo la costruzione dell'AST. Il peephole optimizer percorre l'albero seguendo il flusso di esecuzione (segue la struttura dei next-opcode) e considera piccole porzioni di albero per cercare di semplificare/ridurre gli opcode.
Ad esempio le porzioni di codice che hanno uno spazio di memorizzazione locale per le variabili: singole subroutine.
Il Peephole optimizer può effettuare diversi tipi di ottimizzazione su piccole porzioni di codice.
Con la *Strenght reduction* si va a rimpiazzare operazioni "lente" con operazioni equivalenti e più veloci. Ad esempio: mul ax,2 → shift_left ax.

La *Null sequences* è l'eliminazione di operazioni inutili.

Tramite il *Costant Folding*, se trova operazioni con operandi tutti costanti, calcola l'espressione finale e sostituisce all'espressione di partenza il suo risultato. Ad esempio se trova che ci sono due espressioni costanti tali che la loro somma faccia 42, sostituisce il valore costante con l'operando, ottenendo un AST ottimizzato equivalente.

Con le *Combine Operations* rimpiazza un insieme di operazioni con un operazione singola equivalente.

Le *Algebraic Laws* va ad usare leggi algebriche per semplificare o riordinare le istruzioni, ad esempio (a * 2) + (b * 2) → (a + b) * 2

Le *Special Case Instructions* sono istruzioni speciali progettate per semplici operandi (stringhe, dotted names, …)

Interprete

Quando la compilazione è terminata, l'AST viene passato all'interprete che esegue gli opcode dell'AST nella sequenza indicata. L'interprete usa una macchina basata su stack per eseguire il codice. L'idea di base è che ogni opcode manipola degli operandi, i quali

vengono inseriti in uno stack secondo una modalità molto simile alla notazione polacca inversa, ad esempio se devo eseguire un calcolo su a e su b: Push a, Push b, opcode(pop a, pop b, calcolo, push risultato).

L'interprete Perl crea diversi stack, fra cui:

- *Operand stack:* memorizzazione degli operandi usati dagli operatori e dei risultati
- *Save stack:* memorizzazione delle variabili il cui scope è stato alterato da altre; ad esempio una variabile globale *i* il cui valore è stato offuscato da una variabile locale *i* all'interno di una funzione.
- *Return stack*: memorizzazione degli indirizzi di ritorno delle funzioni

Sempre l'interprete Perl può effettuare compilazioni di espressioni a run-time attraverso la funzione *eval*, la quale effettua una compilazione della espressione prima di passare alla sua esecuzione. Viene svolta in questo modo per motivi di sicurezza: nel caso,ad esempio, in cui l'espressione contenga input dall'esterno, se ne deve controllare la validità prima di eseguirla. In caso di errore salta il resto dell'espressione e inserisce un messaggio di errore in $EVAL_ERROR.

Subroutine speciali

Vengono richiamate (eseguite) in momenti specifici:

- Durante la fase di compilazione
- Durante la fase di esecuzione

La subroutine BEGIN viene eseguita (interpretata) all'inizio della fase di compilazione, ovvero nel punto migliore per inizializzare/modificare l'ambiente prima che inizi la compilazione.
La subroutine INIT viene eseguita all'inizio della fase di esecuzione del programma, nel punto corretto per eventuali inizializzazioni di variabili, strutture dati, ecc….
La subroutine CHECK viene eseguita alla fine della fase di compilazione del programma, nel punto migliore per eventuali controlli sulla buona uscita della compilazione. A differenza di BEGIN e INIT, l'ordine di esecuzione di CHECK multiple è inverso a quello delle chiamate.
La subroutine END viene eseguita alla fine dell'esecuzione del programma, ovvero nel punto corretto per eseguire operazioni di chiusura (chiusura delle connessioni, pulizia dei files, ecc…). è eseguita anche in caso di istruzione die e l'ordine di esecuzione è inverso come per la CHECK.

Controllo sintattico

L'opzione -c permette di invocare il servizio di controllo sintattico del programma: perl -c file.pl
Non esegue, ma ci informa del risultato dell'analisi: file.pl syntax OK

Si può anche ispezionare l'albero AST prodotto dalla fase di compilazione con l'operazione -MO=Concise, ed es. perl -MO=Concise file.pl
Due output (rendering) con diverse opzioni: -basic (default) che mostra la struttura dalla radice alle foglie (visita in preorder) e -exec che mostra gli opcode nell'ordine in cui sono eseguiti (visita in post-order).

Output layout

- La prima colonna contiene l'opcode sequene number (utili nel caso di loop)
- La seconda colonna <x> tipo di opcode (<2> BINOP, <1> UNOP, …)
- La terza colonna contiene il nome dell'opcode (es. add, assign) eventualmente seguito da informazioni specifiche tra parentesi
- La quarta colonna contiene gli opcode flags

Esso tiene traccia di ottimizzazioni che tolgono opcode, gli opcode tolti compaiono con sequence number '-' (non eseguiti) e "ex-opcodeName" come nome.

Opzione -basic

Andiamo a sottolineare questa opzione in quanto mostra gli opcode come sono nell'albero offrendo un rendering top-down con la visita in pre-ordine (radice, sottoalbero sinistro e sottoalbero destro). Riflette il modo in cui lo stack può essere usato per valutare le espressioni dal punto di vista semantico, ad esempio l'opcode *add* opera sui due termini che si trovano sul livello inferiore dell'albero.
La freccia nell'ultima colonna dell'output indica il sequence number del prossimo opcode da eseguire; questa opzione permette di ricavare l'ordine di esecuzione .

5. Gestione della memoria

Riguardo la gestione della memoria nei linguaggi dinamici, esistono due filosofie:

- Gestione della memoria (Heap) con allocazione e rilascio delle risorse, che segue la filosofia del C. La gestione viene lasciata al programmatore attraverso le primitive di sistema (malloc, free, …) che è veloce ma molto soggetta ad errori.
- Gestione della memoria lasciata al sistema, che segue la filosofia di Java. Quindi esiste un gestore automatico della memoria reso disponibile dall'ambiente di esecuzione.

Per quanto riguarda la gestione manuale della memoria possiamo valutarne i vantaggi e gli svantaggi.

Vantaggi:

- La velocità delle operazioni di gestione è elevata (è essenzialmente quella del kernel sottostante)
- i pattern di acquisizione/rilascio sono espliciti

Svantaggi:

- Implica un elevato onere di programmazione, ci sono tanti rischi tra i quali: la gestione dell'aritmetica dei puntatori (segmentation fault), i puntatori a memoria liberata (dangling reference) e memoria non liberata (memory leak).
- La gestione manuale difficilmente scala con le dimensioni del programma (è molto facile dimenticarsi una qualche free() e incorrere in un memory leak)

Gestione della memoria nei linguaggi dinamici

I linguaggi dinamici seguono la filosofia Java: il sistema (a runtime environment) mette a disposizione un gestore automatico della memoria che alloca e dealloca a tempo di esecuzione.
La gestione viene realizzata dal *garbage collector* che è la forma più utilizzata di gestione automatica della memoria. quest'ultimo (a runtime environment), svolge le seguenti operazioni:

- Individua oggetti (variabili, strutture dati, istanze di classe, …) che non potranno mai più essere referenziate dall'applicazione (garbage)
- rilascia la memoria relativa a tali entità

I linguaggi dotati di questo strumento sono: Java, C#, Smalltalk, Lisp, Haskell, Python (dal 2.0), Ruby, Lua, Tcl. I linguaggi nati con una gestione manuale ma estendibili con un garbage collector sono c e C++. Altri linguaggi che fanno uso di tecniche dinamiche alternative alla garbage collection sono Perl e PHP.

Tracing garbage collector

La maggior parte dei garbage collector segue la strategia detta *tracing garbage collection* ovvero traccia gli oggetti ancora referenziabili attraverso una catena di riferimenti che parte da oggetti "radice". Un oggetto si definisce raggiungibile in modo ricorsivo se è raggiungibile per se stesso o se esiste un riferimento da esso tramite un oggetto stesso raggiungibile.

La raggiungibilità può essere di due tipi:

1. *Diretta*, quando una variabile contiene l'oggetto. Gli oggetti radice sono direttamente raggiungibili, ad esempio variabili globali, parametri della funzione corrente in esecuzione

2. *Indiretta*, quando una variabile contiene un puntatore all'oggetto. Ogni oggetto riferito da un oggetto raggiungibile è, a sua volta, un oggetto raggiungibile (proprietà transitiva)

La non raggiungibilità di un oggetto può avere due cause:

- Una causa *Sintattica* (syntactic garbage), se gli oggetti non sono più raggiungibili per via di vincoli sintattici, come ad esempio se riassegno un puntatore o un riferimento. È semplice da verificare perché la non raggiungibilità è intesa solo in senso sintattico da quasi tutti i garbage collector.
- Una causa *Semantica* (semantic garbage), se gli oggetti non sono più raggiungibili per via del flusso del codice eseguito, ad esempio un branch di codice mai usato a runtime. Non esiste un algoritmo in grado di identificare i semantic garbage per ciascun possibile input, quindi si usano euristiche nei garbage collector

Gli algoritmi di garbage collection possono essere richiamati periodicamente (come in Java) o essere attivati sulla base di una soglia (come in Python). Nei sistemi a soglia viene attivato un ciclo quando viene notificato al collector che il sistema ha bisogno di memoria. Alcuni esempi di algoritmi più usati sono l'algoritmo naive *Mark and Sweep,* inefficiente con set dati scandito più volte, e l'algoritmo *Tri-color Making* che è l'evoluzione del precedente.
Il *Mark and Sweep* è uno de primi algoritmi utilizzati, ad ogni oggetto in memoria è associato un flag (un bit) utilizzato dall'algoritmo di garbage collection. Il valore de flag viene interpretato nel modo seguente:

- 0/False significa che l'oggetto non è raggiungibile (default)
- 1/True significa che l'oggetto è riconosciuto come raggiungibile

Ogni oggetto viene scandito una volta, se è raggiungibile come radice o da un oggetto radice si imposta il suo flag a 1. il set viene scandito una seconda volta, ovvero si libera la memoria per gli oggetti con flag a 0 (e si re-imposta a 0 il flag degli oggetti nei quali esso è 1).
L'algoritmo è semplice ma purtroppo poco performante, nel senso che l'interprete si deve interrompere durante l'intera esecuzione dell'algoritmo di garbage collection. L'intero insieme degli oggetti deve essere scandito più volte linearmente (analisi e rilascio di memoria). Ci sono problemi in sistemi con memoria paginata e infine ci sono problemi in ambienti che necessitano di basse latenze di risposta o in sistemi real-time.

La soluzione a tutti questi problemi è *l'algoritmo di Tri-color Marking.*
Questo algoritmo sfrutta tre sottoinsiemi che servono a mantenere lo stato degli oggetti a run-time:

- *White set* contiene gli oggetti candidati alla rimozione (che alla fine dell'algoritmo saranno distrutti)
- *Grey set* contiene gli oggetti che sono raggiungibili, ma i cui oggetti referenziati non sono ancora stati analizzati

- *Black set* contiene gli oggetti raggiungibili che non referenziano alcun oggetto nel *White set*

Nella prima fase dell'algoritmo (inizializzazione), l'algoritmo partiziona nei tre sottoinsiemi l'insieme degli oggetti presenti nella memoria. Tutti gli oggetti che sono referenziati a livello radice vengono inizialmente messi nel Grey set. Tutti gli altri oggetti vengono posizionati inizialmente nel White set. Il Black set è inizialmente vuoto (conterrà alla fine gli oggetti da non rimuovere).
Gli oggetti possono solo passare da White set a Grey Set e da Grey Set a Black set, non sono possibili altri tipi di spostamenti.
Nella seconda fase, l'algoritmo, finché ci sono oggetti nel Grey set sceglie un oggetto O nel Grey set e lo sposta nel Black set. Successivamente identifica tutti gli oggetti referenziati direttamente da O e infine considera l'insieme di oggetti con almeno una referenza nel White set e li sposta nel Grey set.
Nella terza fase dell'algoritmo la logica mostra che gli oggetti nel White set sono:

- non raggiungibili direttamente (prima fase)
- non raggiungibili neppure indirettamente (seconda fase)

che sono poi quelli candidati alla rimozione. La memoria relativa agli oggetti nel White set viene quindi rilasciata.

Uno dei vantaggi dell'algoritmo Tri-color Marking è che può essere eseguito senza richiedere l'interruzione completa dell'interprete. Le prime due fasi dell'algoritmo vengono eseguite contestualmente all'allocazione ed alla modifica di oggetti; solo durante la terza fase viene interrotto il programma.
Inoltre non ci sono molteplici scansioni dell'intero set di oggetti; nella seconda fase viene scandito solo il Grey set con l'accesso diretto agli oggetti riferiti nel white set. Nella terza fase accede solo agli oggetti rimasti nel White set.

Per quanto riguarda il rilascio della memoria sono possibili due modalità:

- rilasciare gli oggetti irraggiungibili senza spostarli
- copiare tutti gli oggetti raggiungibili in una nuova area di memoria

All'apparenza costosa e inefficiente, la seconda strategia porta più vantaggi:

1. Grandi regioni contigue di memoria disponibili. La prima opzione dopo un po' di tempo porta a heap molto frammentati
2. Sono possibili ottimizzazioni in quanto oggetti che si riferiscono l'uno all'altro sono messi vicini aumentando la probabilità che si trovino sulla stessa linea di memoria della cache o della pagina di memoria virtuale.

Generational garbage collector

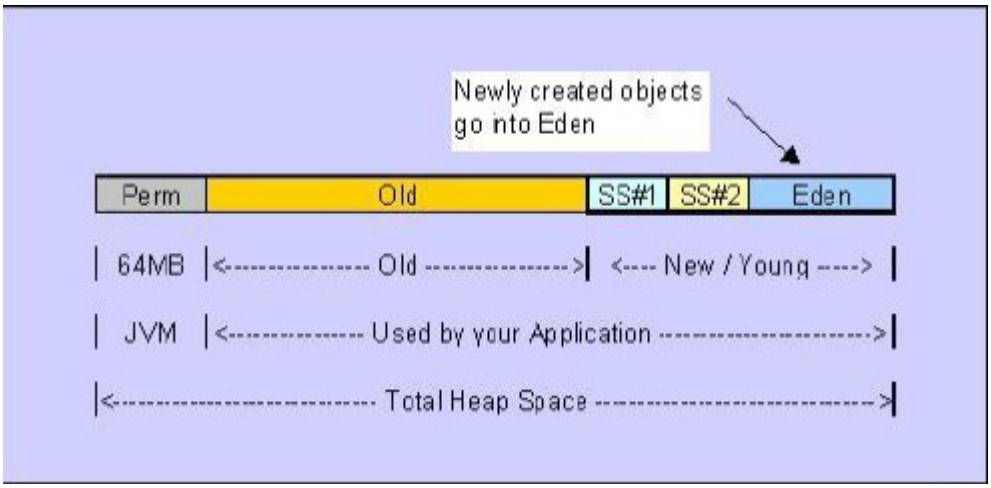
Uno studio empirico dimostra che, per diverse tipologie di programmi, gli oggetti creati più di recente hanno la maggiore probabilità di diventare irraggiungibili nell'immediato futuro. Gli oggetti creati in memoria hanno una mortalità infantile molto elevata. Tale

osservazione è stata presa come ipotesi di lavoro per la realizzazione di strategie di gestione della memoria: ipotesi generazionale.

Un garbage collector generazionale divide gli oggetti per generazioni. Nei vari cicli di esecuzione, fa controlli frequenti solo sugli oggetti delle generazioni più giovani e contemporaneamente tiene traccia della creazione di riferimenti (intra e inter generazione). Se l'ipotesi generazionale è vera, riesce ad essere molto più veloce anche se più impreciso (possono sfuggire temporaneamente oggetti più vecchi diventati irraggiungibili).
L'algoritmo associa "regioni" dell'heap a diverse generazioni via via più vecchie.
New/Young Generation contiene gli oggetti recenti:

- Eden: oggetti appena creati
- Survivor Space 1 e 2: oggetti sopravvissuti a precedenti cicli di garbage collection

Old Generation contiene gli oggetti più vecchi.
Perm: contiene gli oggetti permanenti utilizzati dalla virtual machine (tipo definizioni di classi e metodi).



Ogni volta che una regione tende a riempirsi (soglia) viene invocato un ciclo di garbage collection sugli oggetti in essa contenuti. Gli oggetti raggiungibili della regione vengono copiati nella regione successiva (più vecchia); gli oggetti irraggiungibili vengono eliminati e la regione viene svuotata e può essere utilizzata per l'allocazione di nuovi oggetti.
Esiste un'eccezione per Survivor Space 1, dove gli oggetti rimangono per un certo numero di cicli prima di essere spostati nella generazione Old. Lo spazio di assegnamento alle regioni influenza la frequenza di invocazione (è un compromesso precisione-prestazioni).
Le generazioni più giovani, come la Eden, tendono a riempirsi più in fretta; in sostanza l'algoritmo opera molto frequentemente su un set ridotto. In questo modo realizza una garbage collection incrementale e veloce (agendo su zone ridotte dell'heap). L'unico svantaggio è che l'approccio euristico non è ottimale, alcuni oggetti irraggiungibili

potrebbero non essere scoperti. Per questo motivo, spesso si fa uso di approcci ibridi dove si alternano i cicli dell'algoritmo generazionale uniti a cicli occasionali di "major garbage collection" su tutta la heap.

I termini "*minor cycle*" e "*major cycle*" sono spesso usati per riferirsi ai diversi cicli di garbage collection negli approcci ibridi. Per i cicli di major collecion solitamente viene usati un algoritmo *Mark and Sweep*. Nei minor cycle viene usato un approccio in movimento, spesso usato anche nei major cycle per combattere la frammentazione dell spazio di memoria (Java e .NET framework).

Alternative al Garbage collector → Reference Counting

Alcuni linguaggi dinamici adottano alternative più snelle di un garbage collector classico. La più interessante di tali alternative è il *reference counting* , usato in Perl e PHP. A ciascun ogetto viene associato un contatore degli oggetti che lo stanno referenziando, quando il contatore scende a 0 significa che nessuno sta più referenziano l'oggetto e di conseguenza può essere eliminato.
Questo algoritmo è molto più veloce di qualunque algoritmo di garbage collection, ci si può letteralmente "dimenticare" di chiudere i descrittori dei file. Quando si esce da un blocco di codice, il reference count della variabile descrittore va a 0; la variabile descrittore viene immediatamente distrutta. Con un garbage collector questa operazione non è necessariamente immediata.
Quindi questo algoritmo è molto più reattivo… ma quali sono i suoi difetti?

Di contro, è necessario mantenere e gestire un contatore in ogni oggetto che è continuamente aggiornato. Questo può portare a Overhead di memoria e risorse di calcolo. L'algoritmo di reference counting inoltre presenta il rpoblema dei riferimenti circolari: consideriamo 2 oggetti A e B che puntano l'uno all'altro, sia A che B hanno due riferimenti:

- quello della variabile che li contiene (diretto)
- quello dell'oggetto che punta

Quando uno dei due oggetti (ad esempio A) esce dal suo scope:

- non è più accessibile tramite la sua variabile
- La sua memoria non può essere liberata, dal momento che un altro oggetto la sta referenziando (rischio di dangling reference)

Una soluzione a questo problema sono gli algoritmi di *cycle-detecting* con adozione dei riferimenti deboli, ovvero riferimenti circolari esplicitamente marcati come deboli e non considerati dal reference counting.

6. Metaprogrammazione

I linguaggi dinamici mettono a disposizione meccanismi di gestione avanzata del codice. Permettono quindi di adattare a runtime il comportamento del programma senza bisogno di ricompilazione. Questa caratteristica è veloce e flessibile in quanto permette l'adattamento del codice a situazioni differenti.
Il cuore del metaprogramming sta:

1. Nella capacità di (auto) analisi del codice a tempo di esecuzione
2. Nella generazione e/o modificazione del codice a tempo di esecuzione

Inoltre ci permette di gestire in modo dinamico funzioni speciali e intercettare e gestire errori a runtime.

Metaprogramming significa scrivere un programma (metaprogramma) in grado di generare, analizzare e/o modificare altri programmi o (parti di) se stesso a runtime.
L'*obiettivo* è quello di produrre programmi estremamente flessibili, in grado di reagire autonomamente rispetto a situazioni/configurazioni che cambiano a runtime, senza intervento umano (scrittura di nuovo codice) né ricompilazione completa. Implica il riuso del codice e la riduzione del tempo di sviluppo.
Nonostante ciò, questa tecnica ha alcune implicazioni: un metaprogramma deve poter effettuare a tempo di esecuzione alcune operazioni che in altri linguaggi sono effettuate solo a tempo di compilazione (ad esempio compilatore in ambiente runtime, type checking dinamico, …).

La forma più usata di metaprogramming è la *Reflection*, la quale è comune nei linguaggi dinamici e inoltre usata, anche se in modo molto limitato, in alcuni aspetti di altri linguaggi, ad esempio Java.
La Reflection è la capacità di un programma di eseguire elaborazioni (analizzare / modificare) la propria struttura e il proprio comportamento a runtime. Ad esempio in Java un programma in esecuzione può esaminare le classi da cui è costituito, i nomi e le signature dei loro metodi (Package java.lang.reflect).
La Reflection viene realizzata tramite un insieme di meccanismi suppletivi all'ambiente runtime. L'ambiente runtime deve essere in grado di

1. *Analizzare* costrutti di codice (classi, metodi) a runtime (*introspection*)
2. *Convertire* una stringa contenente il nome simbolico di un oggetto in un riferimento all'oggetto a runtime
3. *Valutare* a runtime una espressione (stringa) come se fosse una sequenza di nuovo codice.

Non tutti i linguaggi offrono tutti questi meccanismi.

Introspection

La Introspection è la capacità di un linguaggio (object oriented) di determinare tipo e proprietà di un oggetto (o di una classe) a tempo di esecuzione. Si basa sul type checking dinamico.
Viene utilizzato principalmente per ispezionare classi, attributi e metodi senza bisogno di conoscerne il nome a tempo di compilazione (hard-coded nel programma). Vi è inoltre una possibilità di istanziare oggetti e invocare metodi.
Ad esempio, in un processo si va ad individuare il tipo di una data classe, poi si individua la lista dei metodi della classe e infine si eseguono.

In Python ci sono meccanismi di auto-analisi (reflection) sulle classi, simili a quelli visti per Java. L'accesso alle classi e ai loro metodi viene effettuato senza avere nomi di entità hard-coded nel codice, il che permette flessibilità, manutentibilità e riuso.

Gli strumenti di Python sono:

- *__dict__* che è un attributo speciale di classi/moduli, contiene tutta la lista degli attributi che descrivono l'oggetto in questione. Viene usato per alterare o leggere gli attributi
- *locals()* è una funzione che ritorna il dizionario contenente le variabili definite nel namespace locale
- *global()* è una funzione che ritorna un dizionario che rappresenta il namespace globale del modulo
- *getattr(object, name)* è una funzione che ritorna il valore contenuto nell'attributo dell'oggetto specificato
- *import inspect* importa una libreria di Python che fornisce funzioni utili a ricavare informazioni da oggetti, moduli, classi, metodi, funzioni, traceback, frame objects e code objects
- *inspect.isfunction(object)* ritorna True se l'oggetto è una funzione Python che include funzioni create tramite l'espressione *lambda*

[*lambda in Python:* è una funzione anonima composta da una singola espressione che viene valutata quando viene chiamata la funzione. La sintassi per creare una funzione lambda è *lambda [parameters]: expression*]

locals() e globals() restituiscono un dictionary avente come chiavi gli identificatori e come valori i valori delle entità associate.

getattr(object, namespace) accede all'entità di nome (stringa) di object (istanza di classe). Se name è il nome di un attributo, restituisce il valore; se name è il nome di un metodo, restituisce il riferimento al metodo.

Invocazione di un metodo senza reflection:
obj = foo() #istanzio la classe tramite il costruttore

obj.hello() #chiamo il metodo

Invocazione di un metodo con reflection:
class_name = “foo”
method = “hello”
obj = globals()[class_name]()
getattr(obj, method)()

Recupero dei metodi di una classe

L’attributo speciale obj.__dict__ è il dizionario che contiene tutti gli attributi di un oggetto obj. Se object è una classe, __dict__ contiene tutti gli attributi e i metodi della classe:
c = globals()[“classe”] #riferimento alla classe
c.__dict__

{‘__doc__’: None, ‘f’:<function classe.f at 0x37b847a63c8>}

import inspect
inspect.isfunction(getattr(globals()[“classe”](), ‘f’))

Valutazione dinamica di espressioni

Il meccanismo di valutazione dinamica di espressioni ha la capacità di considerare a runtime il contenuto di una stringa come nuovo codice. Il nuovo codice può essere letto da file o generato a runtime sotto forma di stringa. L’espressione che si ha intenzione di valutare o eseguire viene memorizzata all’interno di una stringa, per espressione si può intendere:

- espressioni matematiche e logiche
- blocchi di codice
- funzioni

Successivamente si esegue o si valuta (interpretazione con restituzione del risultato) il contenuto della stringa.
Come vantaggio è possibile costruire del codice polimorfo, in grado di modificarsi e adattarsi a tempo id esecuzione, ad esempio una configurazione da file.
D’altra parte, lo svantaggio lo incontriamo se le stringhe rappresentanti il codice non sono ben controllate (inserimento attraverso form o lettura da file di configurazione), è possibile incorrere in rischi legati alla sicurezza.

Gestione dinamica delle istruzioni

Una *funzione anonima* è una funzione che non è legata ad un nome ed ha una signature. Vengono utilizzate attraverso un riferimento o passaggio di parametri e il loro uso tipico sta nel contenere funzionalità di uso a breve termine in un punto specifico del programma per specializzare il funzionamento.
Le funzioni anonime sono disponibili solo se il linguaggio supporta funzioni come first-class entities (non supportate da Java), ovvero funzioni assegnabili a variabili, passabili come argomenti e come valori di ritorno.

Un esempio di uso risiede nel passaggio sotto forma di argomento a una funzione di livello superiore per specializzarne il comportamento (ordinare delle entità).
Un esempio di funzione di livello superiore può essere una funzione di ordinamento per gestire diversi tipi di dato e ordinare secondo diversi criteri: in presenza di dati strutturati, ovvero composti da più elementi (tuple, liste, o classi) la funzione di ordinamento può agire su campi diversi del dato.
La funzione *sorted(iterable [, key][, reverse])* ritorna la lista ordinata. L'opzione *Key* rappresenta una funzione che ritorna la "chiave" su cui fare l'ordinamento (il confronto). *Reverse* è un valore booleano che imposta il tipo di ordinamento (default False).

Esempio di definizione:

```
dividi = lambda dividendo: dividendo / 5
print(dividi(10))
(risultato:) 2
g = lambda x,y:x**2+y
g(5,8)
(risultato:) 33
```

Uso di una funzione anonima in sorted:

```
students = [('jhonny', 'A', 25), ('jane', 'B', 17), ('daniel', 'C', 21)]
sorted(students, key = lambda x: x[2])
```

in questo modo vado a ordinare in base al secondo numero della tupla (considerando che iniziamo a contare partendo da 0)

[('vane', 'B', 17), ('daniel', 'C', 21), ('jhonny', 'A', 25)]

La funzione Key assume di ricevere in ingresso ogni elemento dell'iterabile e di ritornare il valore su cui effettuare il confronto

Altro esempio:

pairs = [(1, ‘one’), (2, ‘two’), (3, ‘three’), (4, ‘four’)]
sorted(pairs, key = lambda pair: pair[1])
(risultato:) [(4, ‘four’), (1, ‘one’), (3, ‘three’), (2, ‘two’)]

Gestione degli errori a runtime

Tutti i linguaggi dinamici prevedono un meccanismo per intercettare e gestire gli errori a runtime, si chiama meccanismo delle eccezioni software. È reso popolare da Java, presente in C#, Perl, Python, Ruby, PHP5, Javascript.
La maggior parte dei linguaggi adotta per la gestione degli errori un approccio basato su classi (Java, Python, …). Perl adotta un approccio più semplificato basato sulle istruzioni speciali (eval e die).

Il runtime environment (RE) prevede la definizione di una classe madre rappresentante la generica eccezione software, che contiene:

- Un identificatore dell’istanza di eccezione
- una rappresentazione dello stack corrente

Ciascuna anomalia a runtime viene associata ad una sottoclasse specifica della superclasse (in Java l’interfaccia Exception ha molteplici implementazioni, es. RuntimeException, SQLException, …). Le eccezioni possono essere sollevate:

- automaticamente, in seguito ad anomalie provocate da istruzioni a runtime (es. divisione per 0)
- manualmente, dal programmatore (throw, raise)

In caso di eccezione viene eseguito il codice di gestione; se non è presente si usa il gestore di default che stampa lo stack ed esce.

In Python, il costrutto try-except-finally è simile a quello try-catch-finally di Java:

```
try:
        <codice>
except Exception as e:
        <codice gestione>
except Exception as e2:
        <codice gestione 2>
.
.
.
except:
        <gestione anomalia di una qualunque eccezione>
else:
        <codice eseguito in assenza di eccezioni>
finally:
```

<codice eseguito alla fine del try-except, in ogni caso>

Per quanto riguarda il sollevamento manuale di una eccezione, in Python, si utilizza la parola chiave *raise*:
raise NameException(arg1, arg2)
NameException è il tipo di eccezione
(arg1, arg2) è la lista opzionale di argomenti passati

Per il recupero degli argomenti, *except NameException as inst:*
inst contiene l'istanza dell'eccezione sollevata. *Inst.args* è una tupla che contiene gli argomenti stampati

```
import traceback

traceback.print_exc() #stampa le eccezioni
```