

Scala Macros and Metaprogramming

Arianna Soriani

20 settembre 2024

Indice

1	Metaprogramming	3
1.1	Definizione	3
1.2	Scala 2 Metaprogramming	4
1.3	Scala 3 Metaprogramming	6
2	Macros	7
2.1	Definizione attraverso le 5W	7
3	Macros in Scala 2	10
3.1	Design	10
3.2	Hygiene	10
3.3	Step-by-step	11
3.4	Esempi di applicazioni	12
3.5	Altri Tipi di Macros	18
3.6	Le due facce di Scala macros	18
3.7	Vantaggi	20
3.8	Problemi	20
4	Macros in Scala 3	21
4.1	Obiettivi	21
4.2	Implementazione	21
4.3	Sintassi	21
4.4	Confronto di features	22
4.5	Transparently macros	23
4.6	Migrazione da Scala 2 a Scala 3	25
4.7	Migrazione da Scala 2 a Scala 3 Macros	25
4.8	Inline	26
4.9	Inline vs Macros	29
4.10	Macro = Inline + Quotes + Splice	31
4.11	Quoted package	34
4.12	Quotes '{...}'	35
4.13	Scala Quoted Type	39
4.14	Macro APIs	41
4.15	Level checking	43
4.16	Reflection	44
4.17	AST	45
4.18	Casi d'uso	48
4.19	Compiler e Bytecode	53
4.20	Conclusioni	54

1 Metaprogramming

1.1 Definizione

La **metaprogrammazione** è uno stile di programmazione popolare, nato negli anni '70 e '80, allo scopo di manipolare i *programmi come dati*.

In alcuni linguaggi, la differenza tra *programmazione* e *metaprogrammazione* non è così significativa.

- Nei linguaggi *a tipizzazione dinamica* (es. Python) la metaprogrammazione semplifica la manipolazione del programma attraverso l'ausilio di altro codice.
- Nei linguaggi *a tipizzazione statica* (es. Scala e Java) la metaprogrammazione è più limitata e meno comune. Diventa però molto utile per risolvere problemi di progettazione avanzati; è richiesta, dunque, maggiore esperienza e *maggior formalità* al fine di distinguere la manipolazione *a compile-time* da quella *a runtime*.

La metaprogrammazione si manifesta in diverse forme. Con il termine *reflection*, ci si riferisce all'introspezione del codice in fase di esecuzione (runtime), ad esempio quando si chiede un valore o un tipo di metadati propri. I metadati includono tipicamente dettagli sul tipo, sui metodi, sui campi, ecc. Pertanto, un programma è progettato in modo tale da leggere, analizzare, trasformare altri programmi o sè stesso mentre è in esecuzione.

La **Reflection** è una delle feature più importanti per ogni linguaggio di programmazione che vuole facilitare la metaprogrammazione.

Meccanismi specifici di metaprogrammazione, come le **Scala Macros**, lavorano come *constrained compiler plug-in* poichè manipolano l'Abstract Syntax Tree (AST) prodotto dal codice sorgente dopo il parsing. Le Macros sono invocate per manipolare l'AST prima delle fasi di compilazione che portano alla generazione di byte-code.

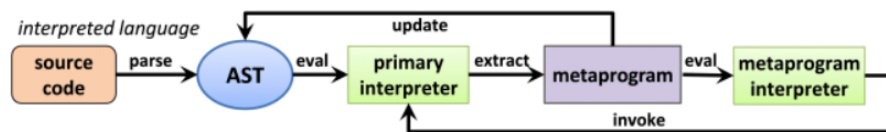


Figura 1: AST e Metaprogramming

1.2 Scala 2 Metaprogramming

Scala 2 ha un sistema di metaprogrammazione chiamato **Scalameta**, da sempre considerato sperimentale sebbene sia stato ampiamente utilizzato da autori di librerie per scenari avanzati.

La funzionalità cardine di Scalameta è l'accesso all'AST, abilitandone la lettura, l'analisi, la trasformazione e la generazione di programmi Scala ad un certo livello di astrazione.

L'**AST** è una rappresentazione del codice che lo rende più facile da analizzare.

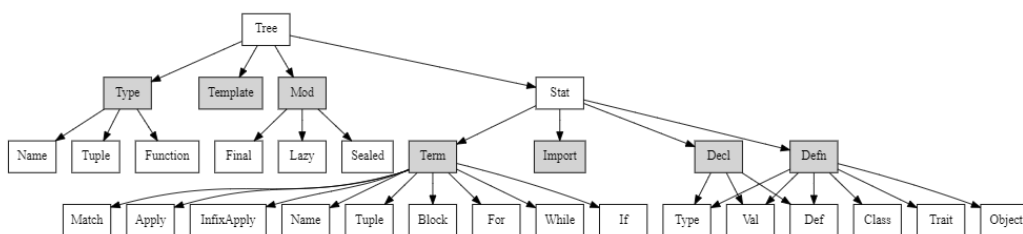


Figura 2: AST

I Scalameta trees sono *lossless*, ossia, rappresentano i programmi Scala in modo sufficientemente dettagliato da consentire il passaggio dal testo agli alberi, e viceversa. Gli alberi a sintassi lossless sono ideali per l'analisi dettagliata del source code, la quale risulta utile per diverse applicazioni, tra cui:

- Formatting
- Refactoring
- Linting
- Documentation tools

Scalameta è fornito insieme ad un *parser* per produrre i syntax trees a partire dal source code di Scala. Esistono due modalità primarie per costruire un AST: costruttori normali e *quasiquotes*, utilizzati da Scalameta.

Quasiquotes sono interpolatori di stringhe che si espandono a compile-time in normali chiamate al costruttore.

```
println(q"function(argument)".structure)
```

Figura 3: Quasiquotes

Le funzionalità principali di Scalameta sono:

1. **Parsing e manipolazione ASTs:** Scalameta fornisce strumenti per parsare il codice Scala in un Abstract Syntax Tree (AST) che può essere manipolato a piacere.
2. **Quasiquotes:** Permette la manipolazione di ASTs, attraverso l'interpolazione di stringhe, simile alle Macro di Scala 2.
3. **Analisi semantica:** Scalameta supporta l'analisi semantica, permettendo di risolvere simboli e tipi nel codice sorgente.
4. **Compatibilità:** È progettato per essere compatibile con diverse versioni di Scala, riducendo i problemi legati alla migrazione tra versioni del linguaggio.

Nonostante questo, però, viene tutt'ora considerata sperimentale. Presenta, inoltre, diverse limitazioni, difficilmente sormontabili:

1. **Complessità:** Nonostante sia più stabile, Scalameta introduce una curva di apprendimento piuttosto ripida rispetto alle Macro tradizionali di Scala 2.
2. **Performance:** Manipolare l'AST può essere oneroso in termini di performance, specialmente per progetti di grandi dimensioni.
3. **Supporto limitato per Scala 3:** Anche se Scalameta è molto potente per Scala 2, il suo supporto per Scala 3 è in fase di sviluppo e può mancare di alcune funzionalità rispetto alle nuove API di metaprogrammazione introdotte in Scala 3.
4. **Ecosistema limitato:** Anche se esistono alcune librerie costruite su Scalameta, non tutte le librerie di terze parti lo supportano, il che potrebbe limitare l'integrazione in progetti esistenti.

In sintesi, Scalameta è una potente libreria per fare metaprogrammazione in Scala con maggiore stabilità e compatibilità rispetto alle Macro tradizionali, ma richiede una buona conoscenza delle tecniche di manipolazione dell'AST e potrebbe non essere adatta a tutti i contesti, specialmente con l'emergere delle nuove funzionalità di Scala 3.

1.3 Scala 3 Metaprogramming

La documentazione di Scala 3 metaprogramming descrive 5 features fondamentali per il supporto della metaprogrammazione:

1. **Inline:** Il modificatore *inline* indica al compilatore di effettuare l'inline della definizione al momento di utilizzo. Inlining riduce l'overhead delle invocazioni di metodi e funzioni e dell'accesso ai valori, ma può largamente espandere la dimensione globale del byte-code, se la definizione è usata in diversi punti. Tuttavia, quando è usato insieme a clausole condizionali e di match che includono costanti a compile-time, l'inlining rimuove i branches non utilizzati. Inlining appare prima di *typer phase* nel processo di compilazione, così che la logica possa essere usata nelle fasi seguenti (macro expansion).
2. **Macros:** Una combinazione di *quotation*, dove sezioni di codice sono convertite in una tree-like data structure, e *splicing*, che funziona in senso opposto, convertendo le quotation nuovamente in codice. Usato con inline così che le macros siano applicate a compile-time. Scala 3 introduce un nuovo *macro system*, non più considerato sperimentale. Sostituire Scala 2 macros con l'implementazione in Scala 3 è una tra le sfide più grandi che alcuni responsabili della manutenzione delle librerie devono affrontare per affrontare la migrazione.
3. **Staging:** L'analogo costruttore di codice delle macro a runtime. Staging usa anche quotes e splices, ma non inline. Il termine staging deriva dall'idea che manipolare il codice a runtime rompe l'esecuzione in stage multipli, mescolando fasi di processamento normale con quelle di metaprogramming.
4. **TASTy reflection:** TASTy è la rappresentazione intermedia generata dai compilatori Scala 3. Abilita un'introspezione più ricca del codice, una migliore interoperabilità tra moduli compilati con diverse versioni di Scala e altri benefici. TASTy reflection produce un AST tipato (da cui il nome), che è una rappresentazione "white-box" del codice, contro la "black-box" view fornita dalle quotations.
5. **TASTy inspection:** Quando gli AST sono serializzati in file binari, forniscono l'estensione *.tasty*. TASTy inspection fornisce funzionalità per ispezionare i contenuti di questi files.

Queste features sono completate da altri costrutti come match types e by-name context parameters.

E' buona norma, però, utilizzare i costrutti propri della metaprogrammazione per risolvere problemi di progettazione *solo* come ultima risorsa.

2 Macros

2.1 Definizione attraverso le 5W

WHO

Le Macro di Scala si sono ispirate a concetti di metaprogrammazione presenti in altri linguaggi (es. Lisp). Una parte significativa dell'ispirazione e dello sviluppo iniziale delle Macro di Scala è attribuita a *Eugene Burmako*, che ha lavorato al progetto mentre era studente presso l'Università di San Pietroburgo. Il progetto era originariamente concepito come un esperimento divertente, ma si è evoluto in una funzionalità stabile che è stata inclusa in Scala 2.10. Burmako, studente di *Martin Odersky*, noto professore all'EPFL e creatore del linguaggio Scala, presentò le macros attraverso un primo semplice sketch.

WHEN

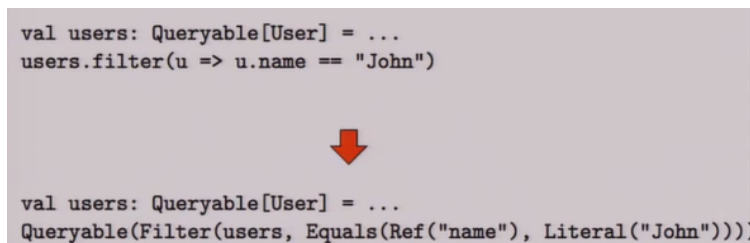
Le macros in Scala sono state introdotte per la prima volta in Scala 2.10, rilasciato nel 2013. Questa funzionalità ha consentito di eseguire metaprogrammazione, permettendo ai programmatori di scrivere codice che genera o manipola altro codice durante la compilazione.

WHY

Aumenta sempre di più la richiesta da parte delle aziende dell'utilizzo della metaprogrammazione.

Un esempio di progetto in cui si sono rivelate indispensabili è **Slick project**.

- *Scala Language-Integrated Connection kit*.
- Le Macro di Scala sono state utilizzate intensivamente nello sviluppo di librerie come Slick, che mirano a fornire un'interfaccia più sicura e tipizzata per interagire con i database. Slick, infatti, sfrutta le Macro per generare query SQL tipizzate e sicure a tempo di compilazione, eliminando molti degli errori comuni legati alla gestione diretta delle stringhe SQL.
- L'idea era avere *queries* integrate al linguaggio, così che le persone potessero scrivere codice come:



```
val users: Queryable[User] = ...
users.filter(u => u.name == "John")

↓

val users: Queryable[User] = ...
Queryable(Filter(users, Equals(Ref("name"), Literal("John"))))
```

Figura 4: Slick

Ossia, scrivendo query con codice Scala "normale", senza ulteriori ausili. Le query venivano così tradotte in maniera trasparente in strutture dati più specifiche.

Inizialmente, lo stesso Ordensky era scettico all'uso delle Macro a causa del complesso bagaglio teorico e implementativo che si portano dietro. E' da allora che nasce l'idea di semplificarle quanto più possibile.

WHAT

Attraverso le Macro è possibile scrivere metodi che vengono eseguiti a tempo di compilazione; questi metodi possono generare codice che potrà essere utilizzato normalmente.

- Le Macro realizzano *astrazioni testuali*
- Il compilatore espande questi snippets a *compile-time*
- Il programmatore definisce snippets come **normali funzioni Scala**

WHERE

Le Macro sono diventate strumenti molto utili in tantissimi contesti. Largamente utilizzati in:

- Diverse popolari librerie
- In moltissime industrie
- Nell'ambito della ricerca
- Molti costrutti di Scala si basano sull'ausilio delle Macro

Le Macro furono ufficialmente introdotte in Scala 2.10. Questa versione ha permesso ai programmatori di definire Macro a livello di funzioni e tipi, utilizzando l'API *scala.reflect*.

Questa API permetteva alle Macro di analizzare e trasformare alberi sintattici durante la fase di compilazione realizzando, così, potenti tecniche di generazione del codice e validazione a tempo di compilazione.

Evoluzione

Dopo Scala 2.10, le Macro sono rimaste una funzionalità sperimentale nelle versioni successive di Scala, con l'obiettivo di esplorare ulteriormente le potenzialità di questa tecnica.

Le Macro hanno continuato ad evolversi e, con l'introduzione di Scala 3 (Dotty), il *macro system* è stato completamente rinnovato; ora, si basa su principi differenti, con un focus maggiore su sicurezza e robustezza. Le Macro sono diventate in poco tempo uno strumento indispensabile per la programmazione più complessa, senza cui ora è difficile realizzare diverse applicazioni.

Questa loro rapida e "semplice" espansione è dovuta alle loro peculiarità base con i costrutti nativi di questo linguaggio:

- Si affidano a concetti familiari come chiamate a metodi tipizzati. Non c'è infatti molta differenza tra metodi "normali" e le Macro
- Potenziano in modo trasparente le funzionalità rappresentate con le chiamate ai metodi. Di seguito un elenco di alcune feature che catturano benefici da modifiche a compile-time ottenute attraverso l'utilizzo delle Macro:

1. **Fields**
2. **Application**
3. **Pattern matching**
4. **Implicits**
5. **For comprehensions**
6. **String interpolation**
7. **Dynamic**
8. ...

3 Macros in Scala 2

3.1 Design

Le Macro, come anticipato, necessitano di un design che deve essere il più minimalista possibile, a costo di abbandonare certe feature; *deve essere semplice* per coesistere in Scala.

Nella loro prima implementazione, perciò, vengono definite come:

Just a single feature: unhygienic expansion of typed method calls.

Questo significa che nativamente le Macro sono state definite come "semplici" espansioni non-higiene a chiamate a metodi tipizzati. Le Macro vengono definite attraverso le *def* di Scala e l'ausilio della *reflection*.

Altri costrutti, quali *quasiquotes* e *hygiene*, vengono costruiti sopra questa struttura più semplice sottostante.

3.2 Hygiene

Nella programmazione con Macro, il termine "*hygiene*" si riferisce al meccanismo che impedisce conflitti di nomi o catture indesiderate di variabili durante la generazione di codice. Questo concetto è particolarmente importante quando le Macro generano del codice che introduce nuovi identificatori o utilizza quelli esistenti.

Problema della non-higiene

Quando una Macro espande il codice a call-side, potrebbe accadere che alcuni identificatori, generati da quest'ultima, entrino in conflitto con quelli già esistenti nel codice dell'utente, o viceversa. Ad esempio, se la Macro introduce una variabile con lo stesso nome di una variabile già presente nel context in cui viene espansa, potrebbe verificarsi un conflitto, causando risultati imprevedibili. Il concetto di hygiene è stato sviluppato per evitare questi problemi di conflitto. Una Macro "hygiene" evita la cattura accidentale di variabili o identificatori attraverso l'uso di tecniche, come la rinominazione automatica dei simboli. Ciò significa che le variabili definite all'interno della Macro sono rese "locali" e non entrano in conflitto con quelle definite all'esterno.

Le tecniche utilizzate sono:

1. **Isolamento dei Nomi:** Le variabili definite all'interno della Macro non devono entrare in conflitto con quelle esistenti nel contesto in cui essa viene chiamata. Questo è spesso ottenuto generando nomi unici per le variabili interne (con *gensym* in Lisp, o *freshName* in Scala).
2. **Cattura e Sostituzione:** Le Macro hygiene evitano la cattura accidentale di variabili che non sono esplicitamente passate alla Macro. Ad esempio, se una Macro utilizza un nome di variabile *x* che è anche

utilizzato a livello call-side, la Macro hygiene sostituirà `x` con un nome unico, al fine di evitare conflitti.

3. **Espansione Sicura:** Il codice generato dalla Macro deve essere inserito a call-side in modo che le sue dipendenze siano correttamente risolte e non ci siano effetti collaterali indesiderati dovuti all'espansione.

Le Macro in Scala 2 non sono hygiene in modo nativo. Questo significa che lo sviluppatore è responsabile di garantire che non ci siano conflitti. A differenza di Scala 2, Scala 3 introduce un nuovo macro system che è hygiene per default.

3.3 Step-by-step

```
class Queryable[T](val query: Query[T]) {  
  macro def filter(p: T => Boolean): Queryable[T] = <[  
    val liftedp = ${lift(p)}  
    Queryable(Filter($this.query, liftedp))  
  ]>  
}  
  
val users: Queryable[User] = ...  
users.filter(u => u.name == "John")
```

Figura 5: Macros def example

Quello mostrato in figura è il primo sketch presentato da Burmako alla presentazione delle Macro in Scala 2.

In questo codice, la funzione Macro verrà espansa altrove: si parla di trasferimento di codice da *def-side* a *call-side*. In particolare:

- Viene chiamata la funzione *filter* (macro).
- Viene eseguita la *def* mostrata in figura.
- Il risultato viene inserito a call-side.
- Perciò, tutta la componente macro viene "spostata" dalla definizione all'invocazione.
- Il tutto, a compile-time.

Da questa prima "bozza" sono state apportate alcune migliorie che sono state poi rilasciate con la prima versione di Scala Macro, in Scala 2.10.

Di seguito, un esempio di definizione di una Macro, nella sua prima implementazione:

```
class Queryable[T](val query: Query[T]) {  
  def filter(p: T => Boolean) = macro Macros.filter[T]  
}  
  
object Macros {  
  def filter[T: c.WeakTypeTag]  
    (c: Context { type PrefixType = Queryable[T] })  
    (p: c.Expr[T => Boolean]) =  
    c.universe.reify {  
      val liftedp = lift(p).splice  
      new Queryable(Filter(c.prefix.splice.query, liftedp))  
    }  
}
```

Figura 6: Filter Macros

L'idea perciò è quella di:

1. Scrivere Macro come **normali funzioni Scala**, utilizzando, quindi, le *def*, le *type signature* e nella parte destra, dove di solito viene incluso il corpo della funzione, viene introdotta la keyword **macro**, seguita dal *riferimento alla funzione*, che sarà eseguita a *compile-time*.
2. La funzione prenderà l'AST e genererà nuovo codice.
3. Potrà utilizzare quasiquotes e hygiene attraverso *reify*. Reify è una funzione, una macro, che prende in input un template e genera il corrispondente AST, consentendo l'inserimento dinamico di codice all'interno del template attraverso la chiamata alla funzione *splice*.
4. Lo scopo è quello di scrivere codice minimale.
5. *Call-side*: rappresenta il punto in cui la macro viene effettivamente espansa ed eseguita.

3.4 Esempi di applicazioni

Di seguito vengono presentati alcuni esempi di come le macro inficiano sulla semplicità della programmazione e sul potenziamento di quest'ultima.

1. Il primo esempio mostra come le macro possano aggiungere espressività al codice delle chiamate a metodi.

```

val futureDOY: Future[Response] =
  WS.url("http://api.day-of-year/today").get

val futureDaysLeft: Future[Response] =
  WS.url("http://api.days-left/today").get

futureDOY.flatMap { doyResponse =>
  val dayOfYear = doyResponse.body
  futureDaysLeft.map { daysLeftResponse =>
    val daysLeft = daysLeftResponse.body
    Ok(s"$dayOfYear: $daysLeft days left!")
  }
}

```

Figura 7: From synch to async

Nell'immagine, viene riportato il codice necessario per trasformare un programma sincrono in un programma asincrono; questo procedimento non è semplice perchè richiede di gestire manualmente callbacks, futures, introdurre var temps, ecc.

Ovviamente, la complessità aumenta esponenzialmente con l'aumentare delle richieste.

Le *scala/async macros* effettuano la trasformazione automaticamente!

```

def async[T](body: => T): Future[T] = macro ...
def await[T](future: Future[T]): T = macro ...

async {
  val dayOfYear = await(futureDOY).body
  val daysLeft = await(futureDaysLeft).body
  Ok(s"$dayOfYear: $daysLeft days left!")
}

```

Figura 8: Empowered method calls with Macros

- E' possibile usare **async** come macro che orchestra la trasformazione da codice sincrono a codice asincrono
 - Questo non significa realizzare una nuova fase nella pipeline del compilatore; ma consiste nel mero utilizzo di una macro, ossia di una funzione di libreria
2. Il secondo esempio riguarda la capacità delle macro di potenziare l'interpolazione di stringhe.

```
scala> val x = "42"
x: String = 42

scala> "%d".format(x)
j.u.IllegalArgumentException: d != java.lang.String
  at java.util.Formatter$FormatSpecifier.failConversion...

scala> f"$x%d"
<console>:31: error: type mismatch;
  found   : String
  required: Int
```

Figura 9: String Interpolation

Il tipo String viene da sempre considerato come qualcosa di "bad", non tipato; un costrutto non-safe.

Ma con le macro non è più così! L'interpolazione di stringhe diventa simile ad un printf, statico e tipato. Quindi a compile-time se si specifica erroneamente il modifier o se viene effettuato lo splice su un tipo non corretto, si ottiene un errore a compile-time e tutto questo grazie alle macro.

```
implicit class Formatter(c: StringContext) {
  def f(args: Any*): String = macro ...
}

val x = "42"
f"$x%d" // rewritten into: StringContext("", "%d").f(x)

↓

{
  val arg$1: Int = x // doesn't compile
  "%d".format(arg$1)
}
```

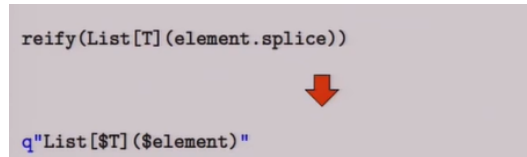
Figura 10: Empowered Interpolation

N.B: String interpolation altro non è che zucchero sintattico a chiamate di metodi.

- f è una macro. Quando il compilatore incontra una chiamata a f , distribuisce la chiamata ad un'estensione al metodo f .
- In generale, f può essere:
 - un metodo "normale"; tutto viene processato a runtime

- una macro; essa inserisce una type description in punti strategici che consentirà al compilatore di emettere errori di incorrettezza di tipo
3. Nel terzo esempio, viene affrontato il problema della *string interpolation*. Attraverso l'interpolazione di stringhe è possibile, in generale, incorporare all'interno di Scala, altri linguaggi arbitrari e, attraverso le macro, è possibile processare queste parti a compile-time. E' possibile persino incorporare Scala in Scala.
- A tale scopo, il metodo *reify* non funziona molto bene; richiede che i suoi argomenti siano staticamente tipati.
- In alcuni casi, come la scrittura di un template da utilizzare all'interno in una macro, questo metodo richiede di effettuare immediatamente il type-check, il che comporta il fatto di non poter contenere istanze di variabili free.
- Inoltre, questo metodo non è adatto a sistemi più complessi che incorporano parti di altri linguaggi poichè, se separate, queste perdono di significato.

Con le macro questo problema si risolve! Vengono utilizzate le *quasiquotes* che non si preoccupano dei tipi; esse rendono possibile l'inserimento di codice di altri linguaggi all'interno di Scala. L'esempio illustrato è un esempio cardine del minimalismo delle Macro in Scala.



```
reify(List[T](element.splice))  
↓  
q"List[$T]($element)"
```

Figura 11: Empowered Interpolation

4. Nel quarto esempio, si vuole realizzare una libreria di serializzazione, basata sulla funzione *pickle*, la quale prende in input un argomento da serializzare.

```
trait Pickler[T] {  
  def pickle(picklee: T): Pickle  
}  
  
def pickle[T](picklee: T)(implicit p: Pickler[T]): Pickle
```

Figura 12: Serialization function

Si vuole inoltre rendere questa funzione estensibile.

Si possono usare le *type classes* per rendere le parti da trasferire trasferibili dalla funzione.

Si crea dunque il type class speculare che ha il metodo *pickle* così che

l'utente possa implementare la strategia per utilizzare il metodo. L'utente ha completo controllo su cosa può serializzare e come.

```
def pickle[T](picklee: T)(implicit p: Pickler[T]): Pickle

implicit val IntPickler = new Pickler[Int] {
  def pickle(picklee: Int): Pickle = ...
}
pickle(42) // you write
pickle(42)(IntPickler) // you get

pickle("42") // compilation error
```

Figura 13: Example Pickle usage

Per fare questo, come mostrato in figura, vengono utilizzati gli implicit.

Grazie a questo ausilio non è necessario fornire il parametro "implicit" poichè il compilatore è in grado di rappresentarlo da solo. Perciò, invocando, ad esempio, la funzione `pickle` di `Int`, grazie alla keyword `implicit` il compilatore capisce la necessità di fornire un `pickle` di `Integer`; viceversa, se gli viene indicato un tipo non riconosciuto restituisce un errore ben formato. Questa modalità fornisce un certo livello di sicurezza.

Implementare soluzioni ai problemi di serializzazione, è una tra le richieste più scomode che possano capitare.

Ad esempio, alla richiesta di serializzare un "Person" si potrebbe pensare di serializzare ogni componente del data type e di inserirlo all'interno di un programma:

```
case class Person(name: String, age: Int)

implicit val personPickler = new Pickler[Person] {
  def pickle(picklee: Person): Pickle = {
    val p = new Pickle
    p += ("name" -> p.name.pickle)
    p += ("age" -> p.age.pickle)
    p
  }
}
```

Figura 14: Example Pickle usage

Questo funziona per poche istanze; già con alcune decine o centinaia potrebbe essere un problema.

Con le macro questi problemi si risolvono! Il boilerplate può essere generato semplicemente attraverso l'ausilio delle macro, come segue:

```
implicit val personPickler = Pickler.generate[Person]
```

Figura 15: Vanilla Macros

Il bytecode finale risultante è esattamente lo stesso che verrebbe prodotto nel caso precedente.

L'utilizzo delle macro risolve quindi il problema del boilerplate, senza inficiare sulle performance.

Una delle feature più importanti delle Macro è la capacità di *interazione con altri linguaggi* e la *modalità in cui mappano nelle relative chiamate a metodi*.

E' possibile dichiarare metodi come impliciti, ma se il metodo è una macro, non è necessario scriverlo!

```
trait Pickler[T] { def pickle(picklee: T): Pickle }  
  
object Pickler {  
  implicit def materializePickler[T]: Pickler[T] = macro ...  
}
```

Figura 16: Implicit Macros

E' sufficiente scrivere questa implementazione una volta e non preoccuparsi più del problema della serializzazione.

Quando il compilatore necessita di risolvere un parametro "implicit" nella chiamata ad una funzione, segue il lexical scope, cerca ciò che è dichiarato con il flag implicit e prova ad effettuarne il match con il tipo.

Con le macro è leggermente più complicato. La macro viene inserita all'interno di un companion object e l'utente non ha più bisogno di preoccuparsi di niente.

```
pickle(person)  
  
↓  
  
pickle(person)(Pickler.materializePickler[Person])  
  
↓  
  
pickle(person)(new Pickler[Person]{ ... })
```

Figura 17: Implicit Macros Call

Attraverso una normale chiamata a funzione, il compilatore trova un *implicit*, il quale è una macro, perciò la espande e gestisce il tutto in autonomia, senza l'inclusione dell'utente!

3.5 Altri Tipi di Macros

1. Untyped Macro

Sopprime il typechecking degli argomenti prima dell'espansione. Perciò, invece di specificare il tipo nella signature della macro, è possibile utilizzare: `_`.

La *Type safety* non viene sovvertita; nelle espansioni viene effettuato il typecheck come al solito.

2. Type Macro

Finora sono state introdotte le macro come espansioni di metodi; è possibile espandere anche i tipi, *type*.

3. Macro Annotations

Come le *def macro* espandono chiamate a funzioni, questo tipo di macro espandono *definitions*.

3.6 Le due facce di Scala macros

*"Se riesci a nascondere il comportamento della macro sotto la sua type signature, allora è una **Blackbox macro** poichè la sua implementazione può essere trattata come una scatola nera. Se invece non è possibile, allora è una **Whitebox macro**"*

1. Blackbox macros:

fedelmente conformi alle loro type signatures.

Classiche macro presenti da Scala 2.10. Esse offrono:

- **Stabilità dei Tipi:** Le blackbox macros garantiscono che il tipo dell'espressione rimanga lo stesso come se la macro non fosse stata applicata. Questo significa che la macro non influenza il processo di typechecking. Il compiler controlla i tipi *prima* dell'espansione della macro, e questi tipi rimangono stabili.
- **Influenza Limitata:** Poiché il tipo viene determinato prima dell'espansione della macro, le blackbox macros non possono cambiare il tipo apparente di un'espressione. Sono limitate a generare codice che rispetta i vincoli di tipo già noti.
- *Esempio d'Uso:* Le blackbox macros sono tipicamente usate per generare codice boilerplate, come l'inlining o l'ottimizzazione del codice durante la compilazione, dove i tipi sono già noti e non devono cambiare.

2. Whitebox macros:

non possono avere signatures nello Scala's type system. Sono i nuovi tipi di Macro introdotti, ma sono troppo potenti; rompono la vera logica di Scala e sono troppo malleabili.

- **Flessibilità dei Tipi:** Le whitebox macros possono influenzare il tipo del codice che generano. Il tipo dell'output della macro viene determinato *dopo* l'espansione della macro, il che significa che la macro stessa può produrre codice che ha un tipo diverso da quello che ci si aspettava inizialmente.
- **Maggiore Potenza e Flessibilità:** Poiché le whitebox macros possono alterare il tipo di un'espressione, permettono una metaprogrammazione più potente, come la generazione di tipi polimorfici o dipendenti basati sull'input della macro. Tuttavia, questa potenza comporta una maggiore complessità.
- *Esempio d'Uso:* Le whitebox macros sono utili nei casi in cui è necessario generare codice con un tipo che dipende dagli input della macro, o quando è necessario eseguire trasformazioni complesse che influenzano il sistema dei tipi.
- **"Non è vero Scala".** Questa espressione spesso si riferisce al fatto che le whitebox macros possono rompere alcune delle garanzie che il sistema dei tipi di Scala fornisce come la sicurezza dei tipi e la prevedibilità.

Possono infatti comportare:

- **Tipi Imprevedibili:** Poiché le whitebox macros possono cambiare il tipo atteso delle espressioni, possono portare a situazioni in cui i tipi non sono chiari o prevedibili semplicemente leggendo il codice, rendendo più difficile ragionare sullo stesso.
- **Mancato supporto agli Strumenti e all'Ecosistema:** Le whitebox macros possono complicare lo sviluppo di strumenti come IDE o altri strumenti di analisi statica, poiché questi ultimi devono capire ed espandere correttamente le macro per analizzare il codice con precisione.
- **Incoerenza:** Le whitebox macros possono introdurre incoerenze che non sono tipicamente presenti nel codice "puro" di Scala, portando a codice che si comporta in modo inatteso, che può essere fonte di confusione ed errori.

Al contrario, le blackbox macros mantengono il controllo standard dei tipi e non permettono comportamenti che alterano questi ultimi, rendendole più prevedibili e facili da comprendere, allineandosi meglio al paradigma di programmazione tipato di Scala.

3.7 Vantaggi

Le Macro hanno raggiunto in poco tempo un grandissimo successo. Questo grazie a diversi vantaggi che le accompagnano:

- Gli utenti che utilizzano funzioni macro non si accorgono della loro esistenza! Infatti, esse utilizzano le feature di Scala come di consueto. Consistono in normali chiamate a metodi tipati.
- Le features di Scala, a loro volta, risultano internamente migliorate grazie all'utilizzo delle Macro; questo grazie al fatto che molte feature di design e caratteristiche del linguaggio Scala, vengono mappate in chiamate a metodi

3.8 Problemi

Con le Macro di Scala 2 si sono riscontrati, da sempre, diversi problemi di design, tra cui:

- **Portabilità:** diventa difficile trasferirli al nuovo compilatore.
- **Accoppiamento con il compilatore:** da cui il problema precedente. L'architettura delle Macro di Scala 2 sono finemente accoppiate con l'architettura del compilatore di Scala 2, che rende impossibile il riutilizzo così com'è del sistema.
- **API complesse:** derivato sempre dal punto precedente. Il forte accoppiamento, rende l'API piuttosto complessa.
- **Unhygienic:** di default, sono Unhygienic.
- **Sperimentale:** struttura considerata, da sempre, sperimentale.
- **Forte conoscenza:** l'utilizzo delle Macro di Scala 2 richiede un'approfondita conoscenza del linguaggio e dei processi di compilazione interni.

4 Macros in Scala 3

4.1 Obiettivi

1. **API indipendente dal compilatore:** risolvendo quindi i problemi di interoperabilità.
2. **API che scala in complessità:** la complessità nella scrittura di una Macro in Scala 3 si adatta alla complessità del codice che bisogna implementare.
3. **TASTy come layer di compatibilità:** TASTy file format viene utilizzato come rappresentazione binaria di programmi Scala di alto livello in Scala 3; viene utilizzato inoltre come layer di compatibilità tra il compilatore e le Macro, così da essere in grado di trasferirlo alle prossime versioni di Scala.
4. **Semplicità, Sicurezza, Robustezza.**

4.2 Implementazione

Il precedente, e sperimentale, macro system era utilizzato per implementare soluzioni intelligenti per problemi architetturali difficili in alcune librerie avanzate. Tuttavia, per usarlo, era necessaria una conoscenza molto approfondita. Il nuovo Scala 3 macro system offre quasi lo stesso livello di potenza, ma in maniera più accessibile.

Ora le Macro sono definite attraverso l'ausilio di due operazioni complementari: *quotation* e *splicing*.

- **Quotation:** converte un'espressione di codice in un AST tipizzato che rappresenta l'espressione, un'istanza del tipo `scala.quoted.Expr[T]`, dove T è il tipo dell'espressione. Per il tipo T stesso, quotation ritorna una type structure per quest'ultimo, di tipo `scala.quoted.Type[T]`. Questi alberi e queste type structure possono essere manipolate per costruire nuove espressioni e tipi.
- **Splicing:** vanno nella direzione opposta, convertendo un AST Expr[T] in un'espressione di tipo T e convertendo una type structure Type[T] in un tipo T.

4.3 Sintassi

- La sintassi per un'espressione di *quotation* è: `'{...}`
- Per i tipi, invece, è: `'[...]`
- La sintassi per lo *splicing* è: ``${...}`, analogo all'interpolazione di stringhe

Gli **identificatori** possono essere: **quoted** (`'expr`) o **spliced** (``${expr}`).

4.4 Confronto di features

E' possibile confrontare le features presenti nelle due implementazioni di macro (Scala 2 vs Scala 3).

Sinteticamente, è possibile visualizzarle di seguito:

	Scala 2	Scala 3
blackbox	✓	✓
whitebox	✓	✓
untyped	■	✓
'{ ... }'		✓
@annotation	✓	□

Figura 18: Macros features

1. **Blackbox:** Supportate da entrambe le versioni di macro, ma con diverso nome; in scala 3 Macro sono chiamate **inline macros**.
2. **Whitebox:** Supportate da entrambe le versioni di macro, ma con diverso nome; in scala 3 Macro sono chiamate **transparently macros**.
3. **Feature eliminate:**
 - Possibilità di tipizzare parte del codice durante l'uso, dopo l'espansione della macro. Questo rende difficile l'utilizzo da parte degli utenti perchè richiede di ragionare all'interno della macro, sull'implementazione stessa. Viene perciò fornita un'astrazione più semplice, con garanzie più forti, che aiuterà a costruire macro più semplici nel futuro.
4. **Annotation macros:** Le annotation macros sono una forma di macro che permette di eseguire trasformazioni o manipolazioni sul codice usando annotazioni. Le annotation in Scala (e in molti altri linguaggi di programmazione) sono metadati che possono essere aggiunti al codice per fornire informazioni aggiuntive a livello di compilazione o a runtime. In Scala 2, esistono le annotation macros sperimentali mentre, al momento, non sono presenti nella versione di Scala 3 ma è uno degli obiettivi prossimi di progettazione per le versioni successive del compilatore Dotty. Nel frattempo, è possibile utilizzare un plug-in che ne permette l'inserimento all'interno del compilatore, in maniera molto rapida.

4.5 Transparency macros

Come accennato precedentemente, con le macro non è possibile generare nuovi simboli che siano visibili all'utente, siano essi funzioni, variabili o classi; non è possibile neanche aggiungere metodi ai corpi di classi esistenti e non c'è modo, per ora, di aggirare il problema.

Tuttavia, è possibile far comportare un tipo esistente come se fosse qualcosa'altro e decorarlo dinamicamente con nuove funzionalità, ottenendo un'esperienza utente molto simile a quella di aver generato una nuova classe da zero.

Scala offre molti modi per estendere classi con nuovo comportamento senza ricorrere all'ereditarietà, come funzioni di estensione, istanze di `typeclass`, funzioni `unapply`, `typeTests` e così via, molte delle quali possono essere combinate con le macro da un programmatore ingegnoso. I tre ingredienti chiave sono: *refinements* strutturali, invocazioni dinamiche e *transparent inline defs*.

La nuova parola chiave **transparent inline** di Scala 3 è un modo per ottenere questo risultato. Aggiungendo la parola chiave `transparent` a un metodo `inline`, il tipo di ritorno del metodo viene ristretto a un tipo più specifico rispetto a quello dichiarato dopo l'espansione nel punto di chiamata, a seconda del percorso del codice scelto (staticamente):

```
inline def foo(b: Boolean): Any =
  if b then 1 else "hello"

val x1: Any = foo(true) // x1 == 1 but inferred type is declared Any
val x2: Any = foo(false) // x2 == "hello" but inferred type is declared Any

transparent inline def transFoo(b: Boolean): Any =
  if b then 1 else "hello"

val y1: Int = transFoo(true) // x1 == 1 and inferred type narrowed to Int
val y2: String = transFoo(false) // x2 == "hello" and inferred type is narrowed to String
```

Figura 19: Inline and Transparent Inline

In questo caso, il corpo del metodo `transFoo` viene espanso nel punto di chiamata e il compiler può eliminare staticamente l'espressione `if` attraverso l'ottimizzazione *constant-folding*. Ciò che rimane del corpo del metodo `transparent inline`, è o `1` o `"hello"`; il compilatore può quindi restringere il tipo di ritorno a `Int` o `String`. Ovviamente, il restringimento del tipo può essere eseguito solo se un tipo più specifico è effettivamente noto a tempo di compilazione grazie all'uso di costanti e `inline if/inline match`.


```

@Test def testType()=
  val instance = new transparent_inline()
  // Verifica i tipi
  assertTrue(instance.x1.isInstanceOf[Any])
  assertTrue(instance.x2.isInstanceOf[Any])
  assertTrue(instance.y1.isInstanceOf[Int])
  assertTrue(instance.y2.isInstanceOf[String])

```

Figura 20: Test different Types

Se il metodo `transparent inline` è una macro, questo restringimento del tipo di ritorno può essere *sempre eseguito* poiché la macro viene eseguita a tempo di compilazione:

```

class DynamicFoo extends Dynamic {
  transparent inline def selectDynamic(name: String): Any = ${ selectDynamicImpl('name) }
}

// must be public due to compiler bug
def selectDynamicImpl(name: Expr[String])(using q: Quotes): Expr[Any] = {
  import q.reflect.{*, given}
  name.valueOrAbort match
    case "a" => '{ 1: Int }
    case "b" => '{ 2.34f: Float }
    case "c" => '{ "hello": String }
    case n => report.errorAndAbort(s"no field named $n", name)
}

```

Figura 21: Transparent Inline macro

Un significativo svantaggio dei metodi `transparent inline` è che il loro type-checking dipende dall'inlining (ovviamente) e quindi sono soggetti a problemi di ordinamento con altri metodi inline; i risultati dei metodi `transparent` vengono tipizzati solo dopo l'inlining. Allo stesso tempo, viene effettuato il typecheck delle funzioni inline nel punto in cui sono dichiarate, e non dove viene effettuato l'inlining. Se una funzione `transparent inline` viene chiamata all'interno di un'altra funzione inline, avrà comunque il suo tipo dichiarato e non il tipo ristretto quando il sito di chiamata viene controllato a livello di tipi. Questo significa che tutte le macro che si basano su `transparent inline` non funzioneranno all'interno di metodi inline.

I metodi `transparent inline` sono *"whitebox"*, il che significa che dettagli, normalmente invisibili dell'implementazione del metodo, ora hanno un'influenza sull'interfaccia; gli utenti che chiamano il metodo possono "vedere all'interno".

Le classiche macro `blackbox` possono alterare i corpi dei metodi come desiderano e, con l'aiuto di `transparent inline`, diventano quasi "greybox", essendo in grado di generare essenzialmente un tipo di ritorno arbitrario.

Se si riuscisse a fare riferimento al tipo di ritorno inferito di un metodo `inline`, allora si avrebbe una macro che potrebbe generare tipi utilizzabili. Purtroppo, ad oggi, non è possibile scrivere qualcosa come `ReturnTypeInfo[method]`, né qualcosa come `typeOf[result]`.

4.6 Migrazione da Scala 2 a Scala 3

La migrazione, in generale, da Scala 2 a Scala 3 è abbastanza semplice:

- Scala 2.13 e Scala 3 sono per lo più compatibili a livello binario, fatta eccezione solo di alcuni casi più angolari.

Eccezione per le Macros!

- Scala 3 non può espandere Scala 2 Macros
- Scala 2 non può espandere Scala 3 Macros

In questi casi perciò, è necessario **riscrivere** le macro, utilizzando le features apposite di Scala 3 Macros, in nuovi sorgenti che saranno compilati dal nuovo compiler proprio di Scala 3.

4.7 Migrazione da Scala 2 a Scala 3 Macros

Per quel che riguarda, nello specifico, la riscrittura delle macro da Scala 2 a Scala 3 i passi da seguire sono i seguenti:

- La prima regola è quella di *non usare una macro!*
Con l'avvento di Scala 3 sono state introdotte tantissime funzionalità e quello che in Scala 2 era una macro ora potrebbe essere disponibile come feature propria del linguaggio. Perciò, tantissimi casi d'uso delle macro di Scala 2, in Scala 3 scompaiono.
- Viceversa, con i campi che rimangono coperti dalle Macro di Scala 3, è possibile:
 1. Come prima soluzione, utilizzare il costrutto **`inline`**, il quale copre un'ampia gamma delle applicazioni delle Macro di Scala 3.
 2. Se non dovesse essere sufficiente, è possibile utilizzare **`scala.quoted`**; la vera definizione delle Macro di Scala 3.
- Se neanche questi nuovi costrutti dovessero essere sufficienti, è possibile "scendere" e manipolare direttamente gli ASTs e utilizzare la **`reflection`**.
- In alcuni casi è possibile utilizzare dei **`plugin`**, di solito quando il caso d'uso è troppo complesso per essere gestito all'interno di una Macro.

4.8 Inline

Inline method

```
inline def assert(cond:Boolean, msg: =>String):Unit =  
    if !cond then throw new AssertionError(msg)  
  
assert(x==1, "Expected x to be 1 but was "+x)  
  
val cond = myCondition()  
def msg = "Expected x to be 1 but was "+x  
  
if !cond then throw new AssertionError(msg)
```

Figura 22: Inline method

Quando un metodo è dichiarato come *inline*, il compilatore sostituisce tutte le sue invocazioni, nel codice sorgente, con il suo corpo, come se fosse "copiato" al posto della chiamata. Questa tecnica viene utilizzata principalmente per ottimizzazioni in fase di compilazione, evitando la chiamata effettiva del metodo durante l'esecuzione.

Nell'esempio mostrato in figura, viene riportata la definizione del metodo `assert` dichiarato come `inline` (come da standard library).

E' possibile vedere il passaggio di due parametri: `cond` e `msg`, quest'ultimo passato *by-name* (`=>`), il che significa che la sua valutazione è lazy, non viene effettuata immediatamente ma solo quando (e se) viene utilizzata. Questo approccio evita calcoli inutili se la condizione `cond` è vera e l'asserzione non fallisce.

In generale, l'utilizzo di `inline` porta diversi vantaggi, tra cui:

- **Ottimizzazione delle prestazioni:** Per metodi molto semplici, come quelli che eseguono operazioni aritmetiche o accedono a valori costanti, l'inlining riduce il costo della chiamata del metodo.
- **Espressioni condizionali più efficaci:** E' possibile usare `inline` in combinazione con costrutti condizionali, come `if` e `match`, dove il compilatore ottimizzerà la scelta dell'espressione da eseguire, eliminando i rami di codice non necessari.
- **Integrazione con metaprogrammazione:** Le funzioni `inline` possono essere utilizzate con quote e splice per abilitare il metaprogramming, cioè manipolare espressioni durante la compilazione.

Bisogna però fare attenzione a non abusarne!

- **Aumento della dimensione del codice:** Dato che il codice viene copiato ogni volta che viene invocato un metodo inline, un uso eccessivo di metodi inline può portare ad un aumento esponenziale della dimensione del codice.

Inoltre, i metodi inline non possono contenere certe operazioni, come definizioni di variabili locali, cattura di variabili locali esterne o chiamate ricorsive (a meno che non siano tail-recursive).

Inline Parameters

```
inline def assert(inline cond: Boolean, inline msg: String): Unit =
  if !cond then throw new AssertionError(msg)

assert(x == 1, "Expected x to be 1 but was " + x)

if !(x==1) then
  throw new AssertionError("Expected x to be 1 but was " + x)
```

Figura 23: Inline parameters

Entrambi i parametri sono ora definiti come **inline parameters**:

- Qualsiasi espressione passata viene duplicata in ogni punto in cui è utilizzata nella funzione.
- Le espressioni inline sono espanse dal compilatore.
- I parametri inline si comportano in modo simile ai parametri by-name, ossia, non vengono valutati prima della chiamata, ma solo quando effettivamente utilizzati.
- A differenza dei parametri by-name, i parametri inline richiedono la duplicazione del codice
- **Utilità nelle macro:** Nelle macro, i parametri inline consentono di evitare preoccupazioni riguardo la valutazione degli argomenti, poiché il codice viene inserito direttamente nel punto in cui è necessario. Questo può migliorare l'efficienza del codice risultante, poiché non rimangono tracce di variabili temporanee o calcoli intermedi nel programma compilato.

Inline Semantics

1. In Scala, il compilatore esegue prima il typecheck per verificare che il codice sia valido e, solo successivamente, avviene l'inlining, dove il corpo della funzione inline viene inserito direttamente al posto della chiamata alla funzione; questo lo rende quasi equivalente ad una normale chiamata a funzione.

2. Dalla prospettiva di chi utilizza una funzione inline, non ci sono differenze visibili rispetto alla chiamata di una funzione non inline. Il comportamento è lo stesso; l'inlining, infatti, è un'ottimizzazione a livello di compilazione, non a livello di esecuzione.
3. Se durante la compilazione il compiler non riesce a soddisfare i requisiti per l'inlining (ad esempio, per via di limitazioni tecniche come la ricorsione o l'uso di strutture troppo complesse), l'inlining potrebbe fallire. A quel punto, il compilatore non effettuerà l'inline della funzione, e la chiamata verrà trattata come una normale chiamata a funzione, senza segnalare alcun errore.

Inline Features

Inline supporta diverse features tra cui:

1. Inline recursive.
2. Override di un metodo "normale" con un metodo inline.
3. Abstract Inline definition.
4. Inline parameters.
5. Inline conditionals (come Inline If): assicurano che, a compile-time, i branches che non sono valutati o sono valutati parzialmente, vengano eliminati.
6. **scala.compiletime**: contiene diversi metodi per la metaprogrammazione senza il bisogno di addentrarsi nelle Macro vere e proprie. Essi consistono in Macro già implementate, che possono essere utilizzate nelle definizioni inline.
7. Transparent inline (le nuove "whitebox").
8. Supporta le Macro.

scala.compiletime

Nell'esempio sottostante, la chiamata a `zero("AnyVal")` causa un *errore di compilazione* perché la stringa "AnyVal" non è né "Int" né "Long". La funzione `compiletime.error` genera un errore, impedendo la compilazione del codice.

```
transparent inline def zero(inline tpe:String): Int | Long =
  if tpe == "Int" then 0
  else if tpe == "Long" then 0L
  else compiletime.error("No type checked")

val a:Int = zero("Int")
val b:Long = zero("Long")
zero("AnyVal") //err
```

Figura 24: e.g. scala.compiletime

- *inline*: il codice della funzione viene "espanso" nel punto in cui viene chiamato, consentendo ottimizzazioni a livello di compilazione.
- *transparent*: il compiler può dedurre esattamente il tipo di ritorno in base alla logica del codice. In questo caso, se si passa "Int", il compiler sa che il tipo restituito sarà Int, mentre se si passa "Long", il tipo sarà Long.

Questo è utile per garantire che le chiamate alla funzione *zero* siano sicure e che il tipo restituito sia correttamente gestito in base al valore del parametro *tpe*, senza dover eseguire controlli a runtime. Il compilatore si occupa di tutto durante la compilazione, migliorando sia la sicurezza del tipo che l'efficienza del codice.

4.9 Inline vs Macros

Una differenza fondamentale tra l'Inlining e le Macro è il modo in cui vengono valutati.

- L'Inline funziona con la riscrittura del codice e l'esecuzione di ottimizzazioni in base alle regole già note al compilatore.
- D'altro canto, una Macro esegue codice scritto dall'utente che genera il codice in cui si espande la Macro.

Tecnicamente, la compilazione del codice inline `inspectCode('x)` chiama il metodo `inspectCode` in fase di compilazione (tramite la `reflect` di Java) e il metodo viene quindi eseguito come codice "normale".

Per poter eseguire `inspectCode`, è necessario prima compilare il suo codice sorgente. Come conseguenza tecnica, **non è possibile definire e utilizzare una Macro nella stessa classe/file**.

Tuttavia, è possibile avere la definizione della Macro e il relativo richiamo nello stesso progetto purchè sia possibile compilare prima l'implementazione della Macro.

Inline	Macros
Short code	Explicit code
Interpreted	Compiled
Limited functionality	Arbitrary code

Figura 25: Inline vs Macros

Inline

1. Inline fornisce short code che necessita di essere "interpretato" dal compilatore per essere inserito direttamente nel codice chiamante.
2. Ha funzionalità limitate poiché il compilatore deve essere in grado di eseguire o valutare il codice durante la compilazione. Ad esempio, non è possibile gestire tutti i casi di metaprogrammazione complessa.

Macros

1. Dall'altra parte, le Macro sono progettate per avere codice più esplicito: le Macro permettono una **maggiore flessibilità** e **potenza** rispetto alle funzioni inline, poiché possono generare codice più complesso e manipolare l'AST (Abstract Syntax Tree) direttamente.
2. L'idea è quella di mostrare chiaramente la differenza tra il codice che sarà valutato a compile-time, rispetto al codice che sarà generato a runtime; la chiave risiede nel fatto che le Macro possono manipolare il codice in modo più complesso rispetto all'Inlining, consentendo ottimizzazioni e trasformazioni più profonde.
3. Le Macro compilate saranno più veloci rispetto all'Inlining; questo codice compilato non richiede ulteriori espansioni o interpretazioni durante l'esecuzione del programma.
4. Inoltre, supportano codice arbitrario. Le Macro possono eseguire operazioni molto complesse, inclusa la generazione di codice arbitrario. Questo è uno dei punti di forza delle Macro rispetto a Inline, che sono più limitate in termini di capacità di manipolazione del codice.

4.10 Macro = Inline + Quotes + Splice

```
inline def assert(c:Boolean, inline m:String):Int=
  ${ code('c, 'm) }
```

Figura 26: Macro definition

```
def code(c:Expr[Boolean],m:Expr[String])(using Quotes): Expr[Int]= ???
```

Figura 27: Macro implementation

Quotes

Rappresenta il contesto in cui vengono effettuate tutte le operazioni proprie delle Macro ("entry point"). Una quote è un modo per racchiudere un blocco di codice in Scala, trasformandolo in una rappresentazione AST (Abstract Syntax Tree) che può essere manipolata a compile-time. Questo codice racchiuso viene trattato come un valore di tipo Expr[T].

In pratica, una quote viene scritta con la sintassi '{ <codice Scala>}', dove il codice Scala è ciò che verrà "quotato" o racchiuso come espressione.

Le quotes sono essenziali per scrivere Macro in Scala 3. Permettono di catturare frammenti di codice e manipolarli a compile-time. Questi frammenti possono poi essere *spliced* (inseriti) in altri contesti usando il meccanismo di splicing `${...}`.

Quoting e Splicing

- **Quoting:** Converte un'espressione di codice in un valore Expr usando '{...}'.
- **Splicing:** Inserisce un valore Expr all'interno di una quote usando `${...}`. È il processo opposto di quoting, in cui un'espressione è inserita nel codice di una macro.

Nel dettaglio:

1. **Quoting** è il processo di trasformazione di un blocco di codice in un oggetto di tipo Expr[T], che rappresenta un'espressione tipizzata del linguaggio Scala a livello di AST (Abstract Syntax Tree). Questo permette di manipolare il codice *come un dato* durante la compilazione.


```
def example(using Quotes): Expr[Int] = '{
  val x = 3
  x + 1
}
```

Figura 28: Quoting example

- **'{...}**: Modalità di creazione di un Expr. In questo caso, l'espressione `{ val x = 3; x + 1 }` viene trasformata in un AST che rappresenta il codice Scala all'interno delle parentesi.
 - **Expr[T]**: È il tipo generico che rappresenta l'espressione a livello di AST. Nell'esempio, `Expr[Int]` significa che l'espressione rappresenta un valore di tipo `Int`.
2. **Splicing** è il processo inverso di quoting; permette l'inserimento di un'espressione `Expr[T]` all'interno di un'altra quote. Si utilizza il simbolo `${...}` per inserire il contenuto dell'Expr nel codice con quote.

```
inline def addOne(x: Int): Int = ${ addOneImpl('x) }
def addOneImpl(x: Expr[Int])(using Quotes): Expr[Int] = '{
  $x + 1
}
```

Figura 29: Splicing example

- **\${...}**: Nell'esempio, `${ addOneImpl('x) }` congiunge (da cui "splicing") l'implementazione della Macro `addOneImpl` nel punto in cui viene chiamata.
- **'x**: Crea una quote per l'argomento `x` passato alla Macro, trasformandolo in un `Expr[Int]`.
- **\$x**: All'interno della quote, `$x` inserisce l'AST rappresentato da `x` nel punto in cui appare nel codice.

Definizione di una Macro

Per definire una Macro in Scala 3, si utilizza il concetto di inline e quoting/splicing. Una funzione Macro è definita con la parola chiave inline e può accedere e manipolare il codice passato come parametro attraverso quoting e splicing.

```
inline def myMacro(inline x: Int): Int = ${ myMacroImpl('x) }

def myMacroImpl(xExpr: Expr[Int])(using Quotes): Expr[Int] = {
  val xValue = xExpr.valueOrAbort
  '{ $xExpr + 10 }
}
```

Figura 30: Macro example

1. **Quoting:** Nella funzione `myMacroImpl`, `xExpr` è un `Expr[Int]` che rappresenta il codice passato alla macro.
2. **AST Manipulation:** `xExpr.valueOrAbort` consente di ottenere il valore intero da `xExpr` allo scopo di manipolare l'AST aggiungendo 10 al valore.
3. **Splicing:** Viene restituito un nuovo `Expr[Int]` creato con quoting e splicing.

Esecuzione della macro

Alla chiamata della Macro, il compilatore esegue il quoting dell'espressione passata, manipola l'AST attraverso la funzione `myMacroImpl`, e infine esegue lo splicing del risultato nel codice finale.

```
class testmacro_es:
  @Test def TestMacros() =
    assertEquals(myMacro(5), 15)
```

Figura 31: Macro execution

Il tutto avviene sempre a compile-time.

Il processo di creazione e utilizzo di una Macro in Scala 3 coinvolge:

- *Quoting*: per trasformare il codice in un AST
- *Manipolazione dell'AST*: per generare nuovo codice
- *Splicing*: per convertire l'AST in codice ed inserire questo nuovo codice nel punto di utilizzo

Questo consente di scrivere codice generico e riutilizzabile che viene espanso dal compilatore al momento della compilazione, rendendo così il codice più flessibile e potente.

```
myMacro(5)
|
|---> Quoting: '5 --> Expr[Int] (AST per il numero 5)
|
|---> Manipolazione dell' AST: 5 + 10 --> Expr[Int] (AST per 15)
|
|---> Splicing: ${15} --> Codice Scala
|
V
Risultato: 15
```

Figura 32: Macro step

4.11 Quoted package

```
package scala.quoted
/** Code representation for an expression of type T */
trait Expr[+T]
/** Code representation for the type of T */
trait Type[T <: AnyKind]
/** Context needed to operate on code */
trait Quotes:
  /** Reflection API */
  object reflect
```

Figura 33: Quoted package

- Expr[T]: contenente codice non-valutato
- Type[...]: può essere utile per i generici

Expr

```
def boolStr(x:Expr[Boolean])(using Quotes): Expr[String] =  
  x.value match  
    case None =>  
      Expr("?")  
    case Some(bool) =>  
      Expr(bool.toString)
```

Figura 34: Expr example

Nell'esempio, si riceve una expression di tipo Boolean, che contiene codice per una valutazione di tipo booleano. La funzione vuole restituire il valore dell'espressione:

- *x.value*: ritorna un Option di Boolean
- se non lo trova, restituisce una nuova espressione che conterrà la stringa "?"
- se lo trova, restituirà invece il valore di true o false convertito in stringa

4.12 Quotes '{...}'

A partire dall'esempio precedente, è possibile ottenere una versione più complessa usando le quotes.

```
def boolStr(x:Expr[Boolean])(using Quotes): Expr[String] =  
  x.value match  
    case Some(true) => '{ "true" }  
    case Some(false) => '{ "false" }  
    case None => '{ $x.toString }
```

Figura 35: Quote example

E' possibile effettuare il quoting di un frammento di codice e quel codice sarà inserito all'interno dell'Expr.

Quasi-quotation

Seguono le stesse regole dell'interpolazione di stringhe ma con qualche aggiunta in più.

```
s"Hello $world!"  
s"Hello ${world}!"  
  
'{val x = $expr;x}  
'{val x = ${expr};x}  
  
${code('x')}  
${code('{x}})
```

Figura 36: Quasi-quotes example

Vengono mantenute anche nelle Macro in Scala 3.

La prima coppia di stringhe rappresenta la sintassi tipica di una quasi-quote, molto simile all'interpolazione di stringhe.

Entrambe producono lo stesso output se "world" contiene una stringa.

Le parentesi sono opzionali quando si effettua l'interpolazione di una singola variabile o espressione, mentre sono necessarie per espressioni più complesse.

Nel secondo snippet, invece, si confrontano le 2 quotes, uguali, dove viene inserito un frammento di codice all'interno di uno più grande.

Nel terzo, infine, viene mostrato qualcosa in più: se si ha lo splice, si può inserire una quote all'interno dello splice e la sintassi diventa simmetrica.

Illegal reference

Ovviamente, non è possibile referenziare queste variabili in qualsiasi modo; esistono alcune *illegal references* che devono essere evitate.

```
/** Past Variable */  
var x = 1  
'{ x = 2 } //err  
  
/** Future Variable */  
'{  
  var y = 3  
  ${ y = 4; ... } //err  
}
```

Figura 37: Illegal references

1. **Past Variables:** come mostrato nell'esempio, viene definita una variabile `x` e nella quote si cerca di assegnare alla stessa variabile un nuovo valore. Il problema risiede nel "dove" e nel "quando" avviene questo re-assegnamento.

- (a) "var x": è eseguita mentre avviene la compilazione della Macro o mentre viene compilata ed eseguita la Macro
- (b) Il contenuto della quote, però, verrà eseguito nell'applicazione generata, che potrebbe non risiedere nella stessa macchina
- (c) Quello mostrato, è chiaramente un errore di riferimento alla variabile e deve essere evitato

2. **Future Variables:** sono esattamente l'opposto

- (a) Come mostrato nell'esempio, si cerca di effettuare un assegnamento a *y*, la quale sarà definita solo successivamente
- (b) *y* esisterà solo nel codice compilato
- (c) Nell'esempio, si cerca di effettuare un assegnamento *prima* che il codice sia creato

Quotes context object

Un'implementazione di una macro deve *sempre* avere un'istanza di **scala.reflect.Quotes** disponibile.

Quotes è un **context object** con due scopi:

1. Definisce lo **scope dei tipi** dell'API di reflection TASTy (Typed Abstract Syntax Tree). Poiché l'API TASTy è composta interamente da tipi membro astratti, è possibile accedervi solo come tipi dipendenti dal percorso. L'oggetto Quotes rappresenta quel percorso:

```
private def myMacroImpl(using q: Quotes): q.reflect.Term
```

- "Term" is *path-dependent* on *q*

Si applicano le solite avvertenze per lavorare con tipi path-dependent: è necessario tenere traccia del tipo singleton *q.type* affinché il compiler possa ricordare che i tipi sono compatibili.

Come riporta la documentazione ufficiale, non è possibile salvare Quotes in un campo. Se si desidera estrarre metodi di utilità in una classe separata, questa deve essere parametrizzata dal tipo singleton `q.type`, al fine di memorizzare il percorso:

```
class MacroUtilMethods[Q <: Quotes](using q: Q) {  
  def myUtilMethod(): Q.reflect.Term = ???  
}  
  
given q: Quotes = ???  
val utils = new MacroUtilMethods[q.type](using q)  
val term: q.reflect.Term = utils.myUtilMethod()
```

Figura 38: Macro utility methods

2. Quotes cattura il **contesto dell'espansione** della Macro. Questo include la tabella dei simboli del compilatore, che fornisce:
 - (a) Le informazioni su tutti i simboli noti nello scope in cui la Macro verrà espansa (tramite metodi come *Symbol.requiredClass(String)* o *Expr.summon[T]*).
 - (b) La posizione del codice sorgente del sito di chiamata della Macro, tramite *Position.ofMacroExpansion*, che può essere utilizzata insieme ai vari metodi di segnalazione per mostrare errori e avvertimenti a una linea specifica nell'IDE.
 - (c) Il proprietario del codice generato tramite *Symbol.spliceOwner*

4.13 Scala Quoted Type

Scala Quoted Type è molto utile nella scrittura dei Generici o di codice che usa tipi di dato astratti, che non sono conosciuti staticamente a compile-time.

1. Generics

```
inline def timed[T](inline x:T) : (T,Long) = ${ code[T]('x) }

def code[T: Type](x: Expr[T])(using Quotes) =
  '{
    val startTime = System.currentTimeMillis()
    val result:T = $x
    val endTime = System.currentTimeMillis()
    (result, endTime-startTime)
  }
```

Figura 39: Generics example

Nell'esempio, è definita una funzione *timed* su un tipo generico *T*.

All'interno dell'implementazione della macro, il tipo *T* è accompagnato da un *context bound* (**: Type**) il quale assicura che l'informazione del tipo sia disponibile durante la generazione del codice.

Questo permette di utilizzare *T* all'interno di una quote, riferendosi direttamente al tipo stesso e generando il codice necessario che opera su *T*. Il tipo *T* viene utilizzato nella quote.

Viene poi effettuato lo splicing nel codice generato per garantire che l'operazione su *T* sia eseguita correttamente durante la compilazione.

2. Code Patter (Pattern Match)

```
def f(expr: Expr[Option[Any]])(using Quotes) =
  expr match
    case '{ None } => ???
    case '{ Some($x) } => x:Expr[Any]
    case _ => ???
```

Figura 40: Pattern Match example

Viene definita ora una funzione *f* che prende in input un'espressione di *Option[Any]*, a cui viene effettuato il match per valutare se contiene esplicitamente *None* o *Some*; tutto quello che non può essere staticamente riconducibile a questi due costrutti, rientra nel default case.

Nel caso di *Some(x)*, l'espressione splice *\$x* permette di estrarre *x* come *Expr[Any]*, che rappresenta l'espressione contenuta in *Some*.


```
def f2[T:Type](expr: Expr[Option[Any]])(using Quotes) =
  expr match
    case '{ Some($n:Int) } => n:Expr[Int]
    case '{ Some($x: T) } => x:Expr[T]
    case '{ Some($y: t) } => y:Expr[t] //given Type[t] is made available
```

Figura 41: Pattern Match example

E' possibile definire anche un pattern match più complesso che analizza i tipi.

Nell'esempio presentato, è definita una funzione *f2* che prende in input un'espressione (Expr) di tipo Option[Any]. Utilizza il pattern matching per verificare se l'espressione rappresenta un valore Some e, in tal caso, quale tipo di valore è contenuto.

- Nel primo caso, si effettua il matching su un Some contenente un valore Int.
- Nel secondo caso, si utilizza un tipo generico T per effettuare il matching su un Some contenente un valore di tipo T, dove T è un tipo che può variare e non è necessariamente noto a tempo di compilazione.
- Nel terzo caso, il pattern matching è ancora più flessibile, catturando qualsiasi tipo t che potrebbe non essere noto staticamente. Grazie all'inferenza dei tipi di Scala, viene fornita automaticamente l'informazione del tipo t a tempo di compilazione. Questo permette di estrarre y come Expr[t], rendendo possibile la manipolazione di espressioni di tipi non conosciuti a livello di compilazione.

4.14 Macro APIs

Il mondo delle Macro è diviso in due parti:

1. Da una parte, il lato *ordinato e rigido*, type-checked, ben definito con **quotation e splice**.
2. Dall'altra parte, il lato *dinamico ma flessibile* della **riflessione TASTy** (Typed Abstract Syntax Tree).

Si analizzano ora nel dettaglio:

1. La nuova funzione di quotation di Scala 3 (`'{}`) può essere utilizzata, durante la fase di compilazione, per catturare il codice ed esprimerlo come un valore `Expr[E]` al fine di renderlo manipolabile. Al codice di altre espressioni, sotto forma di valori `Expr[E]`, può essere effettuato lo splice in un blocco con quote, attraverso `${}`, in modo simile all'interpolazione di stringhe.

```
val condition: Expr[Boolean] = '{
  val random = newRandomInt()
  random == 123
}

val doPrint: Expr[Unit] = '{
  if ${ condition }
  then println("yes")
  else println("no")
}
```

Figura 42: Macro APIs

Le quotes possono produrre solo un'espressione come risultato, *non* una dichiarazione.

Sebbene sia possibile dichiarare nuove classi in un blocco con quote, queste ultime non saranno visibili all'esterno, similmente ad una classe locale dichiarata nel blocco di un corpo di funzione che non sarà visibile all'esterno.

Mentre `Expr[E]` rappresenta il codice di un valore di tipo `E`, `Type[T]` rappresenta un riferimento ad un tipo `T`.

given `Type[T]`, nello scope, consente di utilizzare il tipo generico `T`, attraverso l'ausilio delle quotes:

```
def myMacro[T](using Type[T]): Expr[Unit] =  
  '{ (callSomeOtherMethod[T]): Unit }
```

Figura 43: `using Type[T]`

In alcuni casi, è possibile convertire direttamente un `Expr[T]` in un valore di tipo `T` usando **`Expr.valueOrAbort`**.

Tale valore deve attraversare il confine tra "code-that-creates-value" e valore a runtime e perciò, è possibile solo se è disponibile un'istanza della type class *`FromExpr[T]`* al fine di effettuare questa conversione, che avviene principalmente solo per i tipi primitivi.

Sia le espressioni a cui è stato effettuato il quoting, che i costruttori di tipo quote, possono essere "matchati" al fine di destrutturarli. Questo tipo di matching è particolarmente utile quando si vuole far riferimento a un `Type[?]` all'interno del codice appartenente al quote. Purtroppo però, questo matching non funziona in tutti i casi.

A partire da Scala 3.4.0, il pattern matching per Types (SIP-53) è stato migliorato per poter eseguire alcune operazioni anche inline; nonostante ciò, è ancora un po' instabile e non sempre funzionante.

A volte si vogliono utilizzare i *given* per un tipo generico in un blocco quote. È importante ricordare che viene effettuato il typecheck della quotation all'interno del context in cui è scritta. Gli splices vengono poi semplicemente sostituiti durante l'espansione della macro, senza alcun ulteriore controllo dei tipi. Spesso perciò, è necessario aggiungere esplicitamente una ricerca differita nel punto di espansione della macro, usando *`Expr.summon`* al fine di individuare le istanze del tipo dato.

`Expr[E]` e `Type[T]` sono *opachi*, ossia, non è consentito vedere cosa c'è "dentro". E' possibile effettuare il quote o lo splice, ma *non* ispezionare il codice.

E' importante evidenziare che poiché deve essere effettuato il typecheck del codice all'interno di una quote, molte espressioni non possono essere istanziate all'interno di una quote. Ad esempio, non è possibile definire un tipo generico `A` poiché non è possibile sapere se questo tipo sarà aperto e costruibile.

```
given Type[A] = ...
'{ new A().asInstanceOf[A] }
```

Figura 44: InstanceOf

In questi casi, diventa necessario "scendere" ad un livello inferiore e costruire manualmente gli alberi delle espressioni attraverso l'ausilio della riflessione TASTy.

2. **TASTy** (Typed Abstract Syntax Tree) è un formato intermedio che rappresenta la struttura di un programma Scala con tutte le informazioni di tipo. TASTy è rappresentato in formato binario all'interno del file .tasty, che viene prodotto dal compilatore e non è altro che una gerarchia di classi che rappresenta l'albero sintattico astratto del codice Scala 3.

Attraverso l'ausilio della riflessione TASTy, è possibile, teoricamente, creare qualsiasi tipo di costrutto sintattico: espressioni o dichiarazioni, annotazioni di tipo, persino costrutti incompleti o illegali. E' possibile anche ispezionare, nel dettaglio, il codice contenuto nelle quotes.

Ad oggi, però, l'API TASTy è purtroppo molto carente in termini di documentazione; il modo più comune per procedere è scrivere "codice di esempio" e cercare di replicare l'AST che il compilatore genera per esso.

Inoltre, TASTy è estremamente verboso e difficile da utilizzare.

4.15 Level checking

Al fine di evitare casi di illegalità, esiste un semplice insieme di regole da seguire, denominato *Level checking*, che sancisce i diversi livelli di definizione.

```
⦿ start at level 0
⦿ scope of '{ .. }' : level + 1
⦿ scope of '${ .. }' : level - 1
⦿ Access variables only at the same level
```

Figura 45: Level checking

1. Si parte dalla definizione di costrutti a *livello 0*, livello iniziale.
2. Se si entra nello scope di una quote, allora verrà **incrementato** il livello di 1.
3. Invece, se si entra nello scope di uno splice, il livello verrà **decrementato** di 1.
4. Quindi, è possibile accedere *solo* alle variabili definite nello stesso livello in cui ci si trova.

4.16 Reflection

E' essenzialmente un'estensione della *Quotes API*; fornisce servizi leggermente diversi tra cui:

- *Error reporting.*
- Visualizzazione e creazione di codice basato su AST, con possibilità di *manipolazione*.
- Fornisce informazioni relative ai file che sono stati compilati (utile in certe applicazioni come il testing).

Da notare però, che, accedendovi, vengono meno alcune proprietà statiche di safety.

1. `quotes.reflect.report`

```
def boolStr3(x: Expr[Boolean])(using Quotes): Expr[String] =  
  import quotes.reflect.report  
  x.value match  
    case None =>  
      report.error(  
        "Expected a know value for x but got "+ x.show, x)  
      Expr("?")  
    case Some(bool) => Expr(bool.toString)
```

Figura 46: Reflection example

report è la funzionalità più utilizzata della Reflection e, come suggerisce il nome, consente di catturare sottoforma di report eventuali errori riscontrati.

2. **`quotes.reflect.*`** è la libreria che consente la manipolazione diretta dell'albero sintattico (AST), relativo al codice associato alla quote corrente, all'interno del contesto delle Macro di Scala 3.

Importando *quotes.reflect._*, si ottiene l'accesso a vari strumenti e classi per lavorare con l'AST, come **Tree**, **Term**, e **report**.

Questo permette l'analisi, la costruzione e la modifica dinamica del codice a compile-time, senza dover utilizzare un alias per Quotes.

```
def useReflection(using Quotes) =  
  import quotes.reflect.*  
  val tree: Tree = ???
```

Figura 47: Tree manipulation example

4.17 AST

Di seguito è riportata una frazione della struttura ramificata, con alcuni braches, del tipo Tree.

```
+~Tree~+~ PackageClause
|
+~ Statment ~+~ Import
|           +- Export
|           +- Definition --+- ClassDef
|           |               +- TypeDef
|           |               +- ...
|           +- Term -----+-Ref +- Ident
|                           |     +- Select
|                           +- Literal
|                           +- If
|                           +- Closure
|                           +- ...
+~ TypeTree ~+~ Inferred
|           +- TypeIdent
|           +- ...
+~ Selector ~+~ SimpleSelector
|           +- ...
+~ Signature
+~ Position
...
```

Figura 48: AST structure

- **Tree:** l'AST di Scala 3 è rappresentato da alberi di tipo Tree. Questi alberi contengono nodi che rappresentano espressioni, definizioni, applicazioni di funzioni, selettori di metodi, ecc.

Ad esempio:

- **Statement:** rappresenta tutto ciò che può appartenere ad un blocco/classe o ad una definizione.
- **Term:** rappresenta quello che di solito appartiene ad un'espressione. Tutte le espressioni sono term, ma non tutti i term sono espressioni valide.
- **TypeTree:** rappresenta i tipi presenti nel file sorgente; contiene informazioni aggiuntive come la posizione all'interno del source file.
- **Patterns**
- **TypeRepr:** rappresenta l'informazione che permette realmente di analizzare nel dettaglio il tipo.

- **Selector**: utile con gli import e gli export.
- **Constants**
- ecc.

Quello che risulta più utile e interessante sono le classi **Tree**, **Term**, **TypeTree**, **TypeRepr** e **Symbol**.

- **Tree**, in breve, rappresenta tutto ciò che viene scritto nel codice sorgente, salve alcune eccezioni.
- **Term** è il sottoinsieme di Tree che rappresenta solo le espressioni, ossia, ogni termine che può essere usato come valore (analogamente a Expr[E]). E' possibile, infatti, convertire Term in Expr[E], e viceversa, attraverso i metodi *asTerm* e *asExprOf[E]*.
Le definizioni di classe, e simili, non sono valori, sono dichiarazioni (**Statement**) e non Term.
- **TypeTree** rappresenta i tipi "così come sono scritti" nel codice sorgente. Questo include elementi come la parte destra di una definizione di alias di tipo (*type Foo = <Bar: TypeTree>*) o l'attribuzione di tipo di un'espressione (*<x: Term>: <Foo: TypeTree>*).
- **TypeRepr**, invece, non è estrapolato "così com'è" nel codice sorgente, ma rappresenta semplicemente il fatto che qualcosa abbia un certo tipo, indipendentemente da dove provenga quella tipizzazione.
Sono analoghi a Type[T]. I TypeRepr appaiono spesso insieme ai Symbol, la classe più importante di tutte: *tutto ciò che ha un nome in Scala, ha un Symbol*.
- **Symbol** è un handle per i metadati su un costrutto sintattico e contiene tutte le informazioni utili che si potrebbero richiedere (nome, tipo, parents, metodi dichiarati, ecc).

Usi di **Symbol**:

- Per scoprire quali metodi sono contenuti in una classe, si potrebbe percorrere il suo albero ClassDef, effettuare il pattern matching su tutti i DefDef inclusi, estrarre i loro TypeTree e poi ripetere il processo ricorsivamente per tutti i genitori dichiarati come ClassDef; oppure, molto più semplicemente, è possibile chiamare **Symbol.methodMembers**. Un Symbol può solitamente essere ottenuto con i metodi *TypeRepr.typeSymbol* o *TypeRepr.termSymbol*.
- Un'altra caratteristica importante dei Symbol è che non sono solo un riassunto di una definizione ma, in un certo senso, sono autorevoli sul codice sorgente scritto; queste informazioni vengono utilizzate nella tabella dei simboli del compiler per governare il typechecking e non solo.
Quando si crea una nuova definizione, viene sempre definito per prima

il Symbol corrispondente e successivamente il Tree, che deve conformarsi alla forma del Symbol. Ad esempio, se una macro sta effettuando un cambiamento di un albero ValDef, da `val x: Int = 1`, a `val x: String = "foo"`, ma il Symbol sancisce ancora che `x` è un `Int`, l'operazione produce problemi; i due devono sempre essere allineati!

In Scala 2, non esisteva un'API pubblica per l'AST e gli autori di Macro usavano le classi interne del compilatore per ispezionare il codice. Inutile dire che ciò portava a molti problemi perché queste classi interne cambiavano frequentemente e interrompevano continuamente tutte le librerie delle Macro. Per evitare questo disastro, esponendo il minor numero possibile di dettagli dell'implementazione, la nuova API reflection di Scala 3 è stata creata in modo molto astratto e peculiare, senza permettere l'accesso diretto alle classi. Inoltre, ogni nodo dell'AST è nascosto dietro un alias di tipo e i metodi vengono aggiunti tramite estensioni.

Type test

```
import quotes.reflect.*
val tree: Tree = ???
tree.show //extension method

tree match
  case sel: Select => sel //TypeTest[Tree,Select]
  case Apply(fun,args) => Apply(fun,args)
```

Figura 49: Tree match case

Importando `quotes.reflect.*` si ha accesso alla manipolazione dell'albero. L'esempio riportato in figura, mostra una semplice chiamata al metodo `show` e un pattern match per effettuare un type test sul Tree.

Expr e Trees

```
def fTree(x: Expr[Int])(using Quotes): Expr[Int] =
  import quotes.reflect.*

  val tree: Term = x.asTerm

  val expr: Expr[Int] = tree.asExprOf[Int]

  expr
```

Figura 50: Tree match case

Viene mostrato un semplice esempio di interoperabilità tra `Term` e `Tree`.

4.18 Casi d'uso

In sintesi, le Macro sono il meccanismo più generale per la trasformazione arbitraria del codice a compile-time, rispetto ad altri meccanismi di metaprogrammazione. Tuttavia, scrivere Macro è complesso e richiede una comprensione approfondita del processo di compilazione.

La programmazione di macro è una competenza molto specializzata, necessaria solo per implementare librerie avanzate con poteri apparentemente "magici".

Vengono presentati ora alcuni esempi di casi d'uso delle Macro, al fine di dare un'idea di ciò che è possibile fare con questo strumento e sviluppare una comprensione di base su come gli implementatori di queste librerie avanzate ottengano i loro risultati "magici".

1. Quando si effettua il debugging, è uso comune effettuare delle stampe di valori attraverso il comando *println*.
E' fastidioso, però, dover scrivere il nome della variabile due volte: una volta per stamparne il nome e un'altra per ottenerne il valore.
Questo problema può essere risolto con una funzione *inline*.

```
inline def debug1(inline expr: Any): Unit =  
  println(codeOf(expr) + " = " + expr)  
  
@main def testDebug1(): Unit = {  
  val someVariable = 42  
  // Classico debug  
  println("someVariable = " + someVariable)  
  // Debug con la macro  
  debug1(someVariable)  
}
```

Figura 51: First use case: Debug

- **codeOf(task)**: restituisce una stringa che rappresenta l'intera espressione *inline*. La stringa proviene dall'albero di parsing dell'espressione, non dal codice sorgente.

La funzione codeOf si trova nel pacchetto *scala.compiletime*.

```
@main def tryCodeOf():Unit =  
  val x = 6  
  val y = 7  
  debug1(x * y) // Prints x.*(y) = 42
```

Figura 52: CodeOf output

2. Si suppone ora di voler indagare il tipo dell'espressione. Non esiste una funzione di libreria per farlo. Perciò, è necessario scrivere una Macro.

Le macro consentono la manipolazione degli alberi sintattici:

- Se **e** è un'espressione Scala di tipo **T**, allora **'{e}** è l'AST di tipo **Expr[T]**.
- Esistono metodi per analizzare e generare alberi sintattici.
- Se **s** è un valore di tipo **Expr[T]**, è possibile utilizzare l'espressione che esso rappresenta all'interno di un'espressione Scala, usando la sintassi **\${s}**.
- Le operazioni **'** e **\$** sono chiamate rispettivamente *quote* e *splice*.

L'idea generale, però, è di fare il più possibile utilizzando la sintassi propria di Scala, sfruttando l'API degli alberi sintattici *solo* quando necessario.

Di seguito allora, viene utilizzata la sintassi di Scala per la concatenazione e l'invocazione di `println`.

L'implementazione completa è la seguente:

```
inline def debug1[T](inline arg: T): Unit = ${debug1Impl('arg)}

private def debug1Impl[T : Type](arg: Expr[T])(using Quotes): Expr[Unit] =
  '{println(${Expr(arg.show)} + ": " + ${Expr(Type.show[T])} + " = " + $arg)}
```

Figura 53: Second use case: use AST

- Per ragioni tecniche, *una Macro deve essere chiamata da una funzione inline*, utilizzando uno splice (\$) a top-level ed effettuando il quoting (') dei suoi argomenti. È consuetudine utilizzare il suffisso *Impl* per la implementazione della macro.
- `'arg` è uguale ad `'{arg}`. Non sono necessarie parentesi graffe con un singolo nome di variabile.
- La macro riceve argomenti di tipo *Expr* e necessita di utilizzare un'istanza fornita di *Quotes*.
- La chiamata `arg.show` restituisce una rappresentazione sotto forma di stringa dell'albero sintattico di `arg`. Per inserirlo (splice), è necessario trasformarlo in un *Expr*.
- Allo stesso modo, `Type.show[T]`, che restituisce una rappresentazione sotto forma di stringa del parametro di tipo, viene trasformato in un *Expr*.
- Infine, la variabile `arg`, che è essa stessa un albero sintattico, viene inserita (splice). Ancora una volta, poiché è una singola variabile, non sono necessarie parentesi graffe.

E' possibile valutarne l'output attraverso un semplice test:

```
@Test def DebugMacroTest() = {
  import ASTmanipol.*
  // Test case 1: Test with a simple integer
  val x = 42
  assertEquals(debug1(x),
    println("x: scala.Int = 42 "))

  // Test case 2: Test with a string expression
  val greeting = "Hello, world!"
  assertEquals(debug1(greeting),
    println("greeting: scala.String = Hello, world!"))

  // Test case 3: Test with a more complex expression
  val sum = x + 58
  assertEquals(debug1(sum),
    println("sum: scala.Int = 100"))

  // Test case 4: Test with a method call
  def square(n: Int): Int = n * n
  assertEquals(debug1(square(4)),
    println("square(4): scala.Int = 16"))

  // Test case 5: Test with a boolean expression
  val isPositive = x > 0
  assertEquals(debug1(isPositive),
    println("isPositive: scala.Boolean = true"))
}
```

Figura 54: Second use case: test use AST

3. La funzione *debug1* stampa una singola espressione. Potrebbe essere utile stamparne più di una.
 - Non esiste un modo "semplice" per stampare i tipi.
 - Si evita di stampare le espressioni per i valori letterali; non avrebbe senso stampare "Test description = Test description".

L'implementazione ora, è un po' più complessa:

```
inline def debugN(inline args: Any*): Unit = ${ debugNImpl('args) }

private def debugNImpl(args: Expr[Seq[Any]])(using q: Quotes): Expr[Unit] =
  import q.reflect.*

  val exprs: Seq[Expr[String]] = args match
    case Varargs(es) =>
      es.map { e =>
        e.asTerm match
          case Literal(c: Constant) => Expr(c.value.toString)
          case _ => '{ ${ Expr(e.show) } + " = " + $e }
      }

  '{ println(${ exprs.reduce((e1, e2) => '{ $e1 + ", " + $e2 }) }) }
```

Figura 55: Third use case: Debug N expressions

Come prima, la funzione *debugN* chiama la macro *debugNImpl*, passando gli argomenti attraverso le quote, come **Expr[Seq[Any]]** e utilizza un'istanza fornita di *Quotes*.

- A questo punto, è richiesta una conoscenza più approfondita dei meccanismi interni di Scala.
 - Il parametro *args* è in realtà un'istanza di **Varargs**, che si può scomporre per ottenere i suoi elementi. Ogni elemento è o un valore letterale o un'espressione per cui vengono stampate la descrizione e il valore.
 - Nel primo caso, viene prodotta un'Expr con la rappresentazione a stringa del valore letterale.
 - Nel secondo caso, viene concatenata la rappresentazione a stringa dell'espressione (*e.show*) con la stringa " = ", a compile-time. Viene così prodotta un'espressione che sarà da concatenare con il valore dell'espressione a runtime.
 - La chiamata al metodo *reduce* permette la concatenazione dei valori delle espressioni.
 - Vengono utilizzati quote e splice nidificati per calcolare la concatenazione di *e1* ed *e2*.
 - È buona norma esaminare i bytecode generati per comprendere le complessità dell'espansione della macro.
4. Viene analizzata ora l'espansione che avviene durante il type checking. In particolare, si vuole verificare, a compile-time, che una String contenga solo cifre, prima di essere analizzata come un intero. Un tale controllo richiede l'invocazione di un metodo, come *String.matches*, a tempo di

compilazione, il ch  non   ammissibile con una sola funzione *inline*. Di nuovo,   necessaria una Macro.

Viene definito un tipo *StringMatching[regex]* per Strings letterali che corrispondono a una espressione regolare, anch'essa specificata come una stringa letterale.

Ad esempio, la dichiarazione:

- `val decimal: StringMatching["[0-9]+"] = "1729"`

avr  successo, ma:

- `val octal: StringMatching["[0-7]+"] = "1729"`

non compiler .

Il tipo *StringMatching*   definito come segue:

```
opaque type StringMatching[R <: String & Singleton] = String
```

Figura 56: StringMatching type

Nel companion object, viene definita una conversione implicita:

```
inline given string2StringMatching[S <: String & Singleton, R <: String & Singleton]:  
  Conversion[S, StringMatching[R]] = str =>  
    inline val regex = constValue[R]  
    inline if matches(str, regex)  
    then str.asInstanceOf[StringMatching[R]]  
    else error(str + " does not match " + regex)
```

Figura 57: Implicit Conversion

matches è la chiamata ad una macro:

```
inline def matches(inline str: String, inline regex: String): Boolean =
  ${ matchesImpl('str', 'regex') }

def matchesImpl(str: Expr[String], regex: Expr[String])(
  using Quotes): Expr[Boolean] =
  Expr(str.valueOrAbort.matches(regex.valueOrAbort))
```

Figura 58: Forth use case: String Match

Sarebbe stato più elegante usare quotes e splices `{ $str.matches($regex) }`, ma non può essere eseguito l'inline. Pertanto, il metodo *matchesImpl* converte manualmente gli alberi di espressione in valori, invoca *String.matches* e produce un Expr con il risultato Booleano.

4.19 Compiler e Bytecode

Come accennato precedentemente, è buona norma esaminare i bytecode generati dal compiler per comprendere le complessità dell'espansione di una Macro. Le modalità per ottenere il codice sono le seguenti:

1. **Compilare il codice e visualizzare i passaggi di compilazione**
Per vedere come la Macro viene espansa e compilata, è possibile eseguire il seguente comando nella shell:

```
scalac -Xprint:typer debugN.scala
```

- **-Xprint:typer:** Mostra il risultato dopo il passaggio del typer, ovvero dopo che la macro è stata espansa. È utile per capire come le espressioni vengono trattate internamente.
- Il risultato stampato a video è l'AST generato dal compilatore Scala, dopo il passaggio del typer, ovvero la fase in cui vengono risolti i tipi e il codice inline viene espanso. L'AST rappresenta la struttura sintattica del programma e viene utilizzato dal compilatore per eseguire trasformazioni e ottimizzazioni.

Se invece si vogliono visualizzare tutti i passaggi di compilazione (incluso il codice di bytecode JVM generato), è possibile usare **-Xprint:all** per avere una vista completa:

```
scalac -Xprint:all DebugMacro.scala
```

2. Generare il bytecode

E' possibile utilizzare lo strumento **javap** per visualizzare il bytecode JVM generato:

- E' necessario compilare prima il file con il comando **scalac**:

```
scalac debugN.scala
```

- Successivamente, è possibile analizzare il bytecode del file .class generato:

```
javap -c debugN.class
```

Questo mostrerà il bytecode generato dal compiler Scala.

4.20 Conclusioni

Le **def-macro** devono seguire ferreamente il seguente pattern:

- Il metodo pubblico deve essere **inline** e non deve fare altro che chiamare immediatamente l'implementazione della macro.
- Gli argomenti passati all'implementazione devono essere racchiusi in una **quote** (all'interno dell' implementazione si ha accesso solo agli *Expr* degli argomenti, non direttamente ai valori degli argomenti).
- I parametri possono essere resi inline, il che consentirà l'accesso all'albero di espressioni che ha creato l'argomento nel sito di chiamata.
- L'implementazione della Macro deve sempre avere un'istanza di **scala.reflect.Quotes** disponibile.

Le def-macro, in generale, sono meno potenti dei costrutti Macro di Scala 2, ma il loro vantaggio è che sono *completamente invisibili* all'utente, offrendogli un'ottima esperienza nell'utilizzo.

La ricerca però non si è conclusa con l'avvento di questa nuova tecnologia; sono necessarie ancora alcune migliorie all'implementazione e, soprattutto, ci sono ancora alcuni aspetti che rimangono oscuri.

Di seguito, alcuni problemi che ci si è posti di affrontare nelle nuove implementazioni di Scala e delle Macro, citati nella presentazione:

- L'impossibilità di generare nuovi membri all'interno di una Macro che siano visibili all'utente, siano essi funzioni, variabili o classi.
- L'assenza delle annotation.
- Il fatto di non riuscire a fare riferimento al tipo di ritorno inferito di un metodo inline.

Bibliografia

- [1] Dean Wampler. Programming Scala, Scalability = Functional Programming + Objects, 3rd Edition
- [2] Cay S. Horstmann. Scala for the Impatient, 3rd Edition.
- [3] Eugene Burmako. Scala Macros: Let Our Powers Combine!
- [4] Martin Odersky & Adrian Moors (2018). Preparing for Scala 3

Sitografia

- Scalameta. <https://scalameta.org/>
- Macro Scala 3 - Nicolas Stucki. <https://www.youtube.com/watch?v=BbTZi8siN28&t=1361s>
- Philosophy of Scala macros. <https://www.youtube.com/watch?v=gg2ItgkCP4k>
- Scala 3 Reference: Reflection. <https://docs.scala-lang.org/scala3/reference/metaprogramming/reflection.html>
- Scala 3 Reference: Macros. <https://docs.scala-lang.org/scala3/reference/metaprogramming/macros.html>
- Inoio - Introduction to macros. <https://inoio.de/blog/2024/07/14/scala3-macros-part1/>
- Metaprogramming in Scala. <https://blog.knoldus.com/meta-programming-in-scala-for-self-transforming-code/>
- Metaprogramming in Scala 3: Inline — Let's talk about Scala 3: <https://www.youtube.com/watch?v=J3VRzMvqW6o>
- Scala3 - Compiler Phases : <https://dotty.epfl.ch/docs/contributing/architecture/phases.html>
- Scala 3 Macros: <https://docs.scala-lang.org/scala3/guides/macros/macros.html>